Brrr-Lang Specification v0.4

From Semantics to Silicon:
A Mathematically Rigorous Intermediate Representation
for Multi-Language Static Analysis

Brrr-Machine Project

January 2026

Abstract

Brrr-Lang is a unified intermediate representation designed for multi-language static analysis with mathematically rigorous foundations. This specification grounds all constructs in established type theory, linear logic, effect algebras, and denotational semantics. We provide formal definitions, typing rules, and F* mechanizations for verification.

This document covers Parts I–III: Type Primitives, Type System, and Ownership & Memory, establishing the algebraic foundations upon which the rest of the language is built.

Contents

Part I
Foundations

1. Semantic Domains

This chapter establishes the denotational foundation for Brrr-Lang. All subsequent definitions are grounded in these semantic domains.

1.1 Domain Theory Preliminaries

Definition 1.1 (Complete Partial Order)**.** A *complete partial order* (CPO) is a partially ordered set $(D, \sqsubseteq)$ such that:

1. There exists a least element $\bot \in D$
2. Every ω-chain $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \cdots$ has a least upper bound $\bigsqcup_{i \in \omega} d_i$

Definition 1.2 (Scott-Continuous Function)**.** A function $f : D \to E$ between CPOs is *Scott-continuous* if:

1. f is monotonic: $d \sqsubseteq d' \implies f(d) \sqsubseteq f(d')$
2. f preserves least upper bounds of chains: $f(\bigsqcup_i d_i) = \bigsqcup_i f(d_i)$

Definition 1.3 (Lifted Domain)**.** For any set A, the *lifted domain* $A_\bot$ is:

$$A_\bot = A \cup \{\bot\}$$

with ordering $\bot \sqsubseteq a$ for all $a \in A$, and $a \sqsubseteq a$ (flat domain).

1.2 Value Domain

Definition 1.4 (Base Value Domain)**.** The domain of base values is:

$$\mathsf{Val}_{\mathsf{base}} = \mathsf{Unit} + \mathsf{Bool} + \mathsf{Int} + \mathsf{Float} + \mathsf{String}$$

where each component is a flat lifted domain.

Definition 1.5 (Location Domain)**.** The domain of heap locations is a countably infinite set:

$$\mathsf{Loc} = \mathbb{N}$$

Definition 1.6 (Value Domain)**.** The full value domain is the solution to the recursive domain equation:

$$\mathsf{Val} = \mathsf{Val}_{\mathsf{base}} + \mathsf{Loc} + (\mathsf{Val} \times \mathsf{Val}) + [\mathsf{Val}] + (\mathsf{Env} \to \mathsf{Comp}[\mathsf{Val}])$$

representing: primitives, references, pairs, lists, and closures.

1.3 Environment and Heap

Definition 1.7 (Environment)**.** An environment maps variables to locations:

$$\mathsf{Env} = \mathsf{Var} \rightharpoonup \mathsf{Loc}$$

where $\rightharpoonup$ denotes partial functions with finite domain.

Definition 1.8 (Heap)**.** A heap maps locations to tagged values:

$$\mathsf{Heap} = \mathsf{Loc} \rightharpoonup (\mathsf{Tag} \times \mathsf{Val})$$

where Tag carries runtime type information.

Definition 1.9 (Ownership State)**.** The ownership state tracks resource ownership:

$$\mathsf{OwnState} = \{\mathsf{owns} : \mathcal{P}(\mathsf{Loc}),\ \mathsf{borrows} : \mathsf{Loc} \rightharpoonup \mathsf{BorrowKind}\}$$

where $\mathsf{BorrowKind} = \{\mathsf{Shared}, \mathsf{Exclusive}\}$.

1.4 Computation Domain

Definition 1.10 (Effect Signature)**.** The effect signature is a set of operation symbols:

$$\Sigma_{\mathsf{Eff}} = \{\mathsf{Throw}[], \mathsf{Panic}, \mathsf{IO}, \mathsf{Alloc}, \mathsf{Read}, \mathsf{Write}, \mathsf{Async}, \mathsf{Div}\}$$

Definition 1.11 (Computation Domain)**.** The computation domain is a graded monad indexed by effects:

$$\mathsf{Comp}_\varepsilon[A] = \mathsf{OwnState} \times \mathsf{Heap} \to (A + \mathsf{Exn}) \times \mathsf{OwnState} \times \mathsf{Heap} \times \mathsf{Trace}_\varepsilon$$

where $\mathsf{Trace}_\varepsilon$ records the effects performed.

Definition 1.12 (Graded Monad Operations)**.** The computation domain forms a graded monad with:

$$\begin{aligned} \mathsf{return}_\perp &: A \to \mathsf{Comp}_\perp[A] \\ \mathsf{return}_\perp(a) &= \lambda(s, h).\ (\mathsf{inl}(a), s, h, \epsilon) \end{aligned}$$

$$\begin{aligned} (\gg\!=)_{\varepsilon_1,\varepsilon_2} &: \mathsf{Comp}_{\varepsilon_1}[A] \to (A \to \mathsf{Comp}_{\varepsilon_2}[B]) \to \mathsf{Comp}_{\varepsilon_1 \sqcup \varepsilon_2}[B] \\ (m \gg\!= f)(s, h) &= \begin{cases} (f(a))(s', h') & \text{if } m(s, h) = (\mathsf{inl}(a), s', h', t) \\ (\mathsf{inr}(e), s', h', t) & \text{if } m(s, h) = (\mathsf{inr}(e), s', h', t) \end{cases} \end{aligned}$$

1.5 F* Mechanization

Semantic Domains in F*

```
module BrrrSemantics.Domains

open FStar.FunctionalExtensionality

(* Base types *)
type base_val =
  | VUnit   : base_val
  | VBool   : bool → base_val
  | VInt    : int → base_val
  | VFloat  : float → base_val
  | VString: string → base_val

(* Locations *)
type loc = nat

(* Borrow kinds *)
type borrow_kind = | Shared | Exclusive

(* Ownership state *)
noeq type own_state = {
  owns    : loc → bool;
  borrows : loc → option borrow_kind
}

(* Tags for runtime type info *)
type tag =
  | TagBase    : base_val → tag
  | TagRef     : loc → tag
  | TagPair    : tag
  | TagArray   : tag
  | TagClosure: tag
```

```
(* Forward declaration for recursive domain *)
val value : Type0
val heap  : Type0
val env   : Type0

(* Heap as partial map *)
let heap = loc → option (tag & value)

(* Environment as partial map *)
let env = string → option loc

(* Effect trace *)
type effect_label =
  | EThrow | EPanic | EIO | EAlloc | ERead | EWrite | EAsync | EDiv

type trace = list effect_label

(* Computation result *)
type comp_result (a:Type) =
  | Ok  : v:a → own_state → heap → trace → comp_result a
  | Err : exn:value → own_state → heap → trace → comp_result a

(* Computation monad *)
let comp (a:Type) = own_state → heap → comp_result a

(* Monad operations *)
let return (#a:Type) (x:a) : comp a =
  fun s h → Ok x s h []

let bind (#a #b:Type) (m:comp a) (f:a → comp b) : comp b =
  fun s h →
    match m s h with
    | Ok v s' h' t →
        (match f v s' h' with
         | Ok v' s'' h'' t' → Ok v' s'' h'' (t @ t')
         | Err e s'' h'' t' → Err e s'' h'' (t @ t'))
    | Err e s' h' t → Err e s' h' t
```

2. Type Primitives

This chapter defines the primitive types of Brrr-Lang with full algebraic structure.

2.1 Boolean Type

Definition 2.1 (Boolean Type)**.** The boolean type Bool is the two-element set with standard operations:

$$\mathsf{Bool} = \{\mathsf{true}, \mathsf{false}\}$$

Definition 2.2 (Boolean Semantics)**.**

$$\begin{aligned}
[\![\mathsf{true}]\!] &= 1 \\
[\![\mathsf{false}]\!] &= 0 \\
[\![\neg b]\!] &= 1 - [\![b]\!] \\
[\![a \wedge b]\!] &= \min([\![a]\!], [\![b]\!]) \\
[\![a \vee b]\!] &= \max([\![a]\!], [\![b]\!])
\end{aligned}$$

2.2 Numeric Types

Definition 2.3 (Integer Type Family)**.** The integer type is parameterized by width and signedness:

$$\mathsf{Int}[w, s] \quad \text{where } w \in \{8, 16, 32, 64, 128, \mathsf{Big}\},\ s \in \{\mathsf{Signed}, \mathsf{Unsigned}\}$$

The semantic domain is:

$$[\![\mathsf{Int}[w, \mathsf{Signed}]]\!] = \{n \in \mathbb{Z} \mid -2^{w-1} \leq n < 2^{w-1}\}$$

$$[\![\mathsf{Int}[w, \mathsf{Unsigned}]]\!] = \{n \in \mathbb{Z} \mid 0 \leq n < 2^{w}\}$$

$$[\![\mathsf{Int}[\mathsf{Big}, \_]]\!] = \mathbb{Z}$$

Definition 2.4 (Float Type Family)**.**

$$\mathsf{Float}[p] \quad \text{where } p \in \{16, 32, 64\}$$

with IEEE 754 semantics for the respective precision.

2.3 Unit and Never Types

Definition 2.5 (Unit Type)**.** The unit type has exactly one value:

$$\mathsf{Unit} = \{()\} \qquad [\![\mathsf{Unit}]\!] = \{\star\}$$

Definition 2.6 (Never Type)**.** The never type (bottom type) is uninhabited:

$$\mathsf{Never} = \emptyset \qquad [\![\mathsf{Never}]\!] = \emptyset$$

Never is the initial object in the category of types:

$$\forall \tau.\ \exists! f : \mathsf{Never} \to \tau$$

2.4 String Type

Definition 2.7 (String Type)**.** The string type represents UTF-8 encoded text:

$$\mathsf{String} = \mathsf{List}[\mathsf{Char}] \quad \text{where } \mathsf{Char} = \text{Unicode scalar value}$$

2.5 F* Mechanization

Primitive Types in F*

```
module BrrrSemantics.Primitives

(* Integer width *)
type int_width = | I8 | I16 | I32 | I64 | I128 | IBig

(* Signedness *)
type signedness = | Signed | Unsigned

(* Integer type family *)
type int_type = {
  width : int_width;
  sign  : signedness
}

(* Value bounds *)
let int_min (t:int_type) : option int =
  match t.width, t.sign with
  | IBig, _ → None
  | I8,  Signed   → Some (-128)
  | I8,   Unsigned → Some 0
  | I16, Signed   → Some (-32768)
  | I16, Unsigned → Some 0
  | I32, Signed   → Some (-2147483648)
  | I32, Unsigned → Some 0
  | I64, Signed   → Some (-9223372036854775808)
  | I64, Unsigned → Some 0
  | I128, Signed  → Some (-170141183460469231731687303715884105728)
  | I128, Unsigned→ Some 0

let int_max (t:int_type) : option int =
  match t.width, t.sign with
  | IBig, _ → None
  | I8,   Signed   → Some 127
  | I8,   Unsigned → Some 255
  | I16, Signed   → Some 32767
  | I16, Unsigned → Some 65535
  | I32, Signed   → Some 2147483647
  | I32, Unsigned → Some 4294967295
  | I64, Signed   → Some 9223372036854775807
  | I64, Unsigned → Some 18446744073709551615
  | I128, _ → (* elided for brevity *) None

(* Well-formed integer value *)
let valid_int (t:int_type) (n:int) : bool =
  match int_min t, int_max t with
  | Some lo, Some hi → lo ≤ n && n ≤ hi
  | None, None → true  (* BigInt *)
  | _ → false

(* Float precision *)
type float_prec = | F16 | F32 | F64

(* Unit type - single inhabitant *)
type unit_t = | Unit

(* Never type - no inhabitants *)
type never = | (* empty *)

(* Absurd elimination for Never *)
let absurd (#a:Type) (n:never) : a = match n with

(* Boolean operations with semantics *)
let bool_sem (b:bool) : nat = if b then 1 else 0

let not_sem (b:bool) : bool = not b

let and_sem (a b:bool) : bool = a && b

let or_sem (a b:bool) : bool = a || b

(* Semantic equivalence lemmas *)
val not_sem_correct: b:bool → Lemma (bool_sem (not_sem b) = 1 -
    bool_sem b)
let not_sem_correct b = ()

val and_sem_correct: a:bool → b:bool →
```

```
  Lemma (bool_sem (and_sem a b) = min (bool_sem a) (bool_sem b))
let and_sem_correct a b = ()

val or_sem_correct: a:bool → b:bool →
  Lemma (bool_sem (or_sem a b) = max (bool_sem a) (bool_sem b))
let or_sem_correct a b = ()
```

3. Top and Bottom Types

3.1 The Any Type (Top)

Definition 3.1 (Any Type). Any is the top type—the supertype of all types:

$$\forall \tau.\ \tau <: \mathsf{Any}$$

Semantically, Any is the terminal object (up to iso):

$$[\![\mathsf{Any}]\!] = \coprod_{\tau \in \mathsf{Type}} [\![\tau]\!]$$

a tagged union of all types.

Definition 3.2 (Unknown Type). Unknown is the safe top type requiring runtime checks:

$$\mathsf{Unknown} \cong \mathsf{Any}$$

but with different typing rules—values of type Unknown cannot be used without explicit type narrowing.

3.2 Subtyping Lattice

Theorem 3.3 (Type Lattice). *The types form a bounded lattice under subtyping:*

$$(\mathsf{Type}, <:, \sqcup, \sqcap, \mathsf{Never}, \mathsf{Any})$$

where:

- Never *is the bottom (least element)*
- Any *is the top (greatest element)*
- $\tau_1 \sqcup \tau_2$ *is the least upper bound (join)*
- $\tau_1 \sqcap \tau_2$ *is the greatest lower bound (meet)*

Definition 3.4 (Subtyping Rules).

$$\frac{}{\tau <: \tau}\ \textsc{S-Refl} \qquad \frac{\tau_1 <: \tau_2 \qquad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}\ \textsc{S-Trans} \qquad \frac{}{\mathsf{Never} <: \tau}\ \textsc{S-Bot} \qquad \frac{}{\tau <: \mathsf{Any}}\ \textsc{S-Top}$$

3.3 F* Mechanization

Top and Bottom Types in F*

```
module BrrrSemantics.TopBottom

open BrrrSemantics.Primitives

(* Type representation *)
noeq type brrr_type =
  | TNever    : brrr_type
  | TUnit     : brrr_type
  | TBool     : brrr_type
  | TInt      : int_type → brrr_type
  | TFloat    : float_prec → brrr_type
  | TString  : brrr_type
  | TOption  : brrr_type → brrr_type
  | TArray   : brrr_type → brrr_type
  | TFunc    : brrr_type → brrr_type → effect_row → brrr_type
  | TPair    : brrr_type → brrr_type → brrr_type
  | TAny     : brrr_type
  | TUnknown : brrr_type

and effect_row =
  | EffPure : effect_row
  | EffCons : effect_label → effect_row → effect_row
  | EffVar  : string → effect_row

(* Subtyping relation *)
let rec subtype (t1 t2:brrr_type) : bool =
  if t1 = t2 then true
  else match t1, t2 with
  | TNever, _ → true                (* Bottom subtypes everything *)
  | _, TAny → true                  (* Everything subtypes top *)
  | TOption t1', TOption t2' → subtype t1' t2'
  | TArray t1', TArray t2' → t1' = t2'   (* Arrays are invariant *)
  | TFunc a1 r1 e1, TFunc a2 r2 e2 →
      subtype a2 a1 &&               (* Contravariant in argument *)
      subtype r1 r2 &&               (* Covariant in result *)
      effect_subtype e1 e2           (* Covariant in effect *)
  | TPair a1 b1, TPair a2 b2 →
      subtype a1 a2 && subtype b1 b2
  | _, _ → false

and effect_subtype (e1 e2:effect_row) : bool =
  match e1, e2 with
  | EffPure, _ → true               (* Pure subtypes any effect *)
  | _, EffVar _ → true              (* Variables are upper bounds *)
  | EffCons l1 r1, EffCons l2 r2 → l1 = l2 && effect_subtype r1 r2
  | _, _ → false

(* Subtyping is reflexive *)
val subtype_refl: t:brrr_type → Lemma (subtype t t)
let subtype_refl t = ()

(* Subtyping is transitive *)
val subtype_trans: t1:brrr_type → t2:brrr_type → t3:brrr_type →
  Lemma (requires subtype t1 t2 ∧ subtype t2 t3)
        (ensures subtype t1 t3)
let rec subtype_trans t1 t2 t3 =
  (* Proof by case analysis - elided for brevity *)
  admit()

(* Never is bottom *)
val never_is_bottom: t:brrr_type → Lemma (subtype TNever t)
let never_is_bottom t = ()

(* Any is top *)
val any_is_top: t:brrr_type → Lemma (subtype t TAny)
let any_is_top t = ()
```

Part II

Type System

4. Effect Algebra

The effect system is grounded in algebraic structure: a bounded join-semilattice with row polymorphism.

4.1 Effect Semilattice

Definition 4.1 (Effect Semilattice)**.** The effect algebra is a tuple $(\mathsf{Eff}, \sqcup, \bot)$ where:

- Eff is the set of effect rows
- $\sqcup : \mathsf{Eff} \times \mathsf{Eff} \to \mathsf{Eff}$ is the join operation
- $\bot \in \mathsf{Eff}$ is the unit (pure/empty effect)

satisfying the semilattice laws.

Theorem 4.2 (Semilattice Laws)**.** *For all* $\varepsilon, \varepsilon_1, \varepsilon_2, \varepsilon_3 \in \mathsf{Eff}$*:*

$$\varepsilon \sqcup \varepsilon = \varepsilon \qquad \textit{(Idempotent)} \tag{4.1}$$

$$\varepsilon_1 \sqcup \varepsilon_2 = \varepsilon_2 \sqcup \varepsilon_1 \qquad \textit{(Commutative)} \tag{4.2}$$

$$(\varepsilon_1 \sqcup \varepsilon_2) \sqcup \varepsilon_3 = \varepsilon_1 \sqcup (\varepsilon_2 \sqcup \varepsilon_3) \qquad \textit{(Associative)} \tag{4.3}$$

$$\bot \sqcup \varepsilon = \varepsilon \qquad \textit{(Identity)} \tag{4.4}$$

Definition 4.3 (Effect Subtyping)**.** The effect ordering is derived from the semilattice:

$$\varepsilon_1 \sqsubseteq \varepsilon_2 \iff \varepsilon_1 \sqcup \varepsilon_2 = \varepsilon_2$$

Equivalently: $\varepsilon_1 \sqsubseteq \varepsilon_2$ iff ε_2 has at least the effects of ε_1.

Lemma 4.4 (Derived Order Properties)**.**

$$\varepsilon \sqsubseteq \varepsilon \qquad \textit{(Reflexive)} \tag{4.5}$$

$$\varepsilon_1 \sqsubseteq \varepsilon_2 \wedge \varepsilon_2 \sqsubseteq \varepsilon_3 \implies \varepsilon_1 \sqsubseteq \varepsilon_3 \qquad \textit{(Transitive)} \tag{4.6}$$

$$\bot \sqsubseteq \varepsilon \qquad \textit{(Pure is bottom)} \tag{4.7}$$

$$\varepsilon_1 \sqsubseteq \varepsilon_1 \sqcup \varepsilon_2 \qquad \textit{(Join upper bound)} \tag{4.8}$$

4.2 Effect Row Syntax

Definition 4.5 (Effect Row Grammar)**.**

$$\begin{aligned}
\varepsilon ::= {} & \bot && \text{(pure effect)} \\
\mid {} & \langle E \mid \varepsilon \rangle && \text{(effect extension)} \\
\mid {} & \rho && \text{(effect variable)}
\end{aligned}$$

$$\begin{aligned}
E ::= {} & \mathsf{Throw}[\tau] && \text{(exception)} \\
\mid {} & \mathsf{Panic} && \text{(panic/abort)} \\
\mid {} & \mathsf{IO} && \text{(I/O)} \\
\mid {} & \mathsf{Async} && \text{(async)} \\
\mid {} & \mathsf{Alloc} && \text{(allocation)} \\
\mid {} & \mathsf{Read} && \text{(memory read)} \\
\mid {} & \mathsf{Write} && \text{(memory write)} \\
\mid {} & \mathsf{Div} && \text{(divergence)}
\end{aligned}$$

Definition 4.6 (Row Equivalence)**.** Effect rows are equivalent modulo:

$$\langle E_1 \mid \langle E_2 \mid \varepsilon \rangle \rangle \equiv \langle E_2 \mid \langle E_1 \mid \varepsilon \rangle \rangle \qquad \text{(Row-Comm)} \tag{4.9}$$

$$\langle E \mid \langle E \mid \varepsilon \rangle \rangle \equiv \langle E \mid \varepsilon \rangle \qquad \text{(Row-Idem)} \tag{4.10}$$

4.3 Graded Monad Structure

Definition 4.7 (Graded Monad)**.** The computation type $\tau @ \varepsilon$ (type τ with effect ε) forms a graded monad T_ε with:

$$\eta_\bot : \tau \to T_\bot[\tau]$$
$$\mu_{\varepsilon_1,\varepsilon_2} : T_{\varepsilon_1}[T_{\varepsilon_2}[\tau]] \to T_{\varepsilon_1 \sqcup \varepsilon_2}[\tau]$$

Theorem 4.8 (Graded Monad Laws)**.**

$$\mu \circ T_{\varepsilon_1}[\eta] = \mathsf{id} \qquad \textit{(GM-Right)} \tag{4.11}$$
$$\mu \circ \eta = \mathsf{id} \qquad \textit{(GM-Left)} \tag{4.12}$$
$$\mu \circ \mu = \mu \circ T[\mu] \qquad \textit{(GM-Assoc)} \tag{4.13}$$

with effect indices properly tracked.

4.4 Effect Typing Rules

Definition 4.9 (Effectful Function Type)**.**

$$\tau_1 \xrightarrow{\varepsilon} \tau_2 \triangleq \tau_1 \to T_\varepsilon[\tau_2]$$

Definition 4.10 (Effect Typing Judgment)**.** The judgment $\Gamma \vdash e : \tau\ [\varepsilon]$ means:

Under context Γ, expression e has type τ and may perform effects in ε.

Definition 4.11 (Core Effect Typing Rules)**.**

$$\frac{\Gamma \vdash e : \tau\ [\bot]}{\Gamma \vdash e : \tau\ [\varepsilon]}\ \textsc{E-Pure} \qquad \frac{\Gamma \vdash e : \tau\ [\varepsilon_1] \qquad \varepsilon_1 \sqsubseteq \varepsilon_2}{\Gamma \vdash e : \tau\ [\varepsilon_2]}\ \textsc{E-Sub}$$

$$\frac{\Gamma \vdash e_1 : \tau_1\ [\varepsilon_1] \qquad \Gamma \vdash e_2 : \tau_2\ [\varepsilon_2]}{\Gamma \vdash e_1; e_2 : \tau_2\ [\varepsilon_1 \sqcup \varepsilon_2]}\ \textsc{E-Seq} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{\varepsilon_f} \tau_2\ [\varepsilon_1] \qquad \Gamma \vdash e_2 : \tau_1\ [\varepsilon_2]}{\Gamma \vdash e_1\ e_2 : \tau_2\ [\varepsilon_1 \sqcup \varepsilon_2 \sqcup \varepsilon_f]}\ \textsc{E-App}$$

4.5 F* Mechanization

Effect Algebra in F*

```
module BrrrSemantics.Effects

(* Effect labels *)
type effect_label =
  | EThrow  : brrr_type → effect_label  (* Exception with type *)
  | EPanic  : effect_label
  | EIO     : effect_label
  | EAsync  : effect_label
  | EAlloc  : effect_label
  | ERead   : effect_label
  | EWrite  : effect_label
  | EDiv    : effect_label

(* Effect row - set-like with variable *)
noeq type effect_row =
  | Pure    : effect_row                              (* Bottom: no
     effects *)
  | Extend  : effect_label → effect_row → effect_row  (* Extension *)
  | Var     : string → effect_row                     (* Polymorphic
     variable *)
```

```
(* Membership check *)
let rec has_effect (e:effect_label) (row:effect_row) : bool =
  match row with
  | Pure → false
  | Extend e' rest → e = e' || has_effect e rest
  | Var _ → true   (* Variables may contain any effect *)

(* Effect join (union) *)
let rec effect_join (r1 r2:effect_row) : effect_row =
  match r1 with
  | Pure → r2
  | Var v → Var v   (* Variable absorbs *)
  | Extend e rest →
      if has_effect e r2
      then effect_join rest r2   (* Idempotent: skip duplicate *)
      else Extend e (effect_join rest r2)

(* Effect subtyping: e1 ≤ e2 iff join e1 e2 = e2 *)
let effect_sub (e1 e2:effect_row) : bool =
  effect_join e1 e2 = e2

(* Semilattice laws *)

(* Idempotent: e ⊔ e = e *)
val join_idempotent: e:effect_row →
  Lemma (effect_join e e = e) [SMTPat (effect_join e e)]
let rec join_idempotent e = match e with
  | Pure → ()
  | Var _ → ()
  | Extend l rest → join_idempotent rest

(* Commutative: e1 ⊔ e2 = e2 ⊔ e1 *)
val join_comm: e1:effect_row → e2:effect_row →
  Lemma (effect_join e1 e2 = effect_join e2 e1)
let join_comm e1 e2 = admit() (* Requires careful proof *)

(* Identity: Pure ⊔ e = e *)
val join_identity: e:effect_row →
  Lemma (effect_join Pure e = e) [SMTPat (effect_join Pure e)]
let join_identity e = ()

(* Pure is bottom *)
val pure_is_bottom: e:effect_row →
  Lemma (effect_sub Pure e) [SMTPat (effect_sub Pure e)]
let pure_is_bottom e = join_identity e

(* Graded monad type *)
noeq type comp (a:Type) (eff:effect_row) =
  | MkComp : (unit → a) → comp a eff

(* Return (pure) *)
let return (#a:Type) (x:a) : comp a Pure =
  MkComp (fun () → x)

(* Bind with effect join *)
let bind (#a #b:Type) (#e1 #e2:effect_row)
         (m:comp a e1) (f:a → comp b e2)
```

```
        : comp b (effect_join e1 e2) =
  match m with
  | MkComp run_m →
      MkComp (fun () →
        let x = run_m () in
        match f x with
        | MkComp run_f → run_f ())
```

5. Algebraic Effect Handlers

5.1 Effect Signatures

Definition 5.1 (Effect Signature)**.** An effect signature declares operations with their types:

$$\mathsf{effect}\ E\ \{\ \mathsf{op}_1 : \tau_1 \rightsquigarrow \sigma_1,\ \ldots,\ \mathsf{op}_n : \tau_n \rightsquigarrow \sigma_n\ \}$$

where $\mathsf{op}_i : \tau_i \rightsquigarrow \sigma_i$ means operation op_i takes argument of type τ_i and returns σ_i.

Example 5.2 (Standard Effect Signatures)**.**

$$\begin{aligned}
&\mathsf{effect\ Exception}[E]\ \{\\
&\quad \mathsf{throw} : E \rightsquigarrow \mathsf{Never}\\
&\}\\[1em]
&\mathsf{effect\ State}[S]\ \{\\
&\quad \mathsf{get} : \mathsf{Unit} \rightsquigarrow S\\
&\quad \mathsf{put} : S \rightsquigarrow \mathsf{Unit}\\
&\}\\[1em]
&\mathsf{effect\ Async}\ \{\\
&\quad \mathsf{await} : \mathsf{Future}[\tau] \rightsquigarrow \tau\\
&\quad \mathsf{spawn} : (\mathsf{Unit} \xrightarrow{[\mathsf{Async}]} \tau) \rightsquigarrow \mathsf{Future}[\tau]\\
&\}
\end{aligned}$$

5.2 Handler Definition

Definition 5.3 (Effect Handler)**.** A handler for effect E with operations $\{\mathsf{op}_i\}$ is:

$$\mathsf{handle}\ e\ \mathsf{with}\ \{\mathsf{return}\ x \Rightarrow e_r,\ \mathsf{op}_1(x,k) \Rightarrow e_1,\ \ldots\}$$

where k is the continuation.

Definition 5.4 (Handler Typing)**.**

$$\frac{\Gamma \vdash e : \tau\ [\langle E \mid \varepsilon\rangle] \qquad \Gamma, x : \tau \vdash e_r : \sigma\ [\varepsilon'] \qquad \forall i.\ \Gamma, x_i : \tau_i, k_i : \sigma_i \to \sigma\ [\varepsilon'] \vdash e_i : \sigma\ [\varepsilon']}{\Gamma \vdash \mathsf{handle}\ e\ \mathsf{with}\ H : \sigma\ [\varepsilon \sqcup \varepsilon']}\ \textsc{T-Handle}$$

5.3 Continuation Linearity

Definition 5.5 (Continuation Modes)**.** • **One-shot** (default): Continuation k can be called at most once. Linear resource.

• **Multi-shot**: Continuation k can be called multiple times. Requires copying.

Definition 5.6 (Linearity Constraint)**.** For one-shot handlers, the continuation k has linear type:

$$k : \sigma_i \multimap \sigma\ [\varepsilon']$$

For multi-shot handlers:

$$k : \sigma_i \to \sigma\ [\varepsilon']$$

5.4 Handler Semantics

Definition 5.7 (Operational Semantics).

$$\text{handle } v \text{ with } H \longrightarrow e_r[x := v]$$
$$\text{handle } E[\mathsf{op}(v)] \text{ with } H \longrightarrow e_{\mathsf{op}}[x := v, k := \lambda y.\, \text{handle } E[y] \text{ with } H]$$

where E is an evaluation context not containing handlers for op.

5.5 F* Mechanization

Effect Handlers in F*

```
module BrrrSemantics.Handlers

open BrrrSemantics.Effects

(* Continuation type - parameterized by linearity *)
type linearity = | OneShot | MultiShot

(* Operation signature *)
noeq type op_sig = {
  op_name : string;
  arg_type : brrr_type;
  ret_type : brrr_type
}

(* Effect signature *)
noeq type effect_sig = {
  eff_name : string;
  operations : list op_sig
}

(* Handler clause *)
noeq type handler_clause (a:Type) (b:Type) =
  | ReturnClause : (a → b) → handler_clause a b
  | OpClause : op_sig →
               (arg:Type) → (ret:Type) →
               (arg → (ret → b) → b) →  (* op handler with
                  continuation *)
               handler_clause a b

(* Full handler *)
noeq type handler (a:Type) (b:Type) = {
  return_clause : a → b;
  op_clauses : list (handler_clause a b);
  cont_linearity : linearity
}

(* Free monad for effects *)
noeq type free (eff:effect_sig) (a:Type) =
  | Pure : a → free eff a
  | Impure : op:op_sig{List.mem op eff.operations} →
             arg:Type →
             (ret:Type → free eff a) →
             free eff a

(* Handle operation *)
let rec handle_free (#eff:effect_sig) (#a #b:Type)
                    (h:handler a b) (m:free eff a) : b =
  match m with
  | Pure x → h.return_clause x
  | Impure op arg cont →
      (* Find matching operation clause *)
      let clause = find_op_clause op h.op_clauses in
      match clause with
      | Some (OpClause _ _ _ handler_fn) →
          handler_fn arg (fun ret → handle_free h (cont ret))
      | None →
          (* Forward unhandled operation - would need outer handler *)
          admit()

and find_op_clause (op:op_sig) (clauses:list (handler_clause 'a 'b))
    : option (handler_clause 'a 'b) =
  match clauses with
  | [] → None
  | OpClause op' _ _ _ :: rest →
      if op.op_name = op'.op_name then Some (OpClause op' _ _ _)
      else find_op_clause op rest
  | _ :: rest → find_op_clause op rest

(* Example: Exception effect *)
let exception_sig : effect_sig = {
  eff_name = "Exception";
  operations = [{op_name = "throw"; arg_type = TAny; ret_type = TNever
     }]
}

(* Example: State effect *)
let state_sig (s:brrr_type) : effect_sig = {
  eff_name = "State";
  operations = [
    {op_name = "get"; arg_type = TUnit; ret_type = s};
    {op_name = "put"; arg_type = s; ret_type = TUnit}
  ]
}

(* Run state handler *)
let run_state (#a:Type) (#s:Type) (init:s)
              (m:free (state_sig (TInt I32 Signed)) a) : a & s =
  let h : handler a (s → a & s) = {
    return_clause = (fun x → fun st → (x, st));
    op_clauses = [
      OpClause {op_name="get"; arg_type=TUnit; ret_type=TInt I32
          Signed}
               unit s
               (fun () k → fun st → k st st);
      OpClause {op_name="put"; arg_type=TInt I32 Signed; ret_type=
          TUnit}
               s unit
               (fun new_st k → fun _ → k () new_st)
    ];
    cont_linearity = OneShot
  } in
  handle_free h m init
```

6. Type Polymorphism

6.1 Parametric Polymorphism

Definition 6.1 (Universal Type).

$$\forall \alpha : \kappa.\, \tau$$

where α is a type variable of kind κ.

Definition 6.2 (Polymorphic Typing Rules).

$$\frac{\Gamma, \alpha : \kappa \vdash e : \tau\ [\varepsilon]}{\Gamma \vdash \Lambda\alpha : \kappa.\, e : \forall \alpha : \kappa.\, \tau\ [\varepsilon]}\ \textsc{T-TAbs} \qquad \frac{\Gamma \vdash e : \forall \alpha : \kappa.\, \tau\ [\varepsilon] \qquad \Gamma \vdash \sigma : \kappa}{\Gamma \vdash e\,[\sigma] : \tau[\alpha := \sigma]\ [\varepsilon]}\ \textsc{T-TApp}$$

6.2 Row Polymorphism

Definition 6.3 (Row-Polymorphic Function). A function polymorphic in its effect row:

$$\forall \rho.\, \tau_1 \xrightarrow{\langle E|\rho\rangle} \tau_2$$

Example 6.4 (Row-Polymorphic Map).

$$\mathsf{map} : \forall \alpha\beta\rho.\, (\alpha \xrightarrow{\rho} \beta) \to \mathsf{Array}[[]\alpha] \xrightarrow{\rho} \mathsf{Array}[[]\beta]$$

The function map preserves whatever effects f has.

6.3 Higher-Kinded Types

Definition 6.5 (Kind System).

$$\begin{aligned} \kappa ::= {} & \star && \text{(type)} \\ \mid {} & \kappa_1 \to \kappa_2 && \text{(type constructor)} \\ \mid {} & \mathsf{Row} && \text{(effect row)} \\ \mid {} & \mathsf{Region} && \text{(lifetime/region)} \end{aligned}$$

Example 6.6 (Higher-Kinded Functor).

$$\mathsf{Functor} : (\star \to \star) \to \star$$

$$\mathsf{fmap} : \forall F : \star \to \star.\, \mathsf{Functor}[F] \Rightarrow \forall \alpha\beta.\, (\alpha \to \beta) \to F[\alpha] \to F[\beta]$$

6.4 Variance

Definition 6.7 (Variance Annotations). • **Covariant** (+): $\tau_1 <: \tau_2 \implies F[\tau_1] <: F[\tau_2]$
• **Contravariant** (−): $\tau_1 <: \tau_2 \implies F[\tau_2] <: F[\tau_1]$
• **Invariant** (∘): No subtyping relationship derived
• **Bivariant** (±): Both directions hold

Theorem 6.8 (Function Variance). *For function types $\tau_1 \xrightarrow{\varepsilon} \tau_2$:*

- *Contravariant in τ_1 (argument)*
- *Covariant in τ_2 (result)*
- *Covariant in ε (effect)*

6.5 F* Mechanization

Polymorphism in F*

```
module BrrrSemantics.Polymorphism

(* Kind *)
type kind =
  | KStar   : kind                        (* Types *)
  | KArrow  : kind → kind → kind        (* Type constructors *)
  | KRow    : kind                        (* Effect rows *)
  | KRegion : kind                        (* Lifetimes *)

(* Type with kind annotation *)
noeq type kinded_type =
  | KTVar   : string → kind → kinded_type
  | KTApp   : kinded_type → kinded_type → kinded_type
  | KTArrow : kinded_type → kinded_type → effect_row → kinded_type
  | KTForall: string → kind → kinded_type → kinded_type

(* Kind checking *)
let rec kind_of (ctx:list (string & kind)) (t:kinded_type) : option
    kind =
  match t with
  | KTVar v k →
      if List.mem (v, k) ctx then Some k else None
  | KTApp f arg →
      (match kind_of ctx f, kind_of ctx arg with
       | Some (KArrow k1 k2), Some k1' →
           if k1 = k1' then Some k2 else None
       | _ → None)
  | KTArrow t1 t2 _ →
      (match kind_of ctx t1, kind_of ctx t2 with
       | Some KStar, Some KStar → Some KStar
       | _ → None)
  | KTForall v k body →
      kind_of ((v, k) :: ctx) body

(* Variance *)
type variance = | Covariant | Contravariant | Invariant | Bivariant

(* Combine variances *)
let combine_variance (v1 v2:variance) : variance =
  match v1, v2 with
  | Bivariant, _ → Bivariant
  | _, Bivariant → Bivariant
  | Invariant, _ → Invariant
  | _, Invariant → Invariant
  | Covariant, Covariant → Covariant
  | Contravariant, Contravariant → Covariant
  | Covariant, Contravariant → Contravariant
  | Contravariant, Covariant → Contravariant

(* Flip variance (for contravariant positions) *)
let flip_variance (v:variance) : variance =
  match v with
  | Covariant → Contravariant
  | Contravariant → Covariant
  | _ → v
```

```
(* Variance of type variable in type *)
let rec variance_of (v:string) (t:kinded_type) : variance =
  match t with
  | KTVar v' _ → if v = v' then Covariant else Bivariant
  | KTApp f arg →
      combine_variance (variance_of v f) (variance_of v arg)
  | KTArrow t1 t2 _ →
      combine_variance
        (flip_variance (variance_of v t1))  (* Contravariant in arg *)
        (variance_of v t2)                   (* Covariant in result *)
  | KTForall _ _ body → variance_of v body

(* Example: Option is covariant *)
let option_variance : variance = Covariant

(* Example: Function argument is contravariant *)
let func_arg_variance : variance = Contravariant
```

Part III
Ownership & Memory

7. Mode Semiring

This chapter grounds ownership in linear logic's resource algebra, replacing ad-hoc Rust-inspired modes with a principled mode semiring.

7.1 Mode Definition

Definition 7.1 (Mode Semiring)**.** The mode semiring $(M, +, \cdot, \mathsf{0}, \mathsf{1})$ consists of:

- $M = \{\mathsf{0}, \mathsf{1}, \omega\}$ (absent, linear, unrestricted)
- $+$: mode addition (parallel use)
- $\cdot$: mode multiplication (sequential use)
- $\mathsf{0}$: additive identity (absent)
- $\mathsf{1}$: multiplicative identity (linear)

Definition 7.2 (Mode Operations)**.**

$+$	$\mathsf{0}$	$\mathsf{1}$	ω
$\mathsf{0}$	$\mathsf{0}$	$\mathsf{1}$	ω
$\mathsf{1}$	$\mathsf{1}$	ω	ω
ω	ω	ω	ω

$\cdot$	$\mathsf{0}$	$\mathsf{1}$	ω
$\mathsf{0}$	$\mathsf{0}$	$\mathsf{0}$	$\mathsf{0}$
$\mathsf{1}$	$\mathsf{0}$	$\mathsf{1}$	ω
ω	$\mathsf{0}$	ω	ω

Theorem 7.3 (Semiring Laws)**.** *For all $m, m_1, m_2, m_3 \in M$:*

$$\mathsf{0} + m = m \qquad \textit{(additive identity)} \tag{7.1}$$

$$m_1 + m_2 = m_2 + m_1 \qquad \textit{(additive commutativity)} \tag{7.2}$$

$$(m_1 + m_2) + m_3 = m_1 + (m_2 + m_3) \qquad \textit{(additive associativity)} \tag{7.3}$$

$$\mathsf{1} \cdot m = m \qquad \textit{(multiplicative identity)} \tag{7.4}$$

$$\mathsf{0} \cdot m = \mathsf{0} \qquad \textit{(multiplicative absorption)} \tag{7.5}$$

$$m_1 \cdot (m_2 + m_3) = m_1 \cdot m_2 + m_1 \cdot m_3 \qquad \textit{(distributivity)} \tag{7.6}$$

Definition 7.4 (Mode Ordering)**.** The subusage ordering:

$$\mathsf{0} \leq \mathsf{1} \leq \omega$$

Interpretation: $m_1 \leq m_2$ means mode m_1 can be weakened to m_2.

7.2 Extended Mode Lattice

Definition 7.5 (Extended Modes)**.** For practical use, we extend with affine and relevant modes:

$$M_{\mathsf{ext}} = \{\mathsf{0}, \mathsf{1}, \mathsf{aff}, \mathsf{rel}, \omega\}$$

with lattice structure:

$$\begin{tikzcd}
 & \omega & \\
\mathsf{aff} \arrow[ur] & & \mathsf{rel} \arrow[ul] \\
 & \mathsf{1} \arrow[ul] \arrow[ur] & \\
 & \mathsf{0} \arrow[u] &
\end{tikzcd}$$

Definition 7.6 (Mode Semantics)**.** • $\mathsf{0}$ (absent): Cannot be used

- $\mathsf{1}$ (linear): Must be used exactly once
- aff: May be used at most once (drop allowed)
- rel: Must be used at least once (copy allowed)
- ω (unrestricted): Any number of uses

7.3 Type-with-Mode Syntax

Definition 7.7 (Mode-Annotated Type)**.**

$$\tau@m$$

denotes type τ with usage mode m.

Example 7.8 (Mode Annotations)**.**

$\mathsf{String}@1$	(linear string, must use once)
$\mathsf{Int}@\omega$	(unrestricted int, can copy)
$\mathsf{Handle}@\mathsf{aff}$	(affine handle, must close or drop)

7.4 F* Mechanization

Mode Semiring in F*

```
module BrrrSemantics.Modes

(* Core modes *)
type mode =
  | MZero    : mode     (* 0: absent, cannot use *)
  | MOne     : mode     (* 1: linear, use exactly once *)
  | MOmega   : mode     (* ω: unrestricted, use any number *)

(* Extended modes *)
type mode_ext =
  | MZeroE     : mode_ext
  | MOneE      : mode_ext
  | MAffine    : mode_ext     (* use at most once *)
  | MRelevant : mode_ext     (* use at least once *)
  | MOmegaE    : mode_ext

(* Mode addition (parallel composition) *)
let mode_add (m1 m2:mode) : mode =
  match m1, m2 with
  | MZero, m → m
  | m, MZero → m
  | MOne, MOne → MOmega
  | MOmega, _ → MOmega
  | _, MOmega → MOmega

(* Mode multiplication (sequential composition) *)
let mode_mul (m1 m2:mode) : mode =
  match m1, m2 with
  | MZero, _ → MZero
  | _, MZero → MZero
  | MOne, m → m
  | m, MOne → m
  | MOmega, MOmega → MOmega

(* Mode ordering (can weaken m1 to m2) *)
let mode_leq (m1 m2:mode) : bool =
  match m1, m2 with
  | MZero, _ → true
  | MOne, MOne → true
  | MOne, MOmega → true
  | MOmega, MOmega → true
  | _, _ → false
```

```
(* Semiring law proofs *)

(* Additive identity: 0 + m = m *)
val add_identity: m:mode → Lemma (mode_add MZero m = m)
let add_identity m = ()

(* Additive commutativity: m1 + m2 = m2 + m1 *)
val add_comm: m1:mode → m2:mode → Lemma (mode_add m1 m2 = mode_add m2
   m1)
let add_comm m1 m2 = ()

(* Multiplicative identity: 1 * m = m *)
val mul_identity: m:mode → Lemma (mode_mul MOne m = m)
let mul_identity m = ()

(* Multiplicative absorption: 0 * m = 0 *)
val mul_absorb: m:mode → Lemma (mode_mul MZero m = MZero)
let mul_absorb m = ()

(* Distributivity: m1 * (m2 + m3) = m1*m2 + m1*m3 *)
val distributive: m1:mode → m2:mode → m3:mode →
  Lemma (mode_mul m1 (mode_add m2 m3) = mode_add (mode_mul m1 m2) (
     mode_mul m1 m3))
let distributive m1 m2 m3 =
  match m1, m2, m3 with
  | MZero, _, _ → ()
  | _, MZero, _ → ()
  | _, _, MZero → ()
  | MOne, MOne, MOne → ()
  | MOne, MOne, MOmega → ()
  | MOne, MOmega, MOne → ()
  | MOne, MOmega, MOmega → ()
  | MOmega, _, _ → ()

(* Type with mode *)
noeq type moded_type = {
  ty   : brrr_type;
  mode : mode
}

(* Context is list of moded bindings *)
type moded_ctx = list (string & moded_type)

(* Split context for linear typing *)
let rec ctx_split (ctx:moded_ctx) : option (moded_ctx & moded_ctx) =
  match ctx with
  | [] → Some ([], [])
  | (x, {ty; mode=MZero}) :: rest →
      (match ctx_split rest with
       | Some (l, r) → Some ((x, {ty; mode=MZero}) :: l,
                             (x, {ty; mode=MZero}) :: r)
       | None → None)
  | (x, {ty; mode=MOne}) :: rest →
      (match ctx_split rest with
       | Some (l, r) →
           (* Linear: goes to exactly one side *)
```

```
            Some ((x, {ty; mode=MOne}) :: l, r)  (* or (l, (x,{ty;mode=
                MOne})::r) *)
        | None → None)
  | (x, {ty; mode=MOmega}) :: rest →
      (match ctx_split rest with
        | Some (l, r) → Some ((x, {ty; mode=MOmega}) :: l,
                               (x, {ty; mode=MOmega}) :: r)
        | None → None)
```

8. Linear Type System

8.1 Linear Typing Judgment

Definition 8.1 (Linear Context)**.** A linear context Γ maps variables to mode-annotated types:

$$\Gamma = x_1 : \tau_1@m_1, \ldots, x_n : \tau_n@m_n$$

Definition 8.2 (Context Operations)**.** • **Addition**: $\Gamma_1 + \Gamma_2$ adds modes pointwise
- **Scaling**: $m \cdot \Gamma$ multiplies all modes by m
- **Split**: $\Gamma = \Gamma_1 + \Gamma_2$ for parallel use

Definition 8.3 (Linear Typing Rules)**.**

$$\frac{}{x : \tau@1 \vdash x : \tau\ [\bot]}\ \textsc{L-Var} \qquad \frac{\Gamma \vdash e : \sigma\ [\varepsilon] \qquad m \geq \omega}{\Gamma, x : \tau@m \vdash e : \sigma\ [\varepsilon]}\ \textsc{L-Weaken}$$

$$\frac{\Gamma, x : \tau@\omega, y : \tau@\omega \vdash e : \sigma\ [\varepsilon]}{\Gamma, z : \tau@\omega \vdash e[z/x, z/y] : \sigma\ [\varepsilon]}\ \textsc{L-Contract} \qquad \frac{\Gamma, x : \tau_1@m \vdash e : \tau_2\ [\varepsilon]}{\Gamma \vdash \lambda x.\, e : (\tau_1@m) \xrightarrow{\varepsilon} \tau_2\ [\bot]}\ \textsc{L-Abs}$$

$$\frac{\Gamma_1 \vdash e_1 : (\tau_1@m) \xrightarrow{\varepsilon_f} \tau_2\ [\varepsilon_1] \qquad \Gamma_2 \vdash e_2 : \tau_1\ [\varepsilon_2]}{\Gamma_1 + \Gamma_2 \vdash e_1\ e_2 : \tau_2\ [\varepsilon_1 \sqcup \varepsilon_2 \sqcup \varepsilon_f]}\ \textsc{L-App}$$

$$\frac{\Gamma_1 \vdash e_1 : \tau_1\ [\varepsilon_1] \qquad \Gamma_2 \vdash e_2 : \tau_2\ [\varepsilon_2]}{\Gamma_1 + \Gamma_2 \vdash (e_1, e_2) : \tau_1 \otimes \tau_2\ [\varepsilon_1 \sqcup \varepsilon_2]}\ \textsc{L-Pair}$$

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \otimes \tau_2\ [\varepsilon_1] \qquad \Gamma_2, x : \tau_1@1, y : \tau_2@1 \vdash e_2 : \sigma\ [\varepsilon_2]}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\,(x, y) = e_1\ \mathsf{in}\ e_2 : \sigma\ [\varepsilon_1 \sqcup \varepsilon_2]}\ \textsc{L-LetPair}$$

8.2 Exponential Modality

Definition 8.4 (Exponential Type)**.** The exponential $!\tau$ internalizes unrestricted use:

$$!\tau \cong \tau@\omega$$

Definition 8.5 (Exponential Typing Rules)**.**

$$\frac{!\Gamma \vdash e : \tau\ [\varepsilon]}{!\Gamma \vdash !e : !\tau\ [\varepsilon]}\ \textsc{L-Promote} \qquad \frac{\Gamma \vdash e : !\tau\ [\varepsilon]}{\Gamma \vdash \mathsf{derelict}(e) : \tau\ [\varepsilon]}\ \textsc{L-Derelict}$$

where $!\Gamma$ means all variables in Γ have mode ω.

8.3 F* Mechanization

Linear Type System in F*

```
module BrrrSemantics.Linear

open BrrrSemantics.Modes
open BrrrSemantics.Effects

(* Expression AST *)
noeq type expr =
  | EVar     : string → expr
  | ELam     : string → moded_type → expr → expr
  | EApp     : expr → expr → expr
  | EPair    : expr → expr → expr
  | ELetPair: string → string → expr → expr → expr
  | EBang    : expr → expr               (* Promote to exponential *)
  | EDerelict: expr → expr               (* Use exponential once *)
  | EUnit    : expr
  | ELit     : base_val → expr

(* Typing result *)
type typing_result =
  | TyOk     : brrr_type → effect_row → moded_ctx → typing_result
  | TyError : string → typing_result

(* Context addition - pointwise mode addition *)
let rec ctx_add (c1 c2:moded_ctx) : option moded_ctx =
  match c1, c2 with
  | [], [] → Some []
  | (x1, t1) :: r1, (x2, t2) :: r2 →
      if x1 = x2 && t1.ty = t2.ty then
        match ctx_add r1 r2 with
        | Some rest → Some ((x1, {ty=t1.ty; mode=mode_add t1.mode t2.
            mode}) :: rest)
        | None → None
      else None
  | _, _ → None

(* Check if context is all ω (for promotion) *)
let rec ctx_all_ω (ctx:moded_ctx) : bool =
  match ctx with
  | [] → true
  | (_, {mode=MOmega}) :: rest → ctx_all_ω rest
  | _ → false

(* Check if variable is used (mode > 0) *)
let is_used (m:mode) : bool = m ≠ MZero

(* Linear type checking *)
let rec typecheck (ctx:moded_ctx) (e:expr) : typing_result =
  match e with
  | EVar x →
      (match List.assoc x ctx with
       | Some {ty; mode} →
           if is_used mode then
             (* Use variable: subtract one use *)
             let ctx' = List.map (fun (y, t) →
              if y = x then (y, {t with mode =
                match t.mode with
                | MOne → MZero
                | MOmega → MOmega
                | _ → MZero})
              else (y, t)) ctx in
            TyOk ty Pure ctx'
          else TyError ("Variable " ^ x ^ " not available")
      | None → TyError ("Unbound variable " ^ x))

  | ELam x param_ty body →
      let extended_ctx = (x, param_ty) :: ctx in
      (match typecheck extended_ctx body with
       | TyOk ret_ty eff ctx' →
           (* Check x was used according to its mode *)
           let func_ty = TFunc param_ty.ty ret_ty eff in
           TyOk func_ty Pure ctx
       | err → err)

  | EApp f arg →
      (match typecheck ctx f with
       | TyOk (TFunc arg_ty ret_ty eff_f) eff1 ctx1 →
           (match typecheck ctx1 arg with
            | TyOk arg_ty' eff2 ctx2 →
                if subtype arg_ty' arg_ty then
                  TyOk ret_ty (effect_join eff1 (effect_join eff2
                     eff_f)) ctx2
                else TyError "Argument type mismatch"
            | err → err)
       | TyOk _ _ _ → TyError "Not a function"
       | err → err)

  | EPair e1 e2 →
      (match typecheck ctx e1 with
       | TyOk t1 eff1 ctx1 →
           (match typecheck ctx1 e2 with
            | TyOk t2 eff2 ctx2 →
                TyOk (TPair t1 t2) (effect_join eff1 eff2) ctx2
            | err → err)
       | err → err)

  | EBang inner →
      if ctx_all_ω ctx then
        match typecheck ctx inner with
        | TyOk t eff ctx' →
            (* Exponential: type becomes copyable *)
            TyOk t eff ctx'  (* In real impl, wrap type *)
        | err → err
      else TyError "Cannot promote: context not all ω"

  | EUnit → TyOk TUnit Pure ctx

  | ELit v →
      let ty = match v with
        | VBool _ → TBool
        | VInt _  → TInt {width=I64; sign=Signed}
        | VFloat _ → TFloat F64
        | VString _ → TString
        | VUnit → TUnit
      in TyOk ty Pure ctx

  | _ → TyError "Not implemented"

(* Verify all linear resources consumed *)
let check_consumed (ctx:moded_ctx) : bool =
  List.for_all (fun (_, {mode}) →
    mode = MZero || mode = MOmega) ctx
```

9. Borrowing as Fractional Permissions

9.1 Fractional Permissions

Definition 9.1 (Permission Fraction)**.** A permission $p \in (0,1]$ represents partial ownership:

- $p = 1$: Full ownership (read, write, move, drop)
- $0 < p < 1$: Partial ownership (read only)

Definition 9.2 (Permission Operations)**.**

$$\begin{aligned} \mathsf{split} &: \tau@p \to (\tau@p/2) \otimes (\tau@p/2) \\ \mathsf{join} &: (\tau@p_1) \otimes (\tau@p_2) \to \tau@(p_1 + p_2) \quad \text{if } p_1 + p_2 \leq 1 \end{aligned}$$

Theorem 9.3 (Permission Invariant)**.** *For any resource, the sum of all permissions equals 1:*

$$\sum_i p_i = 1$$

9.2 Borrowing as Modal Operators

Definition 9.4 (Box Modality (Shared Borrow))**.** $\Box\tau$ represents a frozen/borrowed reference:

$$\&x : \Box\tau \quad \text{(immutable borrow)}$$

Properties:

- $\Box\tau$ can be duplicated (shared borrows coexist)
- $\Box\tau$ provides read-only access

Definition 9.5 (Diamond Modality (Exclusive Borrow))**.** $\Diamond\tau$ represents exclusive access:

$$\&\mathsf{mut}\, x : \Diamond\tau \quad \text{(mutable borrow)}$$

Properties:

- $\Diamond\tau$ is linear (exactly one holder)
- $\Diamond\tau$ provides read-write access

Definition 9.6 (Borrow Typing Rules)**.**

$$\frac{\Gamma \vdash e : \tau@1\ [\varepsilon]}{\Gamma \vdash \mathsf{freeze}(e) : \Box\tau@\omega\ [\varepsilon]}\ \textsc{T-Freeze} \qquad \frac{\Gamma \vdash e : \Box\tau\ [\varepsilon]}{\Gamma \vdash \mathsf{read}(e) : \tau\ [\varepsilon \sqcup \mathsf{Read}]}\ \textsc{T-Read}$$

$$\frac{\Gamma \vdash e : \tau@1\ [\varepsilon] \qquad \text{no other borrows of } e}{\Gamma \vdash \&\mathsf{mut}\, e : \Diamond\tau@1\ [\varepsilon]}\ \textsc{T-MutBorrow}$$

$$\frac{\Gamma \vdash e_1 : \Diamond\tau@1\ [\varepsilon_1] \qquad \Gamma \vdash e_2 : \tau\ [\varepsilon_2]}{\Gamma \vdash *e_1 := e_2 : \mathsf{Unit}\ [\varepsilon_1 \sqcup \varepsilon_2 \sqcup \mathsf{Write}]}\ \textsc{T-Write}$$

9.3 Mapping to Rust-Style Syntax

Definition 9.7 (Ownership Mode Mapping).

Brrr-Lang Mode	Linear Logic	Rust Equivalent
own τ	$\tau@1$	`T` (owned)
ref τ	$\Box\tau@\omega$	`&T`
ref mut τ	$\Diamond\tau@1$	`&mut T`
rc τ	$!(\mathsf{counted}\ \tau)@\omega$	`Rc<T>`
arc τ	$!(\mathsf{atomic\_counted}\ \tau)@\omega$	`Arc<T>`
gc τ	$!(\mathsf{gc\_managed}\ \tau)@\omega$	GC types

9.4 F* Mechanization

Borrowing in F*

```
module BrrrSemantics.Borrow

open BrrrSemantics.Modes

(* Permission fraction (represented as rational) *)
type permission = { num : nat; denom : pos }

let full_perm : permission = { num = 1; denom = 1 }
let half_perm : permission = { num = 1; denom = 2 }

let perm_leq (p1 p2:permission) : bool =
  p1.num * p2.denom ≤ p2.num * p1.denom

let perm_add (p1 p2:permission) : permission =
  { num = p1.num * p2.denom + p2.num * p1.denom;
    denom = p1.denom * p2.denom }

let perm_half (p:permission) : permission =
  { num = p.num; denom = p.denom * 2 }

(* Borrow kind *)
type borrow_mode =
  | BShared     : permission → borrow_mode    (* &T with fraction *)
  | BExclusive : borrow_mode                   (* &mut T *)
  | BOwned      : borrow_mode                   (* T owned *)

(* Reference type with borrow info *)
noeq type ref_type = {
  inner_ty  : brrr_type;
  borrow    : borrow_mode;
  region    : string              (* Lifetime/region variable *)
}

(* Can read? *)
let can_read (b:borrow_mode) : bool = true   (* All borrows allow read
    *)

(* Can write? *)
let can_write (b:borrow_mode) : bool =
  match b with
  | BExclusive → true
  | BOwned → true
  | BShared _ → false

(* Can share (duplicate)? *)
let can_share (b:borrow_mode) : bool =
  match b with
  | BShared _ → true
  | _ → false

(* Split shared borrow *)
let split_shared (p:permission) : option (permission & permission) =
  let half = perm_half p in
  Some (half, half)

(* Join shared borrows *)
let join_shared (p1 p2:permission) : option permission =
  let sum = perm_add p1 p2 in
  if perm_leq sum full_perm then Some sum
  else None  (* Would exceed full permission *)

(* Borrow checking state *)
noeq type borrow_state = {
  loans : list (string & borrow_mode);  (* Active borrows *)
}

(* Check if mutable borrow is allowed *)
let can_mut_borrow (var:string) (state:borrow_state) : bool =
  (* No existing borrows of this variable *)
  not (List.∃ (fun (v, _) → v = var) state.loans)

(* Check if shared borrow is allowed *)
let can_shared_borrow (var:string) (state:borrow_state) : bool =
  (* No exclusive borrows of this variable *)
  not (List.∃ (fun (v, b) →
    v = var && (match b with BExclusive → true | _ → false))
    state.loans)

(* Create shared borrow *)
let create_shared_borrow (var:string) (state:borrow_state)
    : option borrow_state =
  if can_shared_borrow var state then
    Some { loans = (var, BShared half_perm) :: state.loans }
  else None

(* Create mutable borrow *)
let create_mut_borrow (var:string) (state:borrow_state)
    : option borrow_state =
  if can_mut_borrow var state then
    Some { loans = (var, BExclusive) :: state.loans }
  else None

(* Release borrow *)
let release_borrow (var:string) (state:borrow_state) : borrow_state =
  { loans = List.filter (fun (v, _) → v ≠ var) state.loans }
```

10. Region Types and Lifetimes

10.1 Region Variables

Definition 10.1 (Region). A region ρ represents a scope during which memory is valid:

$$\tau@\rho \quad \text{(type } \tau \text{ allocated in region } \rho\text{)}$$

Definition 10.2 (Region Capability). $\mathsf{cap}[\rho]$ is the capability to access region ρ.

Definition 10.3 (Region Ordering (Outlives)). $\rho_1 \leq \rho_2$ means region ρ_1 outlives (includes) ρ_2:

$$\rho_1 \leq \rho_2 \implies \tau@\rho_1 <: \tau@\rho_2$$

10.2 Region Typing Rules

Definition 10.4 (Region Scoping).

$$\frac{\Gamma; \mathsf{cap}[\rho] \vdash e : \tau\ [\varepsilon]}{\Gamma; \mathsf{cap}[\rho] \vdash \mathsf{new\_at}[\rho](e) : \tau@\rho\ [\varepsilon \sqcup \mathsf{Alloc}]}\ \textsc{R-Alloc}$$

$$\frac{\Gamma; \mathsf{cap}[\rho] \vdash e : \tau\ [\varepsilon] \qquad \rho \notin \mathsf{frv}(\tau)}{\Gamma \vdash \mathsf{letregion}\ \rho\ \mathsf{in}\ e : \tau\ [\varepsilon]}\ \textsc{R-LetRegion} \qquad \frac{\Gamma \vdash e : \tau@\rho_1\ [\varepsilon] \qquad \rho_1 \leq \rho_2}{\Gamma \vdash e : \tau@\rho_2\ [\varepsilon]}\ \textsc{R-Sub}$$

Definition 10.5 (Static Region). static is the region that outlives all others:

$$\forall \rho.\ \mathsf{static} \leq \rho$$

10.3 Lifetime-Bounded References

Definition 10.6 (Reference with Lifetime).

$$\mathsf{ref}[\rho]\,\tau \quad \text{(reference to } \tau \text{ valid for lifetime } \rho\text{)}$$

Definition 10.7 (Reference Typing).

$$\frac{\Gamma \vdash e : \tau@\rho\ [\varepsilon]}{\Gamma \vdash \&e : \mathsf{ref}[\rho]\,\tau\ [\varepsilon]}\ \textsc{T-Ref} \qquad \frac{\Gamma; \mathsf{cap}[\rho] \vdash e : \mathsf{ref}[\rho]\,\tau\ [\varepsilon]}{\Gamma; \mathsf{cap}[\rho] \vdash *e : \tau\ [\varepsilon \sqcup \mathsf{Read}]}\ \textsc{T-Deref}$$

10.4 F* Mechanization

Regions and Lifetimes in F*

```
module BrrrSemantics.Regions

(* Region identifier *)
type region =
  | RStatic : region                           (* 'static - lives forever *)
  | RNamed  : string → region                 (* Named region variable *)
  | RFresh  : nat → region                    (* Fresh region from letregion
      *)

(* Region ordering (outlives) *)
let region_outlives (r1 r2:region) : bool =
  match r1, r2 with
  | RStatic, _ → true                         (* Static outlives everything
      *)
  | RNamed a, RNamed b → a = b                (* Same region *)
  | RFresh n1, RFresh n2 → n1 ≤ n2           (* Earlier = longer lived *)
  | _, _ → false

(* Type with region annotation *)
noeq type regioned_type = {
  base : brrr_type;
  region : region
}

(* Region capability *)
type region_cap = region

(* Region context *)
type region_ctx = list region_cap

(* Check if region is accessible *)
let has_cap (r:region) (ctx:region_ctx) : bool =
  List.mem r ctx

(* Reference type with lifetime *)
noeq type ref_with_lifetime = {
  pointee : brrr_type;
  lifetime : region;
  mutability : bool    (* true = &mut, false = & *)
}

(* Free region variables in type *)
let rec free_regions (t:brrr_type) : list region =
  match t with
  | TFunc a r _ → free_regions a @ free_regions r
  | TOption t' → free_regions t'
  | TArray t' → free_regions t'
  | TPair a b → free_regions a @ free_regions b
  | _ → []

(* Check region escapes scope *)
let region_escapes (r:region) (t:brrr_type) : bool =
  List.mem r (free_regions t)

(* Letregion scope *)
let letregion_ok (r:region) (body_ty:brrr_type) : bool =
  not (region_escapes r body_ty)

(* Region substitution *)
let rec subst_region (old_r new_r:region) (t:brrr_type) : brrr_type =
   (* Would need regioned_type throughout - simplified here *)
  t

(* Well-formed reference: lifetime must be in scope *)
let ref_wf (ref:ref_with_lifetime) (ctx:region_ctx) : bool =
  has_cap ref.lifetime ctx

(* Subtyping with regions *)
let ref_subtype (r1 r2:ref_with_lifetime) : bool =
  r1.pointee = r2.pointee &&
  r1.mutability = r2.mutability &&
  region_outlives r1.lifetime r2.lifetime
```

11. Separation Logic Compatibility

11.1 Separating Conjunction

Definition 11.1 (Tensor Product)**.** The tensor product $\tau_1 \otimes \tau_2$ represents values in disjoint memory:

$$\tau_1 \otimes \tau_2 \quad (\tau_1 \text{ and } \tau_2 \text{ occupy separate resources})$$

Theorem 11.2 (Separation Property)**.** *If $\Gamma_1 \vdash e_1 : \tau_1$ and $\Gamma_2 \vdash e_2 : \tau_2$ with Γ_1 and Γ_2 disjoint, then:*

$$\Gamma_1 + \Gamma_2 \vdash (e_1, e_2) : \tau_1 \otimes \tau_2$$

The values do not alias.

11.2 Magic Wand

Definition 11.3 (Linear Implication)**.** The magic wand (linear implication) $\tau_1 \multimap \tau_2$ means:

Given τ_1, produce τ_2 (consuming τ_1)

Definition 11.4 (Linear Function Typing)**.**

$$\frac{\Gamma, x : \tau_1@1 \vdash e : \tau_2\ [\varepsilon]}{\Gamma \vdash \lambda x.\, e : \tau_1 \multimap \tau_2\ [\varepsilon]}\ \textsc{T-Lolli-I} \qquad \frac{\Gamma_1 \vdash e_1 : \tau_1 \multimap \tau_2\ [\varepsilon_1] \qquad \Gamma_2 \vdash e_2 : \tau_1\ [\varepsilon_2]}{\Gamma_1 + \Gamma_2 \vdash e_1\ e_2 : \tau_2\ [\varepsilon_1 \sqcup \varepsilon_2]}\ \textsc{T-Lolli-E}$$

11.3 Frame Rule

Theorem 11.5 (Frame Rule)**.** *If $\Gamma \vdash e : \tau\ [\varepsilon]$, then for any Δ disjoint from e's resources:*

$$\Gamma, \Delta \vdash e : \tau\ [\varepsilon]$$

with Δ unchanged.

This enables local reasoning about heap-manipulating programs.

11.4 Ownership Predicates

Definition 11.6 (Points-To Predicate)**.** In separation logic style:

$$x \mapsto v \quad (x \text{ points to value } v\text{, full ownership})$$
$$x \mapsto^p v \quad (x \text{ points to } v \text{ with permission } p)$$

Definition 11.7 (Ownership Mapping)**.**

Brrr-Lang Type	Separation Logic
own $x : \tau$	$x \mapsto v$
ref $x : \tau$	$x \mapsto^{1/n} v$ (shared, read-only)
ref mut $x : \tau$	$x \mapsto v$ (exclusive)

11.5 F* Mechanization

Separation Logic in F*

```
module BrrrSemantics.Separation

open BrrrSemantics.Modes
open BrrrSemantics.Borrow

(* Heap as separation algebra *)
```

```
type heap = loc → option (brrr_type & value)

(* Disjoint heaps *)
let disjoint (h1 h2:heap) : bool =
  ∀ (l:loc). not (Some? (h1 l) && Some? (h2 l))

(* Heap composition (requires disjoint) *)
let heap_compose (h1 h2:heap) : heap =
  fun l → match h1 l with
    | Some v → Some v
    | None → h2 l

(* Separation logic assertion *)
noeq type sl_assert =
  | SLEmp     : sl_assert                             (* Empty heap *)
  | SLPointsTo : loc → value → sl_assert           (* l |→ v *)
  | SLPointsToFrac : loc → permission → value → sl_assert  (* l |→^p v
     *)
  | SLStar    : sl_assert → sl_assert → sl_assert (* P * Q *)
  | SLWand    : sl_assert → sl_assert → sl_assert (* P -* Q *)
  | SLPure    : bool → sl_assert                    (* Pure assertion *)

(* Satisfaction relation *)
let rec satisfies (h:heap) (p:sl_assert) : bool =
  match p with
  | SLEmp →
      ∀ (l:loc). None? (h l)
  | SLPointsTo l v →
      h l = Some (TAny, v) (* Simplified *)
  | SLStar p1 p2 →
      ∃ (h1 h2:heap).
        disjoint h1 h2 &&
        heap_compose h1 h2 = h &&
        satisfies h1 p1 &&
        satisfies h2 p2
  | SLWand p1 p2 →
      ∀ (h':heap).
        disjoint h h' && satisfies h' p1 ⟹
        satisfies (heap_compose h h') p2
  | SLPure b → b
  | _ → false

(* Frame rule: {P} c {Q} implies {P * R} c {Q * R} *)
type hoare_triple = {
  pre  : sl_assert;
  cmd  : expr;
  post : sl_assert
}

(* Frame rule is derivable *)
val frame_rule: ht:hoare_triple → frame:sl_assert →
  Lemma (requires satisfies_triple ht)
        (ensures satisfies_triple {
          pre = SLStar ht.pre frame;
          cmd = ht.cmd;
          post = SLStar ht.post frame
        })
```

```
(* Separation logic for ownership transfer *)
let move_ownership (src dst:string) (h:heap) (v:value) : heap =
  fun l →
    if l = lookup_loc src then None
    else if l = lookup_loc dst then Some (TAny, v)
    else h l

(* Borrow creates fractional permission *)
let create_borrow (var:string) (h:heap) : sl_assert =
  match lookup_loc_val var h with
  | Some (l, v) → SLPointsToFrac l half_perm v
  | None → SLPure false
```

Part IV
Advanced Effects

12. Delimited Control

This chapter grounds all control flow in delimited continuations, providing a unified semantic foundation for exceptions, early return, loops, and coroutines.

12.1 Continuation Fundamentals

Definition 12.1 (Continuation)**.** A continuation $\kappa : \tau \to \sigma$ represents "the rest of the computation" expecting a value of type τ and producing σ.

Definition 12.2 (CPS Types)**.** The CPS transformation maps types:

$$[\![\tau]\!]^{\mathsf{cps}} = \forall\alpha.\, (\tau \to \alpha) \to \alpha$$

Definition 12.3 (Evaluation Contexts)**.** An evaluation context E is a term with a hole $[\cdot]$:

$$\begin{aligned} E ::= {} & [\cdot] \\ & \mid E\ e \mid v\ E \\ & \mid \mathsf{let}\, x = E\ \mathsf{in}\ e \\ & \mid (E, e) \mid (v, E) \\ & \mid \mathsf{if}\ E\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \end{aligned}$$

12.2 Prompts and Delimiters

Definition 12.4 (Prompt)**.** A prompt p is a delimiter marking the boundary of continuation capture:

$$\mathsf{reset}\langle p\rangle\ e$$

The prompt p is a label identifying which delimiter to capture up to.

Definition 12.5 (Shift (Continuation Capture))**.** $\mathsf{shift}\langle p\rangle\ (\lambda k.\, e)$ captures the continuation up to the nearest enclosing $\mathsf{reset}\langle p\rangle$:

$$\mathsf{reset}\langle p\rangle\ E[\mathsf{shift}\langle p\rangle\ (\lambda k.\, e)] \longrightarrow \mathsf{reset}\langle p\rangle\ e[k := \lambda x.\, \mathsf{reset}\langle p\rangle\ E[x]]$$

Definition 12.6 (Delimited Control Typing)**.**

$$\frac{\Gamma \vdash e : \tau\ [\langle \mathsf{Prompt}\langle p, \sigma\rangle \mid \varepsilon\rangle]}{\Gamma \vdash \mathsf{reset}\langle p\rangle\ e : \sigma\ [\varepsilon]}\ \textsc{T-Reset} \qquad \frac{\Gamma, k : \tau \to \sigma\ [\varepsilon] \vdash e : \sigma\ [\varepsilon']}{\Gamma \vdash \mathsf{shift}\langle p\rangle\ (\lambda k.\, e) : \tau\ [\langle \mathsf{Prompt}\langle p, \sigma\rangle \mid \varepsilon'\rangle]}\ \textsc{T-Shift}$$

12.3 Control Operators

Definition 12.7 (Abort)**.** Abort discards the continuation entirely:

$$\mathsf{abort}\langle p\rangle\ v \triangleq \mathsf{shift}\langle p\rangle\ (\lambda\_.\, v)$$

Definition 12.8 (Call/CC (Undelimited))**.** Classical call/cc can be encoded:

$$\mathsf{call/cc}\ f \triangleq \mathsf{shift}\langle \mathsf{top}\rangle\ (\lambda k.\, k\ (f\ k))$$

Warning: Undelimited continuations are more powerful but harder to reason about.

12.4 F* Mechanization

Delimited Control in F*

```
module BrrrSemantics.DelimitedControl

(* Prompt label *)
```

```
type prompt = string

(* Answer type for a prompt *)
type answer_type = brrr_type

(* Continuation type *)
noeq type cont (a:Type) (r:Type) =
  | Cont : (a → r) → cont a r

(* Apply continuation *)
let apply_cont (#a #r:Type) (k:cont a r) (x:a) : r =
  match k with Cont f → f x

(* Delimited control monad *)
noeq type dc (a:Type) (r:Type) =
  | Pure   : a → dc a r
  | Shift  : prompt → (cont a r → dc r r) → dc a r

(* Reset: run computation with delimiter *)
let rec reset (#a:Type) (p:prompt) (m:dc a a) : a =
  match m with
  | Pure x → x
  | Shift p' body →
      if p = p' then
        (* Capture continuation *)
        let k : cont a a = Cont (fun x → reset p (Pure x)) in
        reset p (body k)
      else
        (* Propagate to outer reset *)
        admit() (* Would need effect row *)

(* Shift: capture continuation *)
let shift (#a #r:Type) (p:prompt) (f:cont a r → dc r r) : dc a r =
  Shift p f

(* Abort: discard continuation *)
let abort (#a #r:Type) (p:prompt) (v:r) : dc a r =
  Shift p (fun _ → Pure v)

(* Example: early exit *)
let example_early_exit () : int =
  reset "exit" (
    let x = 1 in
    let y = shift "exit" (fun k →
      (* k is: fun v → let y = v in x + y *)
      (* We can use it: *) apply_cont k 10
      (* Or abort: Pure 42 *)
    ) in
    Pure (x + y)
  )
  (* Result: 11 if we use k, or 42 if we abort *)

(* CPS transformation *)
type cps (a:Type) = ∀ r. (a → r) → r

let cps_return (#a:Type) (x:a) : cps a =
  fun k → k x
```

```
let cps_bind (#a #b:Type) (m:cps a) (f:a → cps b) : cps b =
  fun k → m (fun x → f x k)

(* Reset in CPS *)
let cps_reset (#a:Type) (m:cps a) : a =
  m (fun x → x)

(* Shift in CPS *)
let cps_shift (#a #r:Type) (f:(a → r) → r) : cps a =
  fun k → f k
```

13. Control Flow as Delimited Continuations

This chapter derives all standard control flow from delimited continuations.

13.1 Return as Abort

Definition 13.1 (Return Derivation)**.** Every function implicitly has a reset at its boundary:

$$\mathsf{fn}\ f(x:\tau) \to \sigma \triangleq \mathsf{body} \equiv \lambda x.\, \mathsf{reset}\langle \mathsf{ret}_f\rangle\ \mathsf{body}$$

Early return aborts to this delimiter:

$$\mathsf{return}\ e \triangleq \mathsf{abort}\langle \mathsf{ret}_f\rangle\ e$$

Definition 13.2 (Return Typing)**.**

$$\frac{\Gamma \vdash e : \sigma\ [\varepsilon] \qquad \text{enclosing function returns } \sigma}{\Gamma \vdash \mathsf{return}\ e : \mathsf{Never}\ [\langle \mathsf{Return}\langle\sigma\rangle \mid \varepsilon\rangle]}\ \textsc{T-Return}$$

13.2 Exceptions as Effects

Definition 13.3 (Exception Derivation)**.** Exceptions are abort with a handler:

$$\mathsf{throw}\ e \triangleq \mathsf{abort}\langle \mathsf{exn}\rangle\ (\mathsf{Err}\ e)$$

$$\begin{aligned}\mathsf{try}\ e_1\ \mathsf{catch}\ \{E(x) \Rightarrow e_2\} \triangleq \mathsf{reset}\langle \mathsf{exn}\rangle\ e_1 \mid_{\mathsf{match}} \{&\\ &\mathsf{Ok}(v) \Rightarrow v\\ &\mathsf{Err}(x:E) \Rightarrow e_2\\ \}&\end{aligned}$$

Definition 13.4 (Exception Typing)**.**

$$\frac{\Gamma \vdash e : E\ [\varepsilon]}{\Gamma \vdash \mathsf{throw}\ e : \mathsf{Never}\ [\langle \mathsf{Throw}[E] \mid \varepsilon\rangle]}\ \textsc{T-Throw}$$

$$\frac{\Gamma \vdash e_1 : \tau\ [\langle \mathsf{Throw}[E] \mid \varepsilon\rangle] \qquad \Gamma, x : E \vdash e_2 : \tau\ [\varepsilon']}{\Gamma \vdash \mathsf{try}\ e_1\ \mathsf{catch}\ \{E(x) \Rightarrow e_2\} : \tau\ [\varepsilon \sqcup \varepsilon']}\ \textsc{T-TryCatch}$$

13.3 Loops as Fixpoints

Definition 13.5 (While Loop Derivation)**.** Loops are recursive functions (fixpoints):

$$\mathsf{while}\ c\ \{\mathsf{body}\} \triangleq \mathsf{fix}\,(\lambda \mathsf{loop}.\, \lambda().\, \mathsf{if}\ c\ \mathsf{then}\ \{\mathsf{body}; \mathsf{loop}()\}\ \mathsf{else}\ ())(\ )$$

Definition 13.6 (For Loop Derivation)**.**

$$\mathsf{for}\ x\ \mathsf{in}\ \mathsf{iter}\ \{\mathsf{body}\} \triangleq \mathsf{iter.fold}((), \lambda(\_, x).\, \mathsf{body})$$

Definition 13.7 (Infinite Loop)**.**

$$\mathsf{loop}\ \{\mathsf{body}\} \triangleq \mathsf{fix}\,(\lambda \mathsf{loop}.\ \lambda().\ \{\mathsf{body};\mathsf{loop}()\})(\ )$$

Definition 13.8 (Loop Typing with Divergence)**.**

$$\frac{\Gamma \vdash c : \mathsf{Bool}\ [\varepsilon_1] \qquad \Gamma \vdash \mathsf{body} : \mathsf{Unit}\ [\varepsilon_2]}{\Gamma \vdash \mathsf{while}\, c\ \{\mathsf{body}\} : \mathsf{Unit}\ [\varepsilon_1 \sqcup \varepsilon_2 \sqcup \mathsf{Div}]}\ \textsc{T-While}$$

13.4 Labeled Break/Continue

Definition 13.9 (Labeled Loop Structure)**.** A labeled loop introduces two prompts—one for break, one for continue:

$$\begin{aligned}
&\mathsf{label}\,\ell : \mathsf{loop}\ \{\mathsf{body}\}\\
&\quad\triangleq \mathsf{reset}\langle \ell_{\mathsf{break}}\rangle\ \{\\
&\qquad\quad \mathsf{fix}\,(\lambda \mathsf{iter}.\ \lambda().\\
&\qquad\qquad \mathsf{reset}\langle \ell_{\mathsf{continue}}\rangle\ \mathsf{body};\\
&\qquad\qquad \mathsf{iter}()\\
&\qquad\quad)(\ )\\
&\quad\}
\end{aligned}$$

Definition 13.10 (Break Derivation)**.**

$$\mathsf{break}\,\ell \triangleq \mathsf{abort}\langle \ell_{\mathsf{break}}\rangle\ ()$$
$$\mathsf{break}\,\ell(v) \triangleq \mathsf{abort}\langle \ell_{\mathsf{break}}\rangle\ v$$

Definition 13.11 (Continue Derivation)**.**

$$\mathsf{continue}\,\ell \triangleq \mathsf{abort}\langle \ell_{\mathsf{continue}}\rangle\ ()$$

Definition 13.12 (Break/Continue Typing)**.**

$$\frac{\Gamma \vdash e : \tau\ [\varepsilon] \qquad \text{enclosing loop } \ell \text{ expects } \tau}{\Gamma \vdash \mathsf{break}\,\ell(e) : \mathsf{Never}\ [\langle \mathsf{Break}\langle \ell, \tau\rangle \mid \varepsilon\rangle]}\ \textsc{T-Break}$$

$$\frac{\text{enclosing loop } \ell}{\Gamma \vdash \mathsf{continue}\,\ell : \mathsf{Never}\ [\mathsf{Continue}\langle \ell\rangle]}\ \textsc{T-Continue}$$

13.5 Defer as Finalizer

Definition 13.13 (Defer Derivation)**.** Defer installs a finalizer that runs on scope exit:

$$\mathsf{defer}\, e; \mathsf{rest} \triangleq \mathsf{bracket}\ (\lambda k.\ \{e; k()\})\ \mathsf{rest}$$

where bracket ensures the finalizer runs regardless of how scope exits.

Definition 13.14 (Bracket Combinator)**.**

$$\mathsf{bracket} : ((\tau \to \tau) \to \tau \to \tau) \to \tau \to \tau$$

$$\mathsf{bracket}\,\mathsf{finalizer}\,\mathsf{body} = \mathsf{let}\, r = \mathsf{body}\ \mathsf{in}\ \mathsf{finalizer}(\lambda().\ r)$$

Definition 13.15 (Defer Typing)**.**

$$\frac{\Gamma \vdash e_d : \mathsf{Unit}\ [\varepsilon_d] \qquad \Gamma \vdash e : \tau\ [\varepsilon]}{\Gamma \vdash \mathsf{defer}\, e_d; e : \tau\ [\varepsilon_d \sqcup \varepsilon]}\ \textsc{T-Defer}$$

13.6 F* Mechanization

Control Flow Derivations in F*

```
module BrrrSemantics.ControlFlow

open BrrrSemantics.DelimitedControl

(* Control effect *)
type control_effect =
  | CReturn   : brrr_type → control_effect
  | CThrow    : brrr_type → control_effect
  | CBreak    : string → brrr_type → control_effect
  | CContinue : string → control_effect

(* Return as abort *)
let return_impl (#a:Type) (ret_prompt:prompt) (v:a) : dc a a =
  abort ret_prompt v

(* Function wrapper with return handling *)
let fn_wrapper (#a #r:Type) (body:dc r r) : r =
  reset "return" body

(* Exception handling *)
type result (a:Type) (e:Type) =
  | Ok  : a → result a e
  | Err : e → result a e

let throw (#a #e:Type) (exn:e) : dc a (result a e) =
  abort "exn" (Err exn)

let try_catch (#a #e:Type) (body:dc a (result a e)) (handler:e → dc a
   (result a e))
    : dc a (result a e) =
  let r = reset "exn" (
    match body with
    | Pure x → Pure (Ok x)
    | _ → body
  ) in
  match r with
  | Ok v → Pure (Ok v)
  | Err e → handler e

(* Fixpoint combinator *)
let rec fix (#a:Type) (f:(a → a) → a → a) (x:a) : a =
  f (fix f) x

(* While loop as fixpoint *)
let while_loop (cond:unit → bool) (body:unit → unit) : unit =
  fix (fun loop () →
    if cond () then (body (); loop ())
    else ()
  ) ()

(* Labeled loop with break/continue *)
type loop_control =
  | LBreak    : option 'a → loop_control
  | LContinue : loop_control
  | LNormal   : loop_control
```

```
let labeled_loop (#a:Type) (label:string) (body:unit → dc loop_control
    loop_control)
    : option a =
  reset (label ^ "_break") (
    fix (fun iter () →
      let ctrl = reset (label ^ "_continue") (body ()) in
      match ctrl with
      | Pure LNormal → iter ()
      | Pure LContinue → iter ()
      | Pure (LBreak v) → Pure (LBreak v)
      | _ → iter ()
    ) ()
  ) |> (function
    | Pure (LBreak (Some v)) → Some v
    | _ → None
  )

(* Break implementation *)
let break (#a:Type) (label:string) (v:option a) : dc loop_control
   loop_control =
  abort (label ^ "_break") (LBreak v)

(* Continue implementation *)
let continue (label:string) : dc loop_control loop_control =
  abort (label ^ "_continue") LContinue

(* Defer/bracket pattern *)
let bracket (#a:Type) (finalizer:unit → unit) (body:unit → a) : a =
  let result = body () in
  finalizer ();
  result

let with_defer (#a:Type) (cleanup:unit → unit) (body:unit → a) : a =
  bracket cleanup body

(* Example: file handling with defer *)
let example_file_handling () : string =
  with_defer
    (fun () → (* close file *) ())
    (fun () →
      (* open file *)
      (* read contents *)
      "file contents"
    )
```

14. Generators and Coroutines

14.1 Generator as Effect

Definition 14.1 (Yield Effect).

$$\mathsf{effect\ Yield}[Y, R]\ \{\mathsf{yield} : Y \rightsquigarrow R\}$$

A generator yields values of type Y and receives resumption values of type R.

Definition 14.2 (Generator Type).

$$\mathsf{Generator}[Y, R, T] \cong \mathsf{Unit} \xrightarrow{[\mathsf{Yield}[Y,R]]} T$$

Definition 14.3 (Generator Typing).

$$\frac{\Gamma \vdash e : Y\ [\varepsilon]}{\Gamma \vdash \mathsf{yield}\, e : R\ [\langle \mathsf{Yield}[Y, R] \mid \varepsilon\rangle]}\ \textsc{T-Yield}$$

$$\frac{\Gamma \vdash e : T\ [\langle \mathsf{Yield}[Y, R] \mid \varepsilon\rangle]}{\Gamma \vdash \mathsf{generator}\ e : \mathsf{Generator}[Y, R, T]\ [\varepsilon]}\ \textsc{T-Generator}$$

14.2 Generator Semantics via Handlers

Definition 14.4 (Generator Runner).

$$\begin{aligned}
&\mathsf{run\_generator} : \mathsf{Generator}[Y, R, T] \to \mathsf{Iterator}[Y]\\
&\mathsf{run\_generator}\, g = \mathsf{handle}\, g()\ \mathsf{with}\, \{\\
&\quad \mathsf{return}\, x \Rightarrow \mathsf{Done}(x)\\
&\quad \mathsf{yield}(y, k) \Rightarrow \mathsf{Yielded}(y, \lambda r.\, \mathsf{run\_generator}\ (\lambda().\, k(r)))\\
&\}
\end{aligned}$$

14.3 Async/Await

Definition 14.5 (Async Effect).

$$\mathsf{effect\ Async}\ \{\mathsf{await} : \mathsf{Future}[\tau] \rightsquigarrow \tau,\ \mathsf{spawn} : (\mathsf{Unit} \xrightarrow{[\mathsf{Async}]} \tau) \rightsquigarrow \mathsf{Future}[\tau]\}$$

Definition 14.6 (Future Types).

$$\begin{aligned}
\mathsf{Future}[\tau, \mathsf{Hot}] &\quad \text{(computation already started)}\\
\mathsf{Future}[\tau, \mathsf{Cold}] &\quad \text{(computation deferred until awaited)}
\end{aligned}$$

Definition 14.7 (Async Typing).

$$\frac{\Gamma \vdash e : \tau\ [\langle \mathsf{Async} \mid \varepsilon\rangle]}{\Gamma \vdash \mathsf{async}\, e : \mathsf{Future}[\tau, \mathsf{Cold}]\ [\varepsilon]}\ \textsc{T-Async} \qquad \frac{\Gamma \vdash e : \mathsf{Future}[\tau, \_]\ [\varepsilon]}{\Gamma \vdash \mathsf{await}\, e : \tau\ [\langle \mathsf{Async} \mid \varepsilon\rangle]}\ \textsc{T-Await}$$

$$\frac{\Gamma \vdash e : \tau\ [\langle \mathsf{Async} \mid \varepsilon\rangle]}{\Gamma \vdash \mathsf{spawn}\, e : \mathsf{Future}[\tau, \mathsf{Hot}]\ [\varepsilon]}\ \textsc{T-Spawn}$$

14.4 F* Mechanization

Generators and Async in F*

```
module BrrrSemantics.Generators

(* Generator state *)
noeq type gen_state (y:Type) (r:Type) (t:Type) =
  | GenDone     : t → gen_state y r t
  | GenYielded : y → (r → gen_state y r t) → gen_state y r t

(* Generator as resumable computation *)
noeq type generator (y:Type) (r:Type) (t:Type) =
  | Gen : (unit → gen_state y r t) → generator y r t

(* Create generator *)
let make_generator (#y #r #t:Type) (body:unit → gen_state y r t)
    : generator y r t =
  Gen body
```

```
(* Step generator *)
let step_generator (#y #r #t:Type) (g:generator y r t)
    : gen_state y r t =
  match g with Gen f → f ()

(* Yield operation (used inside generator) *)
let yield (#y #r:Type) (value:y) (cont:r → gen_state y r 'a)
    : gen_state y r 'a =
  GenYielded value cont

(* Iterator from generator *)
noeq type iterator (a:Type) =
  | IterDone : iterator a
  | IterNext : a → (unit → iterator a) → iterator a

let rec gen_to_iter (#y #t:Type) (g:generator y unit t) : iterator y =
  match step_generator g with
  | GenDone _ → IterDone
  | GenYielded v cont →
      IterNext v (fun () → gen_to_iter (Gen (fun () → cont ())))

(* Async/Await *)

(* Future temperature *)
type temperature = | Hot | Cold

(* Future state *)
noeq type future_state (a:Type) =
  | Pending  : (unit → future_state a) → future_state a
  | Resolved : a → future_state a
  | Failed   : exn → future_state a

(* Future with temperature *)
noeq type future (a:Type) (temp:temperature) = {
  state : ref (future_state a);
  temp  : temperature
}

(* Async block - creates cold future *)
let async_cold (#a:Type) (body:unit → a) : future a Cold = {
  state = ref (Pending (fun () → Resolved (body ())));
  temp = Cold
}

(* Spawn - creates hot future (starts immediately) *)
let spawn (#a:Type) (body:unit → a) : future a Hot =
  (* In real impl, would schedule on executor *)
  let result = body () in
  { state = ref (Resolved result); temp = Hot }

(* Await - blocks until resolved *)
let await (#a:Type) (#t:temperature) (fut:future a t) : a =
  let rec poll () : a =
    match !fut.state with
    | Resolved v → v
    | Failed e → raise e
    | Pending next →
```

```
        fut.state := next ();
        poll ()
  in poll ()

(* Async effect signature *)
noeq type async_op =
  | OpAwait : #a:Type → future a Hot → async_op
  | OpSpawn : #a:Type → (unit → a) → async_op

(* Async computation as free monad *)
noeq type async_comp (a:Type) =
  | AsyncPure  : a → async_comp a
  | AsyncBind  : async_op → (unit → async_comp a) → async_comp a

(* Run async computation *)
let rec run_async (#a:Type) (comp:async_comp a) : a =
  match comp with
  | AsyncPure x → x
  | AsyncBind (OpAwait fut) cont →
      let v = await fut in
      run_async (cont ())
  | AsyncBind (OpSpawn body) cont →
      let _ = spawn body in
      run_async (cont ())
```

15. Structured Concurrency

15.1 Task Groups

Definition 15.1 (Task Group)**.** A task group ensures all spawned tasks complete before the group exits:

$$\mathsf{TaskGroup} : \mathsf{Set}[\mathsf{Future}[\tau]]$$

Definition 15.2 (Task Group Operations)**.**

$$\mathsf{spawn\_in} : \mathsf{TaskGroup} \to (\mathsf{Unit} \to \tau\ [\mathsf{Async}]) \to \mathsf{Future}[\tau]$$
$$\mathsf{wait\_all} : \mathsf{TaskGroup} \to [\tau]\ [\mathsf{Async}]$$

Definition 15.3 (Structured Concurrency Typing)**.**

$$\frac{\Gamma, g : \mathsf{TaskGroup} \vdash e : \tau\ [\varepsilon]}{\Gamma \vdash \mathsf{task\_group}\,(\lambda g.\, e) : \tau\ [\varepsilon]}\ \textsc{T-TaskGroup}$$

The task group g cannot escape; all tasks must complete within the scope.

15.2 Cancellation

Definition 15.4 (Cancellation Token)**.**

$$\mathsf{CancelToken} : \{\mathsf{cancelled} : \mathsf{ref\ Bool}\}$$

Definition 15.5 (Cancellation Typing)**.**

$$\frac{\Gamma \vdash \mathsf{token} : \mathsf{CancelToken}\ [\varepsilon]}{\Gamma \vdash \mathsf{check\_cancelled\,token} : \mathsf{Unit}\ [\langle \mathsf{Cancelled} \mid \varepsilon\rangle]}\ \textsc{T-CheckCancel}$$

15.3 Channels

Definition 15.6 (Channel Types)**.**

$$\begin{aligned} &\mathsf{Sender}[\tau] \quad \text{(send end)} \\ &\mathsf{Receiver}[\tau] \quad \text{(receive end)} \\ &\mathsf{Channel}[\tau] = \mathsf{Sender}[\tau] \times \mathsf{Receiver}[\tau] \end{aligned}$$

Definition 15.7 (Channel Operations)**.**

$$\frac{\Gamma \vdash s : \mathsf{Sender}[\tau]\ [\varepsilon_1] \qquad \Gamma \vdash v : \tau\ [\varepsilon_2]}{\Gamma \vdash s.\mathsf{send}(v) : \mathsf{Unit}\ [\langle \mathsf{Async} \mid \varepsilon_1 \sqcup \varepsilon_2 \rangle]}\ \textsc{T-Send} \qquad \frac{\Gamma \vdash r : \mathsf{Receiver}[\tau]\ [\varepsilon]}{\Gamma \vdash r.\mathsf{recv}() : \mathsf{Option}[\tau]\ [\langle \mathsf{Async} \mid \varepsilon \rangle]}\ \textsc{T-Recv}$$

15.4 Select

Definition 15.8 (Select Expression)**.** Select waits on multiple channels:

$$\begin{aligned} &\mathsf{select}\ \{ \\ &\quad r_1.\mathsf{recv}() \Rightarrow e_1 \\ &\quad r_2.\mathsf{recv}() \Rightarrow e_2 \\ &\quad \mathsf{default} \Rightarrow e_d \\ &\} \end{aligned}$$

15.5 F* Mechanization

Structured Concurrency in F*

```
module BrrrSemantics.Concurrency

(* Task handle *)
noeq type task (a:Type) = {
  future : future a Hot;
  cancel : cancel_token
}

(* Cancellation token *)
and cancel_token = {
  cancelled : ref bool;
  children  : ref (list cancel_token)
}

(* Create cancellation token *)
let new_cancel_token () : cancel_token = {
  cancelled = ref false;
  children = ref []
}

(* Cancel token and children *)
let rec cancel (tok:cancel_token) : unit =
  tok.cancelled := true;
  List.iter cancel (!tok.children)

(* Check if cancelled *)
let is_cancelled (tok:cancel_token) : bool =
  !tok.cancelled

(* Task group *)
noeq type task_group = {
  tasks  : ref (list (task unit));
  parent_cancel : cancel_token
}

(* Create task group *)
let new_task_group (parent:cancel_token) : task_group = {
  tasks = ref [];
  parent_cancel = parent
}

(* Spawn in task group *)
let spawn_in (#a:Type) (g:task_group) (body:unit → a) : task a =
  let child_cancel = new_cancel_token () in
  g.parent_cancel.children := child_cancel :: !(g.parent_cancel.
      children);
  let fut = spawn (fun () →
    if is_cancelled child_cancel then raise Cancelled
    else body ()
  ) in
  let t = { future = fut; cancel = child_cancel } in
  g.tasks := ({ future = spawn (fun () → ignore (await fut));
                cancel = child_cancel }) :: !(g.tasks);
  t

(* Wait for all tasks in group *)
let wait_all (g:task_group) : unit =
  List.iter (fun t → ignore (await t.future)) !(g.tasks)

(* Run with task group (structured concurrency) *)
let with_task_group (#a:Type) (body:task_group → a) : a =
  let cancel = new_cancel_token () in
  let g = new_task_group cancel in
  let result = body g in
  wait_all g;  (* Ensure all tasks complete *)
  result

(* Channel *)
noeq type channel (a:Type) = {
  buffer : ref (list a);
  closed : ref bool
}

type sender (a:Type) = channel a
type receiver (a:Type) = channel a

(* Create channel *)
let new_channel (#a:Type) () : sender a & receiver a =
  let ch = { buffer = ref []; closed = ref false } in
  (ch, ch)

(* Send on channel *)
let send (#a:Type) (s:sender a) (v:a) : unit =
  if !(s.closed) then raise ChannelClosed
  else s.buffer := !(s.buffer) @ [v]

(* Receive from channel *)
let recv (#a:Type) (r:receiver a) : option a =
  match !(r.buffer) with
  | [] → if !(r.closed) then None else None (* would block *)
  | x :: xs → r.buffer := xs; Some x

(* Close channel *)
let close (#a:Type) (s:sender a) : unit =
  s.closed := true

(* Select result *)
type select_result (a:Type) (b:Type) =
  | SelectFirst   : a → select_result a b
  | SelectSecond : b → select_result a b
  | SelectNeither : select_result a b

(* Select on two receivers *)
let select2 (#a #b:Type) (r1:receiver a) (r2:receiver b)
    : select_result a b =
  match recv r1 with
  | Some v → SelectFirst v
  | None → match recv r2 with
    | Some v → SelectSecond v
    | None → SelectNeither
```

Part V

Expressions & Patterns

16. Expression Semantics

16.1 Denotational Semantics

Definition 16.1 (Semantic Function)**.** The semantic function maps expressions to computations:

$$[\![\cdot]\!] : \mathsf{Expr} \to \mathsf{Env} \to \mathsf{Comp}[\mathsf{Val}]$$

Definition 16.2 (Core Expression Semantics)**.**

$$\begin{aligned}
[\![x]\!]\rho &= \mathsf{return}\,(\rho(x)) \\
[\![\lambda x.\, e]\!]\rho &= \mathsf{return}\,(\mathsf{Closure}(\rho, x, e)) \\
[\![e_1\, e_2]\!]\rho &= [\![e_1]\!]\rho \gg= \lambda f.\, [\![e_2]\!]\rho \gg= \lambda v.\, \mathsf{apply}(f, v) \\
[\![\mathsf{let}\, x = e_1 \;\mathsf{in}\; e_2]\!]\rho &= [\![e_1]\!]\rho \gg= \lambda v.\, [\![e_2]\!](\rho[x \mapsto v]) \\
[\![\mathsf{if}\, c\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2]\!]\rho &= [\![c]\!]\rho \gg= \lambda b.\, \mathsf{if}\, b\, \mathsf{then}\, [\![e_1]\!]\rho\, \mathsf{else}\, [\![e_2]\!]\rho \\
[\![(e_1, e_2)]\!]\rho &= [\![e_1]\!]\rho \gg= \lambda v_1.\, [\![e_2]\!]\rho \gg= \lambda v_2.\, \mathsf{return}\,(v_1, v_2)
\end{aligned}$$

16.2 Compositional Effect Semantics

Definition 16.3 (Effect Composition)**.** Effects compose via the graded monad structure:

$$[\![e_1; e_2]\!]_{\varepsilon_1 \sqcup \varepsilon_2} = [\![e_1]\!]_{\varepsilon_1} \gg= \lambda\_.\, [\![e_2]\!]_{\varepsilon_2}$$

Theorem 16.4 (Compositionality)**.** *For all n-ary operators* op *and expressions $e_1, \ldots, e_n$:*

$$[\![\mathsf{op}(e_1, \ldots, e_n)]\!]\rho = F_{\mathsf{op}}([\![e_1]\!]\rho, \ldots, [\![e_n]\!]\rho)$$

where F_{op} is the semantic function for op.

16.3 F* Mechanization

Expression Semantics in F*

```
module BrrrSemantics.Expressions

open BrrrSemantics.Domains
open BrrrSemantics.Effects

(* Expression AST *)
noeq type expr =
  | EVar      : string → expr
  | ELit      : base_val → expr
  | ELam      : string → brrr_type → expr → expr
  | EApp      : expr → expr → expr
  | ELet      : string → expr → expr → expr
  | EIf       : expr → expr → expr → expr
  | EPair     : expr → expr → expr
  | EFst      : expr → expr
  | ESnd      : expr → expr
  | ESeq      : expr → expr → expr
  | EReturn   : expr → expr
  | EThrow    : expr → expr
  | ETry      : expr → string → expr → expr

(* Value domain *)
```

```
noeq type value =
  | VBase    : base_val → value
  | VClosure : env → string → expr → value
  | VPair    : value → value → value
  | VInl     : value → value
  | VInr     : value → value

and env = string → option value

(* Empty environment *)
let empty_env : env = fun _ → None

(* Extend environment *)
let extend (e:env) (x:string) (v:value) : env =
  fun y → if y = x then Some v else e y

(* Computation result *)
type result (a:Type) =
  | ROk    : a → result a
  | RErr   : value → result a
  | RDiv   : result a  (* Divergence *)

(* Computation monad *)
let comp (a:Type) = env → result a

(* Monad operations *)
let return (#a:Type) (x:a) : comp a = fun _ → ROk x

let bind (#a #b:Type) (m:comp a) (f:a → comp b) : comp b =
  fun env → match m env with
    | ROk v → f v env
    | RErr e → RErr e
    | RDiv → RDiv

let (let*) = bind

(* Semantic function *)
let rec eval (e:expr) : comp value =
  match e with
  | EVar x →
      fun env → match env x with
        | Some v → ROk v
        | None → RErr (VBase (VString ("Unbound: " ^ x)))

  | ELit v → return (VBase v)

  | ELam x _ body →
      fun env → ROk (VClosure env x body)

  | EApp f arg →
      let* fv = eval f in
      let* av = eval arg in
      (match fv with
       | VClosure cenv x body →
           eval body (extend cenv x av)
       | _ → fun _ → RErr (VBase (VString "Not a function")))
  | ELet x e1 e2 →
      let* v = eval e1 in
      fun env → eval e2 (extend env x v)

  | EIf cond e1 e2 →
      let* cv = eval cond in
      (match cv with
       | VBase (VBool true) → eval e1
       | VBase (VBool false) → eval e2
       | _ → fun _ → RErr (VBase (VString "Not a boolean")))

  | EPair e1 e2 →
      let* v1 = eval e1 in
      let* v2 = eval e2 in
      return (VPair v1 v2)

  | EFst e →
      let* v = eval e in
      (match v with
       | VPair v1 _ → return v1
       | _ → fun _ → RErr (VBase (VString "Not a pair")))

  | ESnd e →
      let* v = eval e in
      (match v with
       | VPair _ v2 → return v2
       | _ → fun _ → RErr (VBase (VString "Not a pair")))

  | ESeq e1 e2 →
      let* _ = eval e1 in
      eval e2

  | EThrow e →
      let* v = eval e in
      fun _ → RErr v

  | ETry body x handler →
      fun env → match eval body env with
        | ROk v → ROk v
        | RErr exn → eval handler (extend env x exn)
        | RDiv → RDiv

  | EReturn e →
      (* Would need continuation to implement properly *)
      eval e

(* Run expression *)
let run (e:expr) : result value = eval e empty_env
```

17. Pattern Matching

Pattern matching is the eliminator for algebraic data types.

17.1 Pattern Syntax

Definition 17.1 (Pattern Grammar).

$$\begin{aligned}
p ::= {} & x && \text{(variable)}\\
\mid {} & \_ && \text{(wildcard)}\\
\mid {} & C(p_1, \ldots, p_n) && \text{(constructor)}\\
\mid {} & (p_1, p_2) && \text{(pair)}\\
\mid {} & [p_1, \ldots, p_n] && \text{(array)}\\
\mid {} & [p_1, \ldots, p_k, \ldots p_r] && \text{(array with rest)}\\
\mid {} & p \,\mathsf{if}\, e && \text{(guard)}\\
\mid {} & p_1 \mid p_2 && \text{(or-pattern)}\\
\mid {} & p \,@\, x && \text{(as-pattern)}
\end{aligned}$$

17.2 Pattern Matching as Elimination

Definition 17.2 (Eliminator Form). Pattern matching on type T with constructors $C_1, \ldots, C_n$ is:

$$\mathsf{elim}_T : (T_1 \to R) \times \cdots \times (T_n \to R) \to T \to R$$

where $C_i : T_i \to T$.

Theorem 17.3 (Eliminator Equation).

$$\mathsf{elim}_T(f_1, \ldots, f_n)(C_i(x)) = f_i(x)$$

Definition 17.4 (Match as Eliminator).

$$\begin{aligned}
&\mathsf{match}\, e\, \{C_1(x_1) \Rightarrow e_1 \mid \cdots \mid C_n(x_n) \Rightarrow e_n\}\\
&\triangleq \mathsf{elim}_T(\lambda x_1.\, e_1, \ldots, \lambda x_n.\, e_n)(e)
\end{aligned}$$

17.3 Pattern Typing

Definition 17.5 (Pattern Typing Judgment). $\Gamma \vdash_{\mathsf{pat}} p : \tau \Rightarrow \Delta$ means pattern p matches values of type τ and binds variables in Δ.

Definition 17.6 (Pattern Typing Rules).

$$\frac{}{\Gamma \vdash_{\mathsf{pat}} x : \tau \Rightarrow x : \tau}\ \textsc{P-Var} \qquad \frac{}{\Gamma \vdash_{\mathsf{pat}} \_ : \tau \Rightarrow \cdot}\ \textsc{P-Wild}$$

$$\frac{C : \tau_1 \times \cdots \times \tau_n \to T \qquad \forall i.\, \Gamma \vdash_{\mathsf{pat}} p_i : \tau_i \Rightarrow \Delta_i}{\Gamma \vdash_{\mathsf{pat}} C(p_1, \ldots, p_n) : T \Rightarrow \Delta_1, \ldots, \Delta_n}\ \textsc{P-Ctor}$$

$$\frac{\Gamma \vdash_{\mathsf{pat}} p_1 : \tau_1 \Rightarrow \Delta_1 \qquad \Gamma \vdash_{\mathsf{pat}} p_2 : \tau_2 \Rightarrow \Delta_2}{\Gamma \vdash_{\mathsf{pat}} (p_1, p_2) : \tau_1 \times \tau_2 \Rightarrow \Delta_1, \Delta_2}\ \textsc{P-Pair}$$

$$\frac{\Gamma \vdash_{\mathsf{pat}} p : \tau \Rightarrow \Delta \qquad \Gamma, \Delta \vdash e : \mathsf{Bool}\ [\bot]}{\Gamma \vdash_{\mathsf{pat}} (p \,\mathsf{if}\, e) : \tau \Rightarrow \Delta}\ \textsc{P-Guard}$$

Definition 17.7 (Match Expression Typing).

$$\frac{\begin{array}{c}\Gamma \vdash e : T\ [\varepsilon_0]\\ \forall i.\, \Gamma \vdash_{\mathsf{pat}} p_i : T \Rightarrow \Delta_i \qquad \forall i.\, \Gamma, \Delta_i \vdash e_i : \sigma\ [\varepsilon_i] \qquad \text{patterns exhaustive}\end{array}}{\Gamma \vdash \mathsf{match}\, e\, \{p_1 \Rightarrow e_1 \mid \cdots\} : \sigma\ [\varepsilon_0 \sqcup \bigsqcup_i \varepsilon_i]}\ \textsc{T-Match}$$

17.4 Exhaustiveness and Redundancy

Definition 17.8 (Exhaustiveness)**.** Patterns $p_1, \ldots, p_n$ are exhaustive for type T iff:

$$\forall v : T.\ \exists i.\ v \in [\![p_i]\!]$$

Definition 17.9 (Redundancy)**.** Pattern p_i is redundant iff:

$$[\![p_i]\!] \subseteq \bigcup_{j<i} [\![p_j]\!]$$

17.5 F* Mechanization

Pattern Matching in F*

```
module BrrrSemantics.Patterns

open BrrrSemantics.Expressions

(* Pattern AST *)
noeq type pattern =
  | PVar    : string → pattern
  | PWild   : pattern
  | PCtor   : string → list pattern → pattern
  | PPair   : pattern → pattern → pattern
  | PArray  : list pattern → option string → pattern   (* rest pattern
    *)
  | PGuard  : pattern → expr → pattern
  | POr     : pattern → pattern → pattern
  | PAs     : pattern → string → pattern
  | PLit    : base_val → pattern

(* Bindings from pattern match *)
type bindings = list (string & value)

(* Match result *)
type match_result =
  | MatchSuccess : bindings → match_result
  | MatchFail    : match_result

(* Combine bindings *)
let combine_bindings (b1 b2:bindings) : bindings = b1 @ b2

(* Pattern matching *)
let rec match_pattern (p:pattern) (v:value) : match_result =
  match p, v with
  | PVar x, _ → MatchSuccess [(x, v)]

  | PWild, _ → MatchSuccess []

  | PLit lit, VBase lit' →
      if lit = lit' then MatchSuccess [] else MatchFail

  | PCtor name pats, v →
      (* Would need type info to destructure *)
      admit()

  | PPair p1 p2, VPair v1 v2 →
      (match match_pattern p1 v1, match_pattern p2 v2 with
       | MatchSuccess b1, MatchSuccess b2 →
```

```
            MatchSuccess (combine_bindings b1 b2)
        | _, _ → MatchFail)

  | PAs p x, v →
      (match match_pattern p v with
       | MatchSuccess binds → MatchSuccess ((x, v) :: binds)
       | MatchFail → MatchFail)

  | POr p1 p2, v →
      (match match_pattern p1 v with
       | MatchSuccess b → MatchSuccess b
       | MatchFail → match_pattern p2 v)

  | _, _ → MatchFail

(* Match expression *)
type arm = pattern & expr

let match_expr (scrutinee:value) (arms:list arm) (env:env)
    : result value =
  let rec try_arms (arms:list arm) : result value =
    match arms with
    | [] → RErr (VBase (VString "Non-exhaustive match"))
    | (pat, body) :: rest →
        match match_pattern pat scrutinee with
        | MatchSuccess binds →
            let env' = List.fold_left
              (fun e (x, v) → extend e x v) env binds in
            eval body env'
        | MatchFail → try_arms rest
  in try_arms arms

(* Exhaustiveness checking *)

(* Value space representation *)
noeq type value_space =
  | VSAll    : brrr_type → value_space            (* All values of type
      *)
  | VSCtor   : string → list value_space → value_space   (* Constructor
      *)
  | VSUnion  : list value_space → value_space     (* Union *)
  | VSEmpty  : value_space                         (* No values *)

(* Subtract pattern from value space *)
let rec subtract_pattern (vs:value_space) (p:pattern) : value_space =
  match p with
  | PWild → VSEmpty
  | PVar _ → VSEmpty
  | PCtor name pats →
      (* Remove this constructor case *)
      admit()
  | POr p1 p2 →
      subtract_pattern (subtract_pattern vs p1) p2
  | _ → vs

(* Check exhaustiveness *)
let is_exhaustive (ty:brrr_type) (patterns:list pattern) : bool =
  let remaining = List.fold_left subtract_pattern (VSAll ty) patterns
     in
  remaining = VSEmpty

(* Check redundancy *)
let rec is_useful (covered:value_space) (p:pattern) : bool =
  (* Pattern is useful if it matches something not yet covered *)
  match covered with
  | VSEmpty → true
  | VSAll _ → not (p = PWild || (match p with PVar _ → true | _ →
     false))
  | _ → admit()
```

18. Algebraic Data Types

18.1 ADT as Initial Algebra

Definition 18.1 (Polynomial Functor)**.** A polynomial functor $F : \mathsf{Type} \to \mathsf{Type}$ is built from:

$$\begin{aligned} F ::= {} & \mathsf{Id} && \text{(identity)} \\ \mid {} & K_\tau && \text{(constant } \tau\text{)} \\ \mid {} & F_1 + F_2 && \text{(coproduct)} \\ \mid {} & F_1 \times F_2 && \text{(product)} \\ \mid {} & F_1 \circ F_2 && \text{(composition)} \end{aligned}$$

Definition 18.2 (Initial Algebra)**.** For functor F, the initial F-algebra is $(\mu F, \mathsf{in}_F)$ where:

- μF is the least fixed point of F
- $\mathsf{in}_F : F(\mu F) \to \mu F$ is the constructor

Definition 18.3 (Catamorphism (Fold))**.** For any F-algebra $(C, \alpha : F(C) \to C)$, there exists a unique:

$$\mathsf{fold}_F(\alpha) : \mu F \to C$$

such that $\mathsf{fold}_F(\alpha) \circ \mathsf{in}_F = \alpha \circ F(\mathsf{fold}_F(\alpha))$

Example 18.4 (List as Initial Algebra)**.**

$$\begin{aligned} F_{\mathsf{List}[A]}(X) &= 1 + A \times X \\ \mathsf{List}[A] &= \mu X.\, 1 + A \times X \\ \mathsf{Nil} &: 1 \to \mathsf{List}[A] \\ \mathsf{Cons} &: A \times \mathsf{List}[A] \to \mathsf{List}[A] \end{aligned}$$

The fold is:

$$\mathsf{fold}(\mathsf{nil}, \mathsf{cons}) : \mathsf{List}[A] \to C$$

with equations:

$$\begin{aligned} \mathsf{fold}(\mathsf{nil}, \mathsf{cons})(\mathsf{Nil}) &= \mathsf{nil} \\ \mathsf{fold}(\mathsf{nil}, \mathsf{cons})(\mathsf{Cons}(a, xs)) &= \mathsf{cons}(a, \mathsf{fold}(\mathsf{nil}, \mathsf{cons})(xs)) \end{aligned}$$

18.2 Fold Typing

Definition 18.5 (Fold Typing Rule)**.**

$$\frac{F : \mathsf{Type} \to \mathsf{Type} \text{ polynomial} \qquad \Gamma \vdash \mathsf{alg} : F(C) \to C\ [\varepsilon]}{\Gamma \vdash \mathsf{fold}_F(\mathsf{alg}) : \mu F \to C\ [\varepsilon]}\ \textsc{T-Fold}$$

18.3 Codata and Final Coalgebras

Definition 18.6 (Final Coalgebra)**.** For functor F, the final F-coalgebra is $(\nu F, \mathsf{out}_F)$ where:

- νF is the greatest fixed point of F
- $\mathsf{out}_F : \nu F \to F(\nu F)$ is the destructor

Definition 18.7 (Anamorphism (Unfold))**.** For any F-coalgebra $(S, \gamma : S \to F(S))$, there exists a unique:

$$\mathsf{unfold}_F(\gamma) : S \to \nu F$$

such that $\mathsf{out}_F \circ \mathsf{unfold}_F(\gamma) = F(\mathsf{unfold}_F(\gamma)) \circ \gamma$

Example 18.8 (Stream as Final Coalgebra)**.**

$$\begin{aligned} F_{\mathsf{Stream}[A]}(X) &= A \times X \\ \mathsf{Stream}[A] &= \nu X.\, A \times X \\ \mathsf{head} &: \mathsf{Stream}[A] \to A \\ \mathsf{tail} &: \mathsf{Stream}[A] \to \mathsf{Stream}[A] \end{aligned}$$

The unfold is:

$$\mathsf{unfold}(\gamma) : S \to \mathsf{Stream}[A]$$

where $\gamma : S \to A \times S$.

18.4 F* Mechanization

Algebraic Data Types in F*

```
module BrrrSemantics.ADT

(* Polynomial functor representation *)
noeq type poly_functor =
  | FId     : poly_functor                                    (* Identity *)
  | FConst  : brrr_type → poly_functor                       (* Constant *)
  | FSum    : poly_functor → poly_functor → poly_functor   (* Coproduct
      *)
  | FProd   : poly_functor → poly_functor → poly_functor   (* Product *)
  | FComp   : poly_functor → poly_functor → poly_functor   (*
      Composition *)

(* Apply functor to type *)
let rec apply_functor (f:poly_functor) (x:Type) : Type =
  match f with
  | FId → x
  | FConst t → (* type from brrr_type *) unit  (* simplified *)
  | FSum f1 f2 → either (apply_functor f1 x) (apply_functor f2 x)
  | FProd f1 f2 → apply_functor f1 x & apply_functor f2 x
  | FComp f1 f2 → apply_functor f1 (apply_functor f2 x)

and either (a:Type) (b:Type) = | Left : a → either a b | Right : b →
    either a b

(* Fixed point (would need proper recursive types) *)
noeq type mu (f:poly_functor) =
  | In : apply_functor f (mu f) → mu f

(* Fold (catamorphism) *)
let rec fold (#f:poly_functor) (#c:Type)
             (alg:apply_functor f c → c) (x:mu f) : c =
  match x with
  | In fx → alg (fmap_fold alg fx)

(* Functor map for fold - would need to be derived from functor
   structure *)
and fmap_fold (#f:poly_functor) (#c:Type)
              (alg:apply_functor f c → c)
              (fx:apply_functor f (mu f))
              : apply_functor f c =
  admit() (* Depends on functor structure *)

(* List as mu (1 + A × X) *)
type list_functor (a:Type) = FSum (FConst TUnit) (FProd (FConst (* a
   *) TUnit) FId)

(* Codata: greatest fixed point *)
noeq type nu (f:poly_functor) =
  | CoIn : (unit → apply_functor f (nu f)) → nu f

(* Destructor *)
let out (#f:poly_functor) (x:nu f) : apply_functor f (nu f) =
  match x with CoIn thunk → thunk ()

(* Unfold (anamorphism) *)
let rec unfold (#f:poly_functor) (#s:Type)
               (coalg:s → apply_functor f s) (seed:s) : nu f =
  CoIn (fun () → fmap_unfold coalg (coalg seed))

and fmap_unfold (#f:poly_functor) (#s:Type)
                (coalg:s → apply_functor f s)
                (fx:apply_functor f s)
                : apply_functor f (nu f) =
  admit() (* Depends on functor structure *)

(* Stream example *)
type stream (a:Type) = nu (FProd (FConst (* a *) TUnit) FId)

let stream_head (#a:Type) (s:stream a) : unit (* would be a *) =
  match out s with (hd, _) → hd

let stream_tail (#a:Type) (s:stream a) : stream a =
  match out s with (_, tl) → tl

(* Iterate: unfold for streams *)
let iterate (#a:Type) (f:a → a) (x:a) : stream a =
  unfold (fun s → ((), (* f s *) s)) x

(* Hylomorphism: unfold then fold *)
let hylo (#f:poly_functor) (#a #b:Type)
         (alg:apply_functor f b → b)
         (coalg:a → apply_functor f a)
         (x:a) : b =
  fold alg (unfold coalg x)
```

Part VI
Metaprogramming

19. Multi-Stage Programming

This chapter grounds metaprogramming in typed multi-stage programming (MSP), replacing stringly-typed `eval` with a proper code type.

19.1 Code Type

Definition 19.1 (Code Type)**.** $\langle\tau\rangle$ is the type of code that, when executed, produces a value of type τ:

$$\langle\tau\rangle : \star \quad \text{where } \tau : \star$$

Definition 19.2 (Stage Annotation)**.** Expressions have a stage level $n \in \mathbb{N}$:

$$\Gamma \vdash^n e : \tau \; [\varepsilon]$$

Stage 0 is runtime; higher stages are compile-time/generation-time.

19.2 Quote and Splice

Definition 19.3 (Quote)**.** Quote lifts an expression one stage higher:

$$\langle e\rangle : \langle\tau\rangle \quad \text{when } e : \tau$$

Definition 19.4 (Splice)**.** Splice inserts code at the current position (lowers stage):

$${\sim}e : \tau \quad \text{when } e : \langle\tau\rangle$$

Splice is only valid inside a quote.

Definition 19.5 (Quote/Splice Typing Rules)**.**

$$\frac{\Gamma \vdash^{n+1} e : \tau \; [\varepsilon]}{\Gamma \vdash^{n} \langle e\rangle : \langle\tau\rangle \; [\varepsilon]}\ \textsc{T-Quote} \qquad \frac{\Gamma \vdash^{n} e : \langle\tau\rangle \; [\varepsilon]}{\Gamma \vdash^{n+1} {\sim}e : \tau \; [\varepsilon]}\ \textsc{T-Splice}$$

$$\frac{\Gamma \vdash^{0} e : \langle\tau\rangle \; [\varepsilon] \qquad \mathsf{closed}(e)}{\Gamma \vdash^{0} \mathsf{run}(e) : \tau \; [\langle \mathsf{DynCode} \mid \varepsilon\rangle]}\ \textsc{T-Run}$$

19.3 Cross-Stage Persistence

Definition 19.6 (Lift)**.** Lift embeds a runtime value into code:

$$\mathsf{lift} : \tau \to \langle\tau\rangle \quad \text{when } \tau \text{ is persistable}$$

Persistable types: literals, closed values, serializable data.

Definition 19.7 (Lift Typing)**.**

$$\frac{\Gamma \vdash^{n} e : \tau \; [\varepsilon] \qquad \mathsf{persistable}(\tau)}{\Gamma \vdash^{n} \mathsf{lift}(e) : \langle\tau\rangle \; [\varepsilon]}\ \textsc{T-Lift}$$

19.4 Staged Power Example

Example 19.8 (Staged Power Function)**.**

$$\begin{aligned}
&\mathsf{gen\_power} : \mathbb{N} \to \langle \mathsf{Int} \to \mathsf{Int}\rangle \\
&\mathsf{gen\_power}(0) = \langle \lambda x.\, 1\rangle \\
&\mathsf{gen\_power}(n) = \mathsf{let}\, r = \mathsf{gen\_power}(n-1) \text{ in } \langle \lambda x.\, x * ({\sim}r)\, x\rangle
\end{aligned}$$

Evaluation:

$$\begin{aligned}
\mathsf{gen\_power}(3) &= \langle \lambda x.\, x * x * x * 1\rangle \\
\mathsf{run}(\mathsf{gen\_power}(3))(2) &= 8
\end{aligned}$$

No loop overhead—multiplication is unrolled at generation time.

19.5 F* Mechanization

Multi-Stage Programming in F*

```
module BrrrSemantics.Staging

(* Code representation *)
noeq type code (a:Type) =
  | CQuote  : expr → code a
  | CLift   : a → code a
  | CSplice : code (code a) → code a

(* Stage level *)
type stage = nat

(* Staged expression *)
noeq type staged_expr =
  | SVar    : string → staged_expr
  | SLit    : base_val → staged_expr
  | SLam    : string → staged_expr → staged_expr
  | SApp    : staged_expr → staged_expr → staged_expr
  | SQuote  : staged_expr → staged_expr            (* <e> *)
  | SSplice : staged_expr → staged_expr            (* ¬e *)
  | SLift   : staged_expr → staged_expr            (* lift(e) *)
  | SRun    : staged_expr → staged_expr            (* run(e) *)

(* Type checking with stages *)
noeq type staged_type =
  | STBase  : brrr_type → staged_type
  | STCode  : staged_type → staged_type            (* Code[T] *)
  | STArrow : staged_type → staged_type → staged_type

(* Staged typing context *)
type staged_ctx = list (string & staged_type & stage)

(* Stage checking *)
let rec check_stage (ctx:staged_ctx) (e:staged_expr) (expected_stage:
    stage)
     : option staged_type =
  match e with
  | SVar x →
      (match List.find (fun (y, _, s) → y = x && s ≤ expected_stage)
          ctx with
       | Some (_, t, _) → Some t
       | None → None)

  | SLit v → Some (STBase TInt)  (* simplified *)

  | SLam x body →
      (* Would need parameter type annotation *)
      admit()

  | SQuote inner →
      (* Check inner at stage n+1, result is Code[T] at stage n *)
      (match check_stage ctx inner (expected_stage + 1) with
       | Some t → Some (STCode t)
       | None → None)
  | SSplice inner →
      (* Check inner at stage n, must be Code[T], result is T at stage
          n+1 *)
      if expected_stage = 0 then None  (* Can't splice at stage 0 *)
      else match check_stage ctx inner (expected_stage - 1) with
        | Some (STCode t) → Some t
        | _ → None

  | SLift inner →
      (* Lift value to code *)
      (match check_stage ctx inner expected_stage with
       | Some t → Some (STCode t)
       | None → None)

  | SRun inner →
      (* Run only at stage 0, requires Code[T], produces T *)
      if expected_stage ≠ 0 then None
      else match check_stage ctx inner 0 with
        | Some (STCode t) → Some t
        | _ → None

  | _ → None

(* Partial evaluation / staging *)
let rec stage_reduce (e:staged_expr) : staged_expr =
  match e with
  | SQuote (SSplice inner) → stage_reduce inner  (* ¬<e> = e *)
  | SSplice (SQuote inner) → stage_reduce inner  (* <¬e> = e *)

  | SQuote inner → SQuote (stage_reduce inner)
  | SSplice inner → SSplice (stage_reduce inner)

  | SApp (SLam x body) arg →
      stage_reduce (subst x arg body)

  | SApp f arg → SApp (stage_reduce f) (stage_reduce arg)
  | SLam x body → SLam x (stage_reduce body)
  | _ → e

and subst (x:string) (v:staged_expr) (e:staged_expr) : staged_expr =
  match e with
  | SVar y → if y = x then v else e
  | SLam y body → if y = x then e else SLam y (subst x v body)
  | SApp f arg → SApp (subst x v f) (subst x v arg)
  | SQuote inner → SQuote (subst x v inner)
  | SSplice inner → SSplice (subst x v inner)
  | _ → e

(* Generate power function *)
let rec gen_power (n:nat) : staged_expr =
  if n = 0 then
    SQuote (SLam "x" (SLit (VInt 1)))
  else
    let rest = gen_power (n - 1) in
    SQuote (SLam "x"
      (SApp (SApp (SVar "*") (SVar "x"))
            (SApp (SSplice rest) (SVar "x"))))
```

```
(* After staging: gen_power 3 reduces to code for x*x*x*1 *)
```

20. Typed Reflection

20.1 TypeRep

Definition 20.1 (Type Representation)**.** $\mathsf{TypeRep}[\tau]$ is a runtime representation of type τ:

$$\mathsf{TypeRep}[\tau] : \star$$

Definition 20.2 (TypeRep Operations)**.**

$$\begin{aligned} \mathsf{typeOf} &: \forall \tau.\, \tau \to \mathsf{TypeRep}[\tau] \\ \mathsf{cast} &: \forall \tau \sigma.\, \tau \to \mathsf{TypeRep}[\sigma] \to \mathsf{Option}[\sigma] \\ \mathsf{eqType} &: \forall \tau \sigma.\, \mathsf{TypeRep}[\tau] \to \mathsf{TypeRep}[\sigma] \to \mathsf{Option}[\tau =:= \sigma] \end{aligned}$$

where $\tau =:= \sigma$ is a type equality witness (Leibniz equality).

Definition 20.3 (TypeRep Typing)**.**

$$\frac{\Gamma \vdash e : \tau\ [\varepsilon]}{\Gamma \vdash \mathsf{typeOf}(e) : \mathsf{TypeRep}[\tau]\ [\varepsilon]}\ \textsc{T-TypeOf} \qquad \frac{\Gamma \vdash e : \tau\ [\varepsilon_1] \qquad \Gamma \vdash r : \mathsf{TypeRep}[\sigma]\ [\varepsilon_2]}{\Gamma \vdash \mathsf{cast}(e, r) : \mathsf{Option}[\sigma]\ [\varepsilon_1 \sqcup \varepsilon_2]}\ \textsc{T-Cast}$$

$$\frac{\Gamma \vdash r_1 : \mathsf{TypeRep}[\tau]\ [\varepsilon_1] \qquad \Gamma \vdash r_2 : \mathsf{TypeRep}[\sigma]\ [\varepsilon_2]}{\Gamma \vdash \mathsf{eqType}(r_1, r_2) : \mathsf{Option}[\tau =:= \sigma]\ [\varepsilon_1 \sqcup \varepsilon_2]}\ \textsc{T-EqType}$$

20.2 Type Equality Witness

Definition 20.4 (Leibniz Equality)**.**

$$\tau =:= \sigma \triangleq \forall F : \star \to \star.\, F[\tau] \to F[\sigma]$$

If we have $\mathsf{eq} : \tau =:= \sigma$, we can coerce any $F[\tau]$ to $F[\sigma]$.

Definition 20.5 (Coercion via Equality)**.**

$$\mathsf{coerce} : \forall \tau \sigma.\, (\tau =:= \sigma) \to \tau \to \sigma$$

$$\mathsf{coerce}(\mathsf{eq}, x) = \mathsf{eq}[\lambda T.\, T](x)$$

20.3 Dynamic Type

Definition 20.6 (Dynamic Type)**.** For interop with dynamically-typed code:

$$\mathsf{Dynamic} = \exists \tau.\, (\tau \times \mathsf{TypeRep}[\tau])$$

Definition 20.7 (Dynamic Operations)**.**

$$\begin{aligned} \mathsf{toDyn} &: \forall \tau.\, \tau \to \mathsf{Dynamic} \\ \mathsf{fromDyn} &: \forall \tau.\, \mathsf{Dynamic} \to \mathsf{TypeRep}[\tau] \to \mathsf{Option}[\tau] \end{aligned}$$

20.4 F* Mechanization

Typed Reflection in F*

```
module BrrrSemantics.Reflection

(* TypeRep as GADT *)
noeq type type_rep : Type → Type =
  | TRInt     : type_rep int
  | TRBool   : type_rep bool
  | TRString : type_rep string
  | TRUnit   : type_rep unit
  | TROption : #a:Type → type_rep a → type_rep (option a)
  | TRList   : #a:Type → type_rep a → type_rep (list a)
  | TRPair   : #a:Type → #b:Type → type_rep a → type_rep b → type_rep
     (a & b)
  | TRArrow  : #a:Type → #b:Type → type_rep a → type_rep b → type_rep
     (a → b)

(* Type equality witness *)
type type_eq (a:Type) (b:Type) = a → b

(* Reflexivity *)
let type_eq_refl (#a:Type) : type_eq a a = fun x → x

(* Symmetry *)
let type_eq_sym (#a #b:Type) (eq:type_eq a b) : type_eq b a =
  admit() (* Requires type-level tricks *)

(* Check type equality *)
let rec eq_type_rep (#a #b:Type) (ra:type_rep a) (rb:type_rep b)
    : option (type_eq a b) =
  match ra, rb with
  | TRInt, TRInt → Some type_eq_refl
  | TRBool, TRBool → Some type_eq_refl
  | TRString, TRString → Some type_eq_refl
  | TRUnit, TRUnit → Some type_eq_refl

  | TROption ra', TROption rb' →
      (match eq_type_rep ra' rb' with
       | Some _ → Some (admit())  (* Would need proper transport *)
       | None → None)

  | TRList ra', TRList rb' →
      (match eq_type_rep ra' rb' with
       | Some _ → Some (admit())
       | None → None)

  | TRPair ra1 ra2, TRPair rb1 rb2 →
      (match eq_type_rep ra1 rb1, eq_type_rep ra2 rb2 with
       | Some _, Some _ → Some (admit())
       | _, _ → None)

  | _, _ → None

(* Get TypeRep for a value (requires type class in real impl) *)
let typeof_int (x:int) : type_rep int = TRInt
let typeof_bool (x:bool) : type_rep bool = TRBool

(* Safe cast using TypeRep *)
let cast (#a #b:Type) (x:a) (target:type_rep b) (source:type_rep a)
    : option b =
  match eq_type_rep source target with
  | Some eq → Some (eq x)
  | None → None
```

```
(* Dynamic type *)
noeq type dynamic =
  | Dyn : #a:Type → type_rep a → a → dynamic

(* Create dynamic *)
let to_dyn (#a:Type) (rep:type_rep a) (x:a) : dynamic =
  Dyn rep x

(* Extract from dynamic *)
let from_dyn (#a:Type) (d:dynamic) (target:type_rep a) : option a =
  match d with
  | Dyn rep x →
      match eq_type_rep rep target with
      | Some eq → Some (eq x)
      | None → None

(* Example usage *)
let example_reflection () : option int =
  let d = to_dyn TRInt 42 in
  from_dyn d TRInt  (* Some 42 *)

let example_cast_fail () : option string =
  let d = to_dyn TRInt 42 in
  from_dyn d TRString  (* None *)
```

21. Macros and Syntax Transformers

21.1 Syntax Types

Definition 21.1 (Syntax Representation)**.** Typed syntax fragments:

$$\begin{aligned} \mathsf{Expr}[\tau] &\quad \text{(expression producing } \tau) \\ \mathsf{Stmt} &\quad \text{(statement)} \\ \mathsf{Pat}[\tau] &\quad \text{(pattern matching } \tau) \\ \mathsf{Type} &\quad \text{(type expression)} \end{aligned}$$

Definition 21.2 (Macro Type)**.** A macro is a function on syntax:

$$\mathsf{macro} : \mathsf{Syntax}_{\mathsf{in}} \to \mathsf{Syntax}_{\mathsf{out}}$$

21.2 Quasi-Quotation

Definition 21.3 (Quasi-Quote)**.** Quasi-quotation allows building syntax with holes:

$$\mathsf{quote}\{e\} : \mathsf{Expr}[\tau] \quad \text{when } e : \tau$$

With antiquotation:

$$\mathsf{quote}\{\ldots \$x \ldots\} \quad \text{where } x : \mathsf{Expr}[\sigma]$$

21.3 Hygiene

Definition 21.4 (Hygienic Macro)**.** A macro is hygienic if:

1. Variables introduced by the macro don't capture user variables
2. Variables in user code aren't captured by macro bindings

Achieved via fresh name generation with scope tracking.

21.4 Macro Typing

Definition 21.5 (Macro Typing Rules)**.**

$$\frac{\Gamma, x_1 : \mathsf{Syntax}_1, \ldots, x_n : \mathsf{Syntax}_n \vdash e : \mathsf{Syntax}_{\mathsf{out}}}{\Gamma \vdash \mathsf{macro}\, m(x_1, \ldots, x_n) = e : \mathsf{Macro}[\mathsf{Syntax}_1 \times \cdots \times \mathsf{Syntax}_n \to \mathsf{Syntax}_{\mathsf{out}}]}\ \textsc{T-MacroDef}$$

$$\frac{\Gamma \vdash m : \mathsf{Macro}[\mathsf{Syntax}_{\mathsf{in}} \to \mathsf{Syntax}_{\mathsf{out}}] \qquad \Gamma \vdash s : \mathsf{Syntax}_{\mathsf{in}}}{\Gamma \vdash m!(s) : \mathsf{Syntax}_{\mathsf{out}}}\ \textsc{T-MacroApp}$$

21.5 F* Mechanization

Macros in F*

```
module BrrrSemantics.Macros

(* Syntax types *)
noeq type syntax =
  | SynExpr : brrr_type → expr → syntax
  | SynStmt : expr → syntax
  | SynPat  : brrr_type → pattern → syntax
  | SynType : brrr_type → syntax

(* Unique identifier for hygiene *)
type ident = {
  name  : string;
  scope : nat;       (* Scope level for hygiene *)
  mark  : nat        (* Unique mark for macro expansion *)
}

(* Fresh name generation *)
let fresh_counter : ref nat = ref 0

let fresh_ident (base:string) (scope:nat) : ident =
  let mark = !fresh_counter in
  fresh_counter := mark + 1;
  { name = base; scope = scope; mark = mark }

(* Quasi-quotation result *)
noeq type quasi_quote =
  | QQLit    : syntax → quasi_quote
  | QQSplice : string → quasi_quote       (* $var to be filled *)
  | QQSeq    : list quasi_quote → quasi_quote

(* Fill holes in quasi-quote *)
let rec fill_quasi (qq:quasi_quote) (env:list (string & syntax)) :
    option syntax =
  match qq with
  | QQLit s → Some s
  | QQSplice var →
      List.assoc var env
  | QQSeq parts →
      let filled = List.map (fun q → fill_quasi q env) parts in
      if List.for_all Some? filled then
        (* Would combine syntaxes *)
        admit()
      else None

(* Macro definition *)
```

```
noeq type macro_def = {
  name   : string;
  params : list (string & (* syntax type *) string);
  body   : quasi_quote
}

(* Macro expansion *)
let expand_macro (m:macro_def) (args:list syntax) : option syntax =
  if List.length args ≠ List.length m.params then None
  else
    let bindings = List.map2 (fun (name, _) arg → (name, arg)) m.
        params args in
    fill_quasi m.body bindings

(* Example: assert_eq macro *)
let assert_eq_macro : macro_def = {
  name = "assert_eq";
  params = [("left", "Expr"); ("right", "Expr")];
  body = QQSeq [
    QQLit (SynStmt (EIf
      (EApp (EApp (EVar "!=") (EVar "$left")) (EVar "$right"))
      (EApp (EVar "panic") (ELit (VString "assertion failed")))
      EUnit));
  ]
}

(* Hygiene: rename variables in macro output *)
let rec rename_syntax (renames:list (string & string)) (s:syntax) :
   syntax =
  match s with
  | SynExpr t e → SynExpr t (rename_expr renames e)
  | SynStmt e → SynStmt (rename_expr renames e)
  | _ → s

and rename_expr (renames:list (string & string)) (e:expr) : expr =
  match e with
  | EVar x →
      (match List.assoc x renames with
       | Some y → EVar y
       | None → e)
  | ELam x body →
      let x' = fresh_ident x 0 in
      ELam x'.name (rename_expr ((x, x'.name) :: renames) body)
  | ELet x e1 e2 →
      let x' = fresh_ident x 0 in
      ELet x'.name (rename_expr renames e1)
                   (rename_expr ((x, x'.name) :: renames) e2)
  | EApp f arg → EApp (rename_expr renames f) (rename_expr renames arg
     )
  | EIf c t e → EIf (rename_expr renames c)
                    (rename_expr renames t)
                    (rename_expr renames e)
  | _ → e

(* Hygienic macro expansion *)
let expand_hygienic (m:macro_def) (args:list syntax) (scope:nat) :
   option syntax =
  match expand_macro m args with
  | Some result →
      (* Generate fresh names for all binders in result *)
      Some (rename_syntax [] result)
  | None → None
```

22. Decorators

22.1 Decorator Type

Definition 22.1 (Decorator). A decorator transforms a function:

$$\mathsf{Decorator}[\tau_1 \xrightarrow{\varepsilon_1} \sigma_1, \tau_2 \xrightarrow{\varepsilon_2} \sigma_2] = (\tau_1 \xrightarrow{\varepsilon_1} \sigma_1) \to (\tau_2 \xrightarrow{\varepsilon_2} \sigma_2)$$

Definition 22.2 (Common Decorator Types).

$$@\mathsf{cache} : \forall \alpha\beta\varepsilon.\, (\alpha \xrightarrow{\varepsilon} \beta) \to (\alpha \xrightarrow{\varepsilon \sqcup \mathsf{Alloc}} \beta)$$
$$@\mathsf{log} : \forall \alpha\beta\varepsilon.\, (\alpha \xrightarrow{\varepsilon} \beta) \to (\alpha \xrightarrow{\varepsilon \sqcup \mathsf{IO}} \beta)$$
$$@\mathsf{retry}(n) : \forall \alpha\beta\varepsilon.\, (\alpha \xrightarrow{\varepsilon \sqcup \mathsf{Throw}[E]} \beta) \to (\alpha \xrightarrow{\varepsilon \sqcup \mathsf{Throw}[E]} \beta)$$

22.2 Decorator Typing

Definition 22.3 (Decorator Application).

$$\frac{\Gamma \vdash d : (\tau_1 \xrightarrow{\varepsilon_1} \sigma_1) \to (\tau_2 \xrightarrow{\varepsilon_2} \sigma_2)\ [\varepsilon_d] \qquad \Gamma \vdash f : \tau_1 \xrightarrow{\varepsilon_1} \sigma_1\ [\varepsilon_f]}{\Gamma \vdash @d\, f : \tau_2 \xrightarrow{\varepsilon_2} \sigma_2\ [\varepsilon_d \sqcup \varepsilon_f]}\ \textsc{T-Decorator}$$

Definition 22.4 (Decorator Composition).

$$@d_1\, @d_2\, f \equiv d_1(d_2(f))$$

22.3 F* Mechanization

Decorators in F*

```
module BrrrSemantics.Decorators

(* Decorator type *)
type decorator (a:Type) (b:Type) (c:Type) (d:Type) =
  (a → b) → (c → d)

(* Identity decorator *)
let id_decorator (#a #b:Type) : decorator a b a b =
  fun f → f

(* Compose decorators *)
let compose_decorator (#a #b #c #d #e #f:Type)
    (d1:decorator c d e f) (d2:decorator a b c d)
    : decorator a b e f =
  fun g → d1 (d2 g)

(* Cache decorator *)
let cache_decorator (#a #b:Type) (hash:a → int) (eq:a → a → bool)
    : decorator a b a b =
  fun f →
    let cache : ref (list (a & b)) = ref [] in
    fun x →
      match List.find (fun (k, _) → eq k x) !cache with
      | Some (_, v) → v
      | None →
          let v = f x in
          cache := (x, v) :: !cache;
          v

(* Log decorator *)
let log_decorator (#a #b:Type) (name:string) : decorator a b a b =
  fun f →
    fun x →
      (* print_string ("Calling " ^ name); *)
      let result = f x in
      (* print_string ("Returned from " ^ name); *)
      result

(* Retry decorator *)
let rec retry_decorator (#a #b:Type) (n:nat) : decorator a b a b =
  fun f →
    fun x →
      if n = 0 then f x
      else
        try f x
        with _ → retry_decorator (n - 1) f x

(* Timing decorator *)
let time_decorator (#a #b:Type) : decorator a b a (b & int) =
  fun f →
    fun x →
      let start = (* get_time () *) 0 in
      let result = f x in
      let elapsed = (* get_time () - start *) 0 in
      (result, elapsed)

(* Decorator application syntax *)
let apply_decorator (#a #b #c #d:Type)
    (dec:decorator a b c d) (f:a → b) : c → d =
  dec f

(* Chained application *)
let example_decorated_function () =
  let base_fn (x:int) : int = x * 2 in
  let decorated =
    apply_decorator (log_decorator "double")
      (apply_decorator (cache_decorator (fun x → x) (=)) base_fn)
  in
  decorated 21  (* Returns 42, with logging and caching *)
```

Part VII
Concurrency & Session Types

23. Session Type Foundations

Session types provide a type discipline for communication protocols, ensuring that interacting processes follow agreed-upon message sequences.

23.1 Binary Session Types

Definition 23.1 (Session Type Grammar).

$$\begin{aligned} S ::= & !\tau.S && \text{(send } \tau\text{, continue as } S\text{)} \\ | & ?\tau.S && \text{(receive } \tau\text{, continue as } S\text{)} \\ | & \ S_1 \oplus S_2 && \text{(internal choice: select)} \\ | & \ S_1 \,\&\, S_2 && \text{(external choice: branch)} \\ | & \ \mu X.S && \text{(recursive session)} \\ | & \ X && \text{(session variable)} \\ | & \ \mathsf{end} && \text{(session termination)} \end{aligned}$$

Definition 23.2 (Session Duality). The dual of a session type $\overline{S}$ swaps sends/receives and choices:

$$\begin{aligned} \overline{!\tau.S} &= ?\tau.\overline{S} \\ \overline{?\tau.S} &= !\tau.\overline{S} \\ \overline{S_1 \oplus S_2} &= \overline{S_1} \,\&\, \overline{S_2} \\ \overline{S_1 \,\&\, S_2} &= \overline{S_1} \oplus \overline{S_2} \\ \overline{\mu X.S} &= \mu X.\overline{S} \\ \overline{X} &= X \\ \overline{\mathsf{end}} &= \mathsf{end} \end{aligned}$$

Theorem 23.3 (Duality Involution). *For all session types S:*

$$\overline{\overline{S}} = S$$

Proof. By structural induction on S. Each case follows directly from the definition of duality. □

23.2 Session Typing Rules

Definition 23.4 (Session Context). A session context Σ maps channel names to session types:

$$\Sigma = c_1 : S_1, \ldots, c_n : S_n$$

Definition 23.5 (Session Typing Judgment). $\Sigma; \Gamma \vdash P$ means process P is well-typed under session context Σ and value context Γ.

Definition 23.6 (Core Session Typing Rules).

$$\frac{\Sigma, c : S; \Gamma \vdash P \qquad \Gamma \vdash e : \tau\ [\varepsilon]}{\Sigma, c : !\tau.S; \Gamma \vdash c!\langle e \rangle.P}\ \textsc{S-Send} \qquad \frac{\Sigma, c : S; \Gamma, x : \tau \vdash P}{\Sigma, c : ?\tau.S; \Gamma \vdash c?(x).P}\ \textsc{S-Recv}$$

$$\frac{\Sigma, c : S_i; \Gamma \vdash P}{\Sigma, c : S_1 \oplus S_2; \Gamma \vdash c \triangleleft l_i.P}\ \textsc{S-Select} \qquad \frac{\Sigma, c : S_1; \Gamma \vdash P_1 \qquad \Sigma, c : S_2; \Gamma \vdash P_2}{\Sigma, c : S_1 \ \&\ S_2; \Gamma \vdash c \triangleright \{l_1 : P_1, l_2 : P_2\}}\ \textsc{S-Branch}$$

$$\frac{\Sigma_1; \Gamma \vdash P \qquad \Sigma_2; \Gamma \vdash Q \qquad \Sigma_1 \cap \Sigma_2 = \emptyset}{\Sigma_1, \Sigma_2; \Gamma \vdash P \mid Q}\ \textsc{S-Par} \qquad \frac{\Sigma, c : S, d : \overline{S}; \Gamma \vdash P}{\Sigma; \Gamma \vdash (\nu c\, d)P}\ \textsc{S-New}$$

$$\frac{}{c : \mathsf{end}; \Gamma \vdash \mathbf{0}}\ \textsc{S-End}$$

23.3 Session Type Safety

Theorem 23.7 (Session Fidelity). *If $\Sigma; \Gamma \vdash P$ and $P \longrightarrow P'$, then there exists Σ' such that $\Sigma'; \Gamma \vdash P'$ and the session types progress correctly.*

Theorem 23.8 (Communication Safety). *Well-typed processes do not have communication mismatches:*

- *No message type errors (sending wrong type)*
- *No deadlocks from protocol violations*
- *No orphan messages*

23.4 F* Mechanization

Session Types in F*

```
module BrrrSemantics.SessionTypes

(* Session type *)
noeq type session_type =
  | SSend   : brrr_type → session_type → session_type   (* !τ.S *)
  | SRecv   : brrr_type → session_type → session_type   (* ?τ.S *)
  | SSelect : session_type → session_type → session_type   (* S1 ⊕ S2
    *)
  | SBranch : session_type → session_type → session_type   (* S1 & S2
    *)
  | SRec    : string → session_type → session_type      (* μX.S *)
  | SVar    : string → session_type                      (* X *)
  | SEnd    : session_type                                (* end *)

(* Session duality *)
let rec dual (s:session_type) : session_type =
  match s with
  | SSend t cont → SRecv t (dual cont)
  | SRecv t cont → SSend t (dual cont)
  | SSelect s1 s2 → SBranch (dual s1) (dual s2)
  | SBranch s1 s2 → SSelect (dual s1) (dual s2)
  | SRec x body → SRec x (dual body)
  | SVar x → SVar x
  | SEnd → SEnd

(* Duality is involution *)
let rec dual_involution (s:session_type)
    : Lemma (ensures dual (dual s) = s) (decreases s) =
  match s with
  | SSend t cont → dual_involution cont
  | SRecv t cont → dual_involution cont
  | SSelect s1 s2 → dual_involution s1; dual_involution s2
  | SBranch s1 s2 → dual_involution s1; dual_involution s2
  | SRec x body → dual_involution body
  | SVar _ → ()
  | SEnd → ()

(* Process syntax *)
noeq type process =
  | PSend   : string → expr → process → process        (* c!<e>.P *)
  | PRecv   : string → string → process → process      (* c?(x).P *)
  | PSelect : string → string → process → process      (* c ◁ l.P *)
  | PBranch : string → list (string & process) → process   (* c ▷ {l:P}
      *)
  | PPar    : process → process → process                (* P | Q *)
  | PNew    : string → string → session_type → process → process   (* (
     νc d)P *)
  | PEnd    : process                                      (* 0 *)
  | PRec    : string → process → process                 (* rec X.P *)
  | PVar    : string → process                            (* X *)

(* Session context *)
type session_ctx = list (string & session_type)

(* Lookup in session context *)
let lookup_session (c:string) (ctx:session_ctx) : option session_type
    =
  List.assoc c ctx

(* Remove from session context *)
let remove_session (c:string) (ctx:session_ctx) : session_ctx =
  List.filter (fun (c', _) → c' ≠ c) ctx

(* Update session context *)
let update_session (c:string) (s:session_type) (ctx:session_ctx) :
    session_ctx =
  (c, s) :: remove_session c ctx

(* Session type checking *)
let rec check_process (sctx:session_ctx) (vctx:env) (p:process)
     : option session_ctx =
  match p with
  | PSend c e cont →
      (match lookup_session c sctx with
       | Some (SSend t s) →
           (* Check e has type t, then continue *)
           check_process (update_session c s sctx) vctx cont
       | _ → None)

  | PRecv c x cont →
      (match lookup_session c sctx with
       | Some (SRecv t s) →
           check_process (update_session c s sctx) (extend vctx x (
               VBase VUnit)) cont
       | _ → None)
  | PSelect c l cont →
      (match lookup_session c sctx with
       | Some (SSelect s1 s2) →
           let s = if l = "left" then s1 else s2 in
           check_process (update_session c s sctx) vctx cont
       | _ → None)

  | PBranch c branches →
      (match lookup_session c sctx with
       | Some (SBranch s1 s2) →
           (* Check both branches *)
           admit()
       | _ → None)

  | PPar p1 p2 →
      (* Split context *)
      admit()

  | PNew c d s cont →
      check_process ((c, s) :: (d, dual s) :: sctx) vctx cont

  | PEnd →
      if List.for_all (fun (_, s) → s = SEnd) sctx
      then Some []
      else None

  | _ → None
```

24. Multiparty Session Types

24.1 Global Types

Definition 24.1 (Global Type Grammar)**.** Global types describe the interaction from a global perspective:

$$\begin{aligned} G ::= {} & p \to q : \tau.G && \text{(message from } p \text{ to } q\text{)} \\ & \mid p \to q : \{l_i : G_i\}_{i \in I} && \text{(labeled choice)} \\ & \mid \mu X.G && \text{(recursion)} \\ & \mid X && \text{(variable)} \\ & \mid \mathsf{end} && \text{(termination)} \end{aligned}$$

Definition 24.2 (Projection)**.** Projection $G \upharpoonright p$ extracts participant p's view from global type

G:

$$(q \to r : \tau.G) \upharpoonright p = \begin{cases} !\langle r, \tau\rangle.(G \upharpoonright p) & \text{if } p = q \\ ?\langle q, \tau\rangle.(G \upharpoonright p) & \text{if } p = r \\ G \upharpoonright p & \text{otherwise} \end{cases}$$

$$(q \to r : \{l_i : G_i\}) \upharpoonright p = \begin{cases} \bigoplus_i l_i.(G_i \upharpoonright p) & \text{if } p = q \\ \&_i l_i.(G_i \upharpoonright p) & \text{if } p = r \\ \sqcup_i (G_i \upharpoonright p) & \text{otherwise} \end{cases}$$

$$\begin{aligned} (\mu X.G) \upharpoonright p &= \mu X.(G \upharpoonright p) \\ X \upharpoonright p &= X \\ \mathsf{end} \upharpoonright p &= \mathsf{end} \end{aligned}$$

where $\sqcup$ is the merge operator requiring compatible branches.

24.2 Merge Operator

Definition 24.3 (Session Type Merge)**.** The merge $S_1 \sqcup S_2$ combines compatible session types:

$$\begin{aligned} !\langle q, \tau\rangle.S_1 \sqcup !\langle q, \tau\rangle.S_2 &= !\langle q, \tau\rangle.(S_1 \sqcup S_2) \\ ?\langle q, \tau\rangle.S_1 \sqcup ?\langle q, \tau\rangle.S_2 &= ?\langle q, \tau\rangle.(S_1 \sqcup S_2) \\ (S_1 \& S_1') \sqcup (S_2 \& S_2') &= (S_1 \sqcup S_2) \& (S_1' \sqcup S_2') \\ \mathsf{end} \sqcup \mathsf{end} &= \mathsf{end} \end{aligned}$$

Merge is undefined for incompatible types.

Theorem 24.4 (Projection Consistency)**.** *If global type G is well-formed and projectable to all participants, then for any two participants p, q with dual interactions, their projections are compatible.*

24.3 Well-Formedness

Definition 24.5 (Well-Formed Global Type)**.** A global type G is well-formed iff:

1. Every participant appears in at least one interaction
2. Projection is defined for all participants
3. Recursive types are contractive (guarded)

24.4 F* Mechanization

Multiparty Session Types in F*

```
module BrrrSemantics.MultipartySession

(* Participant identifier *)
type participant = string

(* Global type *)
noeq type global_type =
  | GMsg     : participant → participant → brrr_type → global_type →
     global_type
  | GChoice : participant → participant → list (string & global_type)
     → global_type
  | GRec     : string → global_type → global_type
  | GVar     : string → global_type
  | GEnd     : global_type
```

```
(* Local (endpoint) type *)
noeq type local_type =
  | LSend   : participant → brrr_type → local_type → local_type
  | LRecv   : participant → brrr_type → local_type → local_type
  | LSelect : participant → list (string & local_type) → local_type
  | LBranch : participant → list (string & local_type) → local_type
  | LRec    : string → local_type → local_type
  | LVar    : string → local_type
  | LEnd    : local_type

(* Merge two local types *)
let rec merge_local (l1 l2:local_type) : option local_type =
  match l1, l2 with
  | LEnd, LEnd → Some LEnd

  | LSend p t1 cont1, LSend p' t2 cont2 →
      if p = p' && t1 = t2 then
        match merge_local cont1 cont2 with
        | Some cont → Some (LSend p t1 cont)
        | None → None
      else None

  | LRecv p t1 cont1, LRecv p' t2 cont2 →
      if p = p' && t1 = t2 then
        match merge_local cont1 cont2 with
        | Some cont → Some (LRecv p t1 cont)
        | None → None
      else None

  | LBranch p bs1, LBranch p' bs2 →
      if p = p' then
        (* Merge branch by branch *)
        admit()
      else None

  | _, _ → None

(* Project global type to participant *)
let rec project (g:global_type) (p:participant) : option local_type =
  match g with
  | GMsg sender receiver ty cont →
      (match project cont p with
       | Some cont' →
           if p = sender then Some (LSend receiver ty cont')
           else if p = receiver then Some (LRecv sender ty cont')
           else Some cont'
       | None → None)

  | GChoice sender receiver branches →
      let projected = List.map (fun (l, g') →
        match project g' p with
        | Some l' → Some (l, l')
        | None → None) branches in
      if List.for_all Some? projected then
        let branches' = List.map (fun (Some x) → x) projected in
        if p = sender then Some (LSelect receiver branches')
        else if p = receiver then Some (LBranch sender branches')
        else
          (* Merge all branches for non-involved participant *)
          List.fold_left (fun acc (_, l) →
            match acc with
            | Some acc' → merge_local acc' l
            | None → None
          ) (Some LEnd) branches'
      else None

  | GRec x body →
      (match project body p with
       | Some body' → Some (LRec x body')
       | None → None)

  | GVar x → Some (LVar x)

  | GEnd → Some LEnd

(* Get all participants from global type *)
let rec participants (g:global_type) : list participant =
  match g with
  | GMsg s r _ cont → s :: r :: participants cont
  | GChoice s r branches →
      s :: r :: List.concat (List.map (fun (_, g') → participants g')
         branches)
  | GRec _ body → participants body
  | GVar _ → []
  | GEnd → []

(* Check well-formedness *)
let well_formed (g:global_type) : bool =
  let parts = List.dedup (participants g) in
  List.for_all (fun p → Some? (project g p)) parts

(* Example: Two-buyer protocol *)
let two_buyer_protocol : global_type =
  GMsg "Buyer1" "Seller" TString (                     (* Buyer1 →
     Seller: title *)
  GMsg "Seller" "Buyer1" (TInt {width=I32; sign=Signed}) (   (* Seller
     → Buyer1: price *)
  GMsg "Seller" "Buyer2" (TInt {width=I32; sign=Signed}) (   (* Seller
     → Buyer2: price *)
  GMsg "Buyer1" "Buyer2" (TInt {width=I32; sign=Signed}) (   (* Buyer1
     → Buyer2: share *)
  GChoice "Buyer2" "Seller" [                          (* Buyer2 →
     Seller: choice *)
    ("ok", GMsg "Buyer2" "Seller" TString GEnd);       (* ok: address
       *)
    ("quit", GEnd)                                     (* quit: done *)
  ]))))
```

25. Deadlock Freedom

25.1 Priority-Based Typing

Definition 25.1 (Prioritized Session Type)**.** A prioritized session type annotates each action with a priority level:

$$S ::= !^n\tau.S \mid ?^n\tau.S \mid \ldots$$

where $n \in \mathbb{N}$ is the priority.

Definition 25.2 (Priority Ordering)**.** For deadlock freedom, priorities must be consistently ordered:

- If c sends at priority n and d receives at priority m, and c and d can interact, then $n < m$ or $m < n$
- No circular dependencies in priority ordering

Theorem 25.3 (Deadlock Freedom)**.** *If all session types are consistently prioritized and all processes are well-typed, then the system is deadlock-free.*

25.2 Cycle Detection

Definition 25.4 (Dependency Graph)**.** The dependency graph $\mathcal{D}(P)$ has:

- Nodes: channel names
- Edges: $c \to d$ if process waits on c before acting on d

Theorem 25.5 (Acyclicity Implies Deadlock Freedom)**.** *If $\mathcal{D}(P)$ is acyclic, then P is deadlock-free.*

25.3 F* Mechanization

Deadlock Freedom in F*

```
module BrrrSemantics.DeadlockFreedom

(* Priority level *)
type priority = nat

(* Prioritized session type *)
noeq type pri_session =
  | PriSend   : priority → brrr_type → pri_session → pri_session
  | PriRecv   : priority → brrr_type → pri_session → pri_session
  | PriSelect : priority → pri_session → pri_session → pri_session
  | PriBranch : priority → pri_session → pri_session → pri_session
  | PriEnd    : pri_session

(* Get minimum priority in session type *)
let rec min_priority (s:pri_session) : option priority =
  match s with
  | PriSend p _ cont →
      (match min_priority cont with
       | Some p' → Some (min p p')
       | None → Some p)
  | PriRecv p _ cont →
      (match min_priority cont with
       | Some p' → Some (min p p')
       | None → Some p)
  | PriSelect p s1 s2 →
```

```
      let m1 = min_priority s1 in
      let m2 = min_priority s2 in
      Some p  (* Simplified *)
  | PriBranch p s1 s2 →
      Some p
  | PriEnd → None

(* Check priority consistency for dual session types *)
let priority_consistent (s1 s2:pri_session) : bool =
  match s1, s2 with
  | PriSend p1 _ _, PriRecv p2 _ _ → p1 < p2 || p2 < p1
  | PriRecv p1 _ _, PriSend p2 _ _ → p1 < p2 || p2 < p1
  | _, _ → true

(* Dependency graph as adjacency list *)
type dep_graph = list (string & list string)

(* Build dependency graph from process *)
let rec build_dep_graph (p:process) : dep_graph =
  match p with
  | PSend c _ (PRecv d _ cont) →
      (* Wait on d after sending on c *)
      (c, [d]) :: build_dep_graph cont
  | PRecv c _ (PSend d _ cont) →
      (* Wait on c before sending on d *)
      (c, [d]) :: build_dep_graph cont
  | PPar p1 p2 →
      build_dep_graph p1 @ build_dep_graph p2
  | PNew _ _ _ cont → build_dep_graph cont
  | _ → []

(* Check if graph has cycle (simplified DFS) *)
let rec has_cycle (g:dep_graph) (visited:list string) (node:string) :
   bool =
  if List.mem node visited then true
  else match List.assoc node g with
    | Some neighbors →
        List.∃ (has_cycle g (node :: visited)) neighbors
    | None → false

(* Check deadlock freedom *)
let is_deadlock_free (p:process) : bool =
  let g = build_dep_graph p in
  let nodes = List.map fst g in
  not (List.∃ (has_cycle g []) nodes)

(* Lock ordering type for preventing mutex deadlocks *)
type lock_order = list string  (* Locks in acquisition order *)

(* Check lock acquisition respects order *)
let respects_lock_order (acquired:list string) (order:lock_order) :
   bool =
  (* All acquired locks should appear in order *)
  let positions = List.map (fun l →
    match List.find_index ((=) l) order with
    | Some i → i
    | None → -1) acquired in
```

```
(* Check positions are increasing *)
let rec increasing = function
  | [] → true
  | [_] → true
  | x :: y :: rest → x < y && increasing (y :: rest)
in increasing positions
```

26. Actors

26.1 Actor Model

Definition 26.1 (Actor)**.** An actor is a computational entity that:

1. Has a unique address
2. Has a mailbox (message queue)
3. Processes messages sequentially
4. Can send messages to other actors
5. Can create new actors
6. Can update its own behavior

Definition 26.2 (Actor Type)**.**

$$\mathsf{Actor}[M, S] = \{\mathsf{address} : \mathsf{ActorRef}[M], \mathsf{state} : S, \mathsf{behavior} : M \to S \to \mathsf{ActorComp}[S]\}$$

where M is the message type and S is the state type.

26.2 Actor Typing

Definition 26.3 (Actor Reference)**.** $\mathsf{ActorRef}[M]$ is a capability to send messages of type M:

$$\mathsf{ActorRef}[M] : \star$$

Definition 26.4 (Actor Computation)**.**

$$\begin{aligned}\mathsf{ActorComp}[S] &= \mathsf{Effect}\,[\mathsf{ActorEff}]\,S\\ \mathsf{ActorEff} &::= \mathsf{Send}[\mathsf{ActorRef}[M], M]\\ &\mid \mathsf{Spawn}[M', S', \mathsf{Behavior}[M', S']]\\ &\mid \mathsf{Self}[\mathsf{ActorRef}[M]]\end{aligned}$$

Definition 26.5 (Actor Typing Rules)**.**

$$\frac{\Gamma \vdash r : \mathsf{ActorRef}[M]\ [\varepsilon_1] \qquad \Gamma \vdash m : M\ [\varepsilon_2]}{\Gamma \vdash r!m : \mathsf{Unit}\ [\langle \mathsf{ActorEff} \mid \varepsilon_1 \sqcup \varepsilon_2\rangle]}\ \textsc{T-Send}$$

$$\frac{\Gamma \vdash \mathsf{init} : S\ [\varepsilon_1] \qquad \Gamma \vdash \mathsf{behav} : M \to S \to \mathsf{ActorComp}[S]\ [\varepsilon_2]}{\Gamma \vdash \mathsf{spawn}(\mathsf{init}, \mathsf{behav}) : \mathsf{ActorRef}[M]\ [\langle \mathsf{ActorEff} \mid \varepsilon_1 \sqcup \varepsilon_2\rangle]}\ \textsc{T-Spawn}$$

$$\frac{\text{inside actor handling } M}{\Gamma \vdash \mathsf{self} : \mathsf{ActorRef}[M]\ [\mathsf{ActorEff}]}\ \textsc{T-Self}$$

26.3 Actor Semantics

Definition 26.6 (Actor Configuration)**.** An actor configuration is:

$$\mathcal{C} = \{a_1 \mapsto (S_1, Q_1, B_1), \ldots, a_n \mapsto (S_n, Q_n, B_n)\}$$

where a_i is address, S_i is state, Q_i is message queue, B_i is behavior.

Definition 26.7 (Actor Reduction).

$$\mathcal{C}[a \mapsto (S, m :: Q, B)] \longrightarrow \mathcal{C}'[a \mapsto (S', Q, B')]$$
$$\text{where } (S', B', \mathsf{effects}) = B(m, S)$$

Effects include spawning new actors and sending messages.

26.4 F* Mechanization

Actors in F*

```
module BrrrSemantics.Actors

(* Actor address *)
type actor_id = nat

(* Actor reference (capability to send messages) *)
noeq type actor_ref (msg:Type) = {
  target : actor_id
}

(* Actor effect *)
noeq type actor_effect =
  | AESend  : #m:Type → actor_ref m → m → actor_effect
  | AESpawn : #m:Type → #s:Type → s → (m → s → actor_comp s) →
     actor_effect
  | AESelf  : actor_effect

and actor_comp (s:Type) =
  | ACPure    : s → actor_comp s
  | ACEffect  : actor_effect → (unit → actor_comp s) → actor_comp s

(* Behavior: message handler *)
type behavior (msg:Type) (state:Type) =
  msg → state → actor_comp state

(* Actor state *)
noeq type actor (msg:Type) (state:Type) = {
  id       : actor_id;
  state    : state;
  mailbox  : list msg;
  behavior : behavior msg state
}

(* Actor system configuration *)
noeq type actor_system = {
  actors     : list (actor_id & (∃ m s. actor m s));
  next_id    : actor_id;
  pending    : list (actor_id & (∃ m. m))  (* Pending messages *)
}

(* Send message *)
let send (#m:Type) (ref:actor_ref m) (msg:m) : actor_comp unit =
  ACEffect (AESend ref msg) (fun () → ACPure ())

(* Spawn new actor *)
let spawn (#m #s:Type) (init:s) (behav:behavior m s) : actor_comp (
    actor_ref m) =
  ACEffect (AESpawn init behav) (fun () →
    (* Would return new ref - simplified *)
    ACPure ({ target = 0 }))

(* Get self reference *)
let self (#m:Type) () : actor_comp (actor_ref m) =
  ACEffect AESelf (fun () → ACPure ({ target = 0 }))

(* Run one step of actor computation *)
let rec run_actor_comp (#s:Type) (comp:actor_comp s) (sys:actor_system
    )
    : s & actor_system =
  match comp with
  | ACPure s → (s, sys)
  | ACEffect eff cont →
      match eff with
      | AESend ref msg →
          let sys' = { sys with
            pending = (ref.target, msg) :: sys.pending
          } in
          run_actor_comp (cont ()) sys'
      | AESpawn init behav →
          let new_id = sys.next_id in
          let new_actor = { id = new_id; state = init; mailbox = [];
              behavior = behav } in
          let sys' = { sys with
            actors = (new_id, new_actor) :: sys.actors;
            next_id = new_id + 1
          } in
          run_actor_comp (cont ()) sys'
      | AESelf →
          run_actor_comp (cont ()) sys

(* Process one message for an actor *)
let step_actor (#m #s:Type) (a:actor m s) (sys:actor_system)
    : option (actor m s & actor_system) =
  match a.mailbox with
  | [] → None
  | msg :: rest →
      let (new_state, sys') = run_actor_comp (a.behavior msg a.state)
          sys in
      Some ({ a with state = new_state; mailbox = rest }, sys')

(* Deliver pending messages *)
let deliver_messages (sys:actor_system) : actor_system =
  List.fold_left (fun sys (target, msg) →
    (* Find actor and add to mailbox *)
    { sys with
      actors = List.map (fun (id, a) →
        if id = target then
          (* Add message to mailbox - simplified *)
          (id, a)
        else (id, a)
      ) sys.actors
    }
  ) { sys with pending = [] } sys.pending

(* Example: Ping-pong actors *)
type ping_msg = | Ping : actor_ref pong_msg → ping_msg
```

```
and pong_msg = | Pong : actor_ref ping_msg → pong_msg

let ping_behavior : behavior ping_msg nat =
  fun msg count →
    match msg with
    | Ping sender →
        let _ = send sender (Pong ({ target = 0 })) in
        ACPure (count + 1)

let pong_behavior : behavior pong_msg nat =
  fun msg count →
    match msg with
    | Pong sender →
        let _ = send sender (Ping ({ target = 0 })) in
        ACPure (count + 1)
```

Part VIII
Module System

27. Signatures

The module system is grounded in ML module theory, providing abstraction, separate compilation, and code reuse.

27.1 Module Language

Definition 27.1 (Module Grammar).

$$\begin{aligned} M ::=\ & \{B_1; \ldots; B_n\} && \text{(structure)} \\ \mid\ & M.x && \text{(projection)} \\ \mid\ & F(M) && \text{(functor application)} \\ \mid\ & M : \Sigma && \text{(transparent ascription)} \\ \mid\ & M :> \Sigma && \text{(opaque ascription/sealing)} \end{aligned}$$

Definition 27.2 (Signature Grammar).

$$\begin{aligned} \Sigma ::=\ & \{D_1; \ldots; D_n\} && \text{(signature)} \\ \mid\ & \Sigma_1 \to \Sigma_2 && \text{(functor signature)} \\ \mid\ & \Sigma\ \mathsf{where\ type}\ p = \tau && \text{(type refinement)} \end{aligned}$$

Definition 27.3 (Declaration Grammar).

$$\begin{aligned} D ::=\ & \mathsf{type}\ t && \text{(abstract type)} \\ \mid\ & \mathsf{type}\ t = \tau && \text{(manifest type)} \\ \mid\ & \mathsf{val}\ x : \tau && \text{(value specification)} \\ \mid\ & \mathsf{module}\ M : \Sigma && \text{(nested module)} \end{aligned}$$

27.2 Signature Typing

Definition 27.4 (Signature Typing Rules).

$$\frac{}{\Gamma \vdash \{\} : \mathsf{Sig}}\ \textsc{Sig-Empty} \qquad \frac{\Gamma \vdash \Sigma : \mathsf{Sig}}{\Gamma \vdash \{\mathsf{type}\ t; \Sigma\} : \mathsf{Sig}}\ \textsc{Sig-Type}$$

$$\frac{\Gamma \vdash \tau : \star \qquad \Gamma \vdash \Sigma : \mathsf{Sig}}{\Gamma \vdash \{\mathsf{val}\ x : \tau; \Sigma\} : \mathsf{Sig}}\ \textsc{Sig-Val} \qquad \frac{\Gamma \vdash \Sigma_1 : \mathsf{Sig} \qquad \Gamma \vdash \Sigma_2 : \mathsf{Sig}}{\Gamma \vdash \{\mathsf{module}\ M : \Sigma_1; \Sigma_2\} : \mathsf{Sig}}\ \textsc{Sig-Module}$$

27.3 Signature Matching

Definition 27.5 (Signature Subtyping). $\Sigma_1 <: \Sigma_2$ (signature Σ_1 matches Σ_2) iff:

- For each $\mathsf{type}\ t$ in Σ_2, either:
  - Σ_1 has $\mathsf{type}\ t = \tau$ (more specific)
  - Σ_1 has $\mathsf{type}\ t$ (same abstraction)
- For each $\mathsf{val}\ x : \tau$ in Σ_2, Σ_1 has $\mathsf{val}\ x : \tau'$ with $\tau' <: \tau$
- Additional declarations in Σ_1 are allowed

Definition 27.6 (Signature Matching Rules).

$$\frac{}{\Sigma <: \{\}}\ \textsc{Match-Empty} \qquad \frac{\Sigma <: \Sigma' \qquad \mathsf{type}\ t \in \Sigma \lor \mathsf{type}\ t = \_ \in \Sigma}{\Sigma <: \{\mathsf{type}\ t; \Sigma'\}}\ \textsc{Match-Type-Abs}$$

$$\frac{\Sigma <: \Sigma' \qquad \mathsf{type}\ t = \tau \in \Sigma}{\Sigma <: \{\mathsf{type}\ t = \tau; \Sigma'\}}\ \textsc{Match-Type-Manifest}$$

$$\frac{\Sigma <: \Sigma' \qquad \mathsf{val}\ x : \tau' \in \Sigma \qquad \tau' <: \tau}{\Sigma <: \{\mathsf{val}\ x : \tau; \Sigma'\}}\ \textsc{Match-Val}$$

27.4 F* Mechanization

Signatures in F*

```
module BrrrSemantics.Signatures

(* Signature declaration *)
noeq type sig_decl =
  | SigTypeAbstract : string → sig_decl                (* type t *)
  | SigTypeManifest : string → brrr_type → sig_decl (* type t = τ *)
  | SigVal          : string → brrr_type → sig_decl (* val x : τ *)
  | SigModule       : string → signature → sig_decl (* module M : Σ *)

(* Signature *)
and signature =
  | SigEmpty   : signature
  | SigCons    : sig_decl → signature → signature
  | SigFunctor: string → signature → signature → signature   (* (X : Σ1
    ) → Σ2 *)
  | SigWhere   : signature → string → brrr_type → signature   (* Σ where
      type p = τ *)

(* Module expression *)
noeq type mod_expr =
  | ModStruct     : list mod_binding → mod_expr
  | ModProj       : mod_expr → string → mod_expr
  | ModApp        : mod_expr → mod_expr → mod_expr
  | ModAscribe    : mod_expr → signature → mod_expr     (* M : Σ *)
  | ModSeal       : mod_expr → signature → mod_expr     (* M :> Σ *)
  | ModFunctor    : string → signature → mod_expr → mod_expr
  | ModIdent      : string → mod_expr

(* Module binding *)
and mod_binding =
  | BindType    : string → brrr_type → mod_binding
  | BindVal     : string → expr → mod_binding
  | BindModule : string → mod_expr → mod_binding

(* Lookup declaration in signature *)
let rec lookup_sig_decl (name:string) (sig_:signature) : option
    sig_decl =
  match sig_ with
  | SigEmpty → None
  | SigCons decl rest →
      let decl_name = match decl with
        | SigTypeAbstract n → n
        | SigTypeManifest n _ → n
```

```
          | SigVal n _ → n
          | SigModule n _ → n
        in
        if decl_name = name then Some decl
        else lookup_sig_decl name rest
    | SigFunctor _ _ _ → None
    | SigWhere base _ _ → lookup_sig_decl name base

(* Check signature subtyping *)
let rec sig_subtype (s1 s2:signature) : bool =
  match s2 with
    | SigEmpty → true
    | SigCons decl rest →
        (match decl with
         | SigTypeAbstract t →
             (* s1 must have type t (abstract or manifest) *)
             (match lookup_sig_decl t s1 with
              | Some (SigTypeAbstract _) → sig_subtype s1 rest
              | Some (SigTypeManifest _ _) → sig_subtype s1 rest
              | _ → false)
         | SigTypeManifest t τ →
             (* s1 must have type t = τ *)
             (match lookup_sig_decl t s1 with
              | Some (SigTypeManifest _ τ') → τ = τ' && sig_subtype s1
                 rest
              | _ → false)
         | SigVal x τ →
             (* s1 must have val x with subtype *)
             (match lookup_sig_decl x s1 with
              | Some (SigVal _ τ') → subtype τ' τ && sig_subtype s1 rest
              | _ → false)
         | SigModule m sig_m →
             (match lookup_sig_decl m s1 with
              | Some (SigModule _ sig_m') → sig_subtype sig_m' sig_m &&
                 sig_subtype s1 rest
              | _ → false))
    | SigFunctor x s_arg s_res →
        (match s1 with
         | SigFunctor _ s_arg' s_res' →
             sig_subtype s_arg s_arg' && sig_subtype s_res' s_res   (*
                Contravariant in arg *)
         | _ → false)
    | SigWhere base path τ →
        (* Check base and verify type equality *)
        sig_subtype s1 base

(* Ascription: check module against signature *)
let check_ascription (m:mod_expr) (sig_:signature) : bool =
   (* Would need full module type checking *)
   admit()
```

28. Functors

28.1 Functor Definition

Definition 28.1 (Functor)**.** A functor is a module parameterized by another module:

$$\mathsf{functor}(X : \Sigma_1) \Rightarrow M : \Sigma_1 \to \Sigma_2$$

Definition 28.2 (Functor Typing)**.**

$$\frac{\Gamma, X : \Sigma_1 \vdash M : \Sigma_2}{\Gamma \vdash \mathsf{functor}(X : \Sigma_1) \Rightarrow M : \Sigma_1 \to \Sigma_2}\ \textsc{T-Functor}$$

$$\frac{\Gamma \vdash F : \Sigma_1 \to \Sigma_2 \qquad \Gamma \vdash M : \Sigma_1}{\Gamma \vdash F(M) : \Sigma_2[X := M]}\ \textsc{T-FunctorApp}$$

28.2 Applicative vs Generative

Definition 28.3 (Applicative Functor)**.** Applicative semantics: $F(M) = F(M')$ when $M \equiv M'$.

- Types are shared across applications
- $F(M).t = F(M).t$ (type equality preserved)

Definition 28.4 (Generative Functor)**.** Generative semantics: $F(M) \neq F(M')$ even when $M \equiv M'$.

- Fresh abstract types on each application
- Used for stateful modules, generativity

Definition 28.5 (Functor Mode)**.**

$$\mathsf{functor}^{\mathsf{app}}(X : \Sigma) \Rightarrow M \quad \text{(applicative, default)}$$

$$\mathsf{functor}^{\mathsf{gen}}(X : \Sigma) \Rightarrow M \quad \text{(generative)}$$

28.3 Higher-Order Functors

Definition 28.6 (Functor as First-Class Module)**.** Functors can take functors as arguments:

$$(\Sigma_1 \to \Sigma_2) \to \Sigma_3$$

Example 28.7 (Functor Composition)**.**

$$\begin{aligned}&\mathsf{module\ Compose} = \\ &\quad \mathsf{functor}(F : \Sigma_1 \to \Sigma_2)(G : \Sigma_2 \to \Sigma_3)(X : \Sigma_1) \Rightarrow G(F(X))\end{aligned}$$

28.4 F* Mechanization

Functors in F*

```
module BrrrSemantics.Functors

open BrrrSemantics.Signatures

(* Functor mode *)
type functor_mode = | Applicative | Generative

(* Functor type *)
noeq type functor_type = {
  param_name : string;
  param_sig  : signature;
  result_sig : signature;
  mode       : functor_mode
}

(* Functor value *)
noeq type functor_val = {
  ftype : functor_type;
  body  : mod_expr
}

(* Type stamp for generative functors *)
type type_stamp = nat

(* Stamped type (for generativity) *)
noeq type stamped_type =
  | Unstamped : brrr_type → stamped_type
  | Stamped   : type_stamp → string → stamped_type

(* Global stamp counter *)
let stamp_counter : ref type_stamp = ref 0

let fresh_stamp () : type_stamp =
  let s = !stamp_counter in
  stamp_counter := s + 1;
  s

(* Apply functor *)
let apply_functor (f:functor_val) (arg:mod_expr) : mod_expr =
  (* Substitute parameter in body *)
  let body' = subst_mod f.ftype.param_name arg f.body in
  match f.ftype.mode with
  | Applicative → body'
  | Generative →
      (* Stamp all abstract types with fresh stamps *)
      stamp_abstract_types (fresh_stamp ()) body'

and subst_mod (name:string) (replacement:mod_expr) (m:mod_expr) :
   mod_expr =
  match m with
  | ModIdent n → if n = name then replacement else m
  | ModStruct bindings →
      ModStruct (List.map (subst_binding name replacement) bindings)
  | ModProj m' field → ModProj (subst_mod name replacement m') field
  | ModApp f arg →
      ModApp (subst_mod name replacement f) (subst_mod name
         replacement arg)
  | ModFunctor param sig_ body →
      if param = name then m  (* Shadowed *)
      else ModFunctor param sig_ (subst_mod name replacement body)
  | _ → m

and subst_binding (name:string) (replacement:mod_expr) (b:mod_binding)
    : mod_binding =
  match b with
  | BindModule n m → BindModule n (subst_mod name replacement m)
  | _ → b

and stamp_abstract_types (stamp:type_stamp) (m:mod_expr) : mod_expr =
  (* Would traverse and stamp abstract types *)
  m

(* Check applicativity: result types equal for equal arguments *)
let applicative_property (f:functor_val) (m1 m2:mod_expr) : bool =
  if f.ftype.mode = Generative then true  (* Not required *)
  else
```

```
    (* If m1 ≡ m2, then F(m1).t = F(m2).t for all types t *)
    admit()

(* Example: Set functor *)
let set_functor_sig : functor_type = {
  param_name = "Elem";
  param_sig = SigCons (SigTypeAbstract "t")
              (SigCons (SigVal "compare" (TFunc (TVar "t") (TFunc (
                 TVar "t") TInt) Pure))
               SigEmpty);
  result_sig = SigCons (SigTypeAbstract "set")
               (SigCons (SigVal "empty" (TVar "set"))
               (SigCons (SigVal "add" (TFunc (TVar "Elem.t") (TFunc (
                  TVar "set") (TVar "set")) Pure))
               (SigCons (SigVal "member" (TFunc (TVar "Elem.t") (TFunc
                   (TVar "set") TBool) Pure))
                SigEmpty)));
  mode = Applicative
}

(* Higher-order functor: takes functor as argument *)
noeq type ho_functor = {
  param_functor : functor_type;
  result : signature
}
```

29. Sealing and Abstraction

29.1 Opaque Ascription

Definition 29.1 (Sealing)**.** Opaque ascription $M :> \Sigma$ hides implementation details:

- Abstract types in Σ become opaque
- Type equalities not in Σ are forgotten
- Creates an abstraction barrier

Definition 29.2 (Sealing Typing)**.**

$$\frac{\Gamma \vdash M : \Sigma' \qquad \Sigma' <: \Sigma}{\Gamma \vdash M :> \Sigma : \Sigma} \text{ T-SEAL}$$

The result type is exactly Σ, not a subtype.

Theorem 29.3 (Abstraction Theorem)**.** *If $M :> \Sigma$ and* type t *is abstract in Σ, then no client can distinguish the implementation type of t from any other implementation satisfying Σ.*

29.2 Existential Types

Definition 29.4 (Module as Existential)**.** A sealed module $M :> \{\mathsf{type}\ t; \mathsf{val}\ x : t\}$ is equivalent to:

$$\exists t.\, t \times t \cong \mathsf{pack}\,(\tau, v)\ \mathsf{as}\ \exists t.\, t$$

where τ is the hidden implementation type.

Definition 29.5 (Pack and Unpack)**.**

$$\frac{\Gamma \vdash v : \tau[\alpha := \sigma]}{\Gamma \vdash \mathsf{pack}\,(\sigma, v) : \exists \alpha.\, \tau} \text{ T-PACK}$$

$$\frac{\Gamma \vdash e_1 : \exists \alpha.\, \tau \qquad \Gamma, \alpha, x : \tau \vdash e_2 : \sigma \qquad \alpha \notin \mathsf{ftv}(\sigma)}{\Gamma \vdash \mathsf{unpack}\,(\alpha, x) = e_1\ \mathsf{in}\ e_2 : \sigma} \text{ T-UNPACK}$$

29.3 F* Mechanization

Sealing in F*

```
module BrrrSemantics.Sealing

open BrrrSemantics.Signatures
open BrrrSemantics.Functors

(* Sealed module: hides type implementations *)
noeq type sealed_module = {
  exposed_sig : signature;
  impl        : mod_expr;
  hidden_types: list (string & brrr_type)  (* Abstract type
      implementations *)
}

(* Seal a module *)
let seal (m:mod_expr) (sig_:signature) : option sealed_module =
  (* Check m matches sig_ *)
  if check_ascription m sig_ then
    (* Extract abstract type implementations *)
    let hidden = collect_hidden_types m sig_ in
    Some { exposed_sig = sig_; impl = m; hidden_types = hidden }
  else None

and collect_hidden_types (m:mod_expr) (sig_:signature) : list (string
   & brrr_type) =
  (* Find types that are manifest in m but abstract in sig_ *)
  match sig_ with
  | SigEmpty → []
  | SigCons (SigTypeAbstract t) rest →
      (match get_type_impl m t with
       | Some τ → (t, τ) :: collect_hidden_types m rest
       | None → collect_hidden_types m rest)
  | SigCons _ rest → collect_hidden_types m rest
  | _ → []

and get_type_impl (m:mod_expr) (t:string) : option brrr_type =
  match m with
  | ModStruct bindings →
      List.find_map (fun b →
        match b with
        | BindType n τ → if n = t then Some τ else None
        | _ → None) bindings
  | _ → None

(* Existential type *)
noeq type existential (f:Type → Type) =
  | Pack : #a:Type → a → f a → existential f

(* Module as existential: { type t; val x : t } ≅ ∃t. t *)
type abstract_module (sig_:Type → Type) = existential sig_

(* Pack: hide implementation *)
let pack_module (#a:Type) (#f:Type → Type) (impl:a) (witness:f a) :
    existential f =
  Pack impl witness
```

```
(* Unpack: use abstraction *)
let unpack_module (#f:Type → Type) (#r:Type)
                  (e:existential f) (body:(#a:Type → a → f a → r)) : r
                       =
  match e with
  | Pack impl witness → body impl witness

(* Abstraction theorem: parametricity *)
(* If client only knows abstract interface, behavior is same
   regardless of implementation *)

(* Two implementations of "abstract counter" *)
type counter_sig (t:Type) = {
  zero : t;
  inc  : t → t;
  get  : t → int
}

(* Implementation 1: using int *)
let counter_int : counter_sig int = {
  zero = 0;
  inc = fun n → n + 1;
  get = fun n → n
}

(* Implementation 2: using list *)
let counter_list : counter_sig (list unit) = {
  zero = [];
  inc = fun l → () :: l;
  get = fun l → List.length l
}

(* Sealed: client can't distinguish *)
let sealed_counter_1 : existential counter_sig = Pack 0 counter_int
let sealed_counter_2 : existential counter_sig = Pack [] counter_list

(* Client code works with either *)
let use_counter (c:existential counter_sig) : int =
  unpack_module c (fun zero ops →
    ops.get (ops.inc (ops.inc ops.zero)))
(* Returns 2 for both implementations *)
```

30. Mixin Composition

30.1 Include

Definition 30.1 (Module Include). include M incorporates all bindings from M:

$$\begin{aligned}&\textsf{module Extended} = \{\\&\quad \textsf{include Base;}\\&\quad \textsf{val extra} = \ldots\\&\}\end{aligned}$$

Definition 30.2 (Include Typing).

$$\frac{\Gamma \vdash M : \{D_1; \ldots; D_n\}}{\Gamma \vdash \{\textsf{include } M; B\} : \{D_1; \ldots; D_n; \textsf{typeof}(B)\}}\ \textsc{T-Include}$$

30.2 Linear Composition

Definition 30.3 (Module Composition)**.** $M_1 + M_2$ combines two modules with disjoint signatures:

$$\frac{\Gamma \vdash M_1 : \Sigma_1 \qquad \Gamma \vdash M_2 : \Sigma_2 \qquad \Sigma_1 \cap \Sigma_2 = \emptyset}{\Gamma \vdash M_1 + M_2 : \Sigma_1 + \Sigma_2}\ \textsc{T-Compose}$$

30.3 Override

Definition 30.4 (Module Override)**.** M with $\{\mathsf{overrides}\}$ refines module components:

$$\begin{aligned}&\mathsf{module\ Refined} = \mathsf{Base\ with}\ \{\\ &\quad \mathsf{type}\ t = \mathsf{SpecificType};\\ &\quad \mathsf{val}\ x = \mathsf{new\_impl}\\ &\}\end{aligned}$$

30.4 Conflict Resolution

Definition 30.5 (Diamond Problem Resolution)**.** When composing modules with overlapping signatures:

$$M_1 + M_2\ \mathsf{with}\ \{\mathsf{val}\ x = M_1.x\}$$

explicitly selects which binding to use.

30.5 F* Mechanization

Mixin Composition in F*

```
module BrrrSemantics.Mixins

open BrrrSemantics.Signatures

(* Get all binding names from module *)
let rec binding_names (m:mod_expr) : list string =
  match m with
  | ModStruct bindings →
      List.map (fun b →
        match b with
        | BindType n _ → n
        | BindVal n _ → n
        | BindModule n _ → n) bindings
  | _ → []

(* Check disjointness *)
let disjoint (m1 m2:mod_expr) : bool =
  let names1 = binding_names m1 in
  let names2 = binding_names m2 in
  not (List.∃ (fun n → List.mem n names2) names1)

(* Include: flatten module *)
let include_module (base:mod_expr) (additions:list mod_binding) :
    mod_expr =
  match base with
  | ModStruct base_bindings →
      ModStruct (base_bindings @ additions)
  | _ → ModStruct additions  (* Base not a struct, just use additions
      *)
```

```
(* Linear composition *)
let compose_modules (m1 m2:mod_expr) : option mod_expr =
  if disjoint m1 m2 then
    match m1, m2 with
    | ModStruct b1, ModStruct b2 → Some (ModStruct (b1 @ b2))
    | ModStruct b1, _ → Some (ModStruct (b1 @ [BindModule "_m2" m2]))
    | _, ModStruct b2 → Some (ModStruct ([BindModule "_m1" m1] @ b2))
    | _, _ → Some (ModStruct [BindModule "_m1" m1; BindModule "_m2" m2
      ])
  else None  (* Conflict *)

(* Override binding *)
noeq type override =
  | OverrideType : string → brrr_type → override
  | OverrideVal  : string → expr → override

(* Apply overrides to module *)
let apply_overrides (m:mod_expr) (overrides:list override) : mod_expr
   =
  match m with
  | ModStruct bindings →
      let bindings' = List.map (fun b →
        let name = match b with
          | BindType n _ → n
          | BindVal n _ → n
          | BindModule n _ → n
        in
        match List.find (fun o →
          match o with
          | OverrideType n _ → n = name
          | OverrideVal n _ → n = name) overrides with
        | Some (OverrideType _ τ) → BindType name τ
        | Some (OverrideVal _ e) → BindVal name e
        | None → b) bindings in
      ModStruct bindings'
  | _ → m

(* Mixin: module with deferred bindings *)
noeq type mixin = {
  provided  : list mod_binding;      (* What mixin provides *)
  required  : list sig_decl          (* What mixin needs *)
}

(* Complete mixin with implementation *)
let complete_mixin (mx:mixin) (impl:mod_expr) : option mod_expr =
  (* Check impl provides required bindings *)
  let impl_names = binding_names impl in
  let required_names = List.map (fun d →
    match d with
    | SigTypeAbstract n → n
    | SigTypeManifest n _ → n
    | SigVal n _ → n
    | SigModule n _ → n) mx.required in
  if List.for_all (fun n → List.mem n impl_names) required_names then
    compose_modules impl (ModStruct mx.provided)
  else None
```

```
(* Example: Ordered mixin *)
let ordered_mixin : mixin = {
  provided = [
    BindVal "min" (ELam "a" TUnit (ELam "b" TUnit
      (EIf (EApp (EApp (EVar "compare") (EVar "a")) (EVar "b"))
           (EVar "a") (EVar "b"))))
  ];
  required = [
    SigTypeAbstract "t";
    SigVal "compare" (TFunc (TVar "t") (TFunc (TVar "t") TBool) Pure)
  ]
}
```

Part IX
Security & Verification

31. Information Flow Types

Information flow types prevent sensitive data from leaking to unauthorized observers.

31.1 Security Lattice

Definition 31.1 (Security Lattice)**.** A security lattice $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ where:

- L is a set of security levels
- $\sqsubseteq$ is the "flows to" partial order
- $\sqcup$ is least upper bound (join)
- $\sqcap$ is greatest lower bound (meet)
- $\bot$ is public (lowest)
- $\top$ is secret (highest)

Example 31.2 (Two-Point Lattice)**.**

$$L_2 = \{\mathsf{Low}, \mathsf{High}\} \quad \text{with } \mathsf{Low} \sqsubseteq \mathsf{High}$$

Example 31.3 (Four-Point Lattice)**.**

$$\begin{tikzcd}
& \mathsf{TopSecret} & \\
\mathsf{Secret} \arrow[ur] & & \mathsf{Confidential} \arrow[ul] \\
& \mathsf{Public} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

31.2 Security-Typed Values

Definition 31.4 (Labeled Type)**.** A type with security label:

$$\tau^{\ell} \quad \text{(type } \tau \text{ at security level } \ell\text{)}$$

Definition 31.5 (Subtyping with Labels)**.**

$$\tau^{\ell_1} <: \tau^{\ell_2} \iff \tau <: \tau \wedge \ell_1 \sqsubseteq \ell_2$$

A Low value can flow to High, but not vice versa.

31.3 Security Typing Rules

Definition 31.6 (Security Typing Judgment)**.** $\Gamma; \mathsf{pc} \vdash e : \tau^{\ell}$ means:

> Under context Γ with program counter label pc, expression e has type τ at level ℓ.

Definition 31.7 (Core Security Typing)**.**

$$\frac{(x : \tau^{\ell}) \in \Gamma}{\Gamma; \mathsf{pc} \vdash x : \tau^{\ell}}\ \textsc{Sec-Var} \qquad \frac{\Gamma; \mathsf{pc} \vdash e : \tau^{\ell_1} \qquad \ell_1 \sqsubseteq \ell_2}{\Gamma; \mathsf{pc} \vdash e : \tau^{\ell_2}}\ \textsc{Sec-Sub}$$

$$\frac{\Gamma; \mathsf{pc} \vdash e_1 : (\tau_1^{\ell_1} \to \tau_2^{\ell_2})^{\ell_f} \qquad \Gamma; \mathsf{pc} \vdash e_2 : \tau_1^{\ell_1}}{\Gamma; \mathsf{pc} \vdash e_1\, e_2 : \tau_2^{\ell_2 \sqcup \ell_f}}\ \textsc{Sec-App}$$

$$\frac{\Gamma; \mathsf{pc} \vdash e_c : \mathsf{Bool}^{\ell_c} \qquad \Gamma; \mathsf{pc} \sqcup \ell_c \vdash e_1 : \tau^{\ell} \qquad \Gamma; \mathsf{pc} \sqcup \ell_c \vdash e_2 : \tau^{\ell}}{\Gamma; \mathsf{pc} \vdash \mathsf{if}\ e_c\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau^{\ell \sqcup \ell_c}}\ \textsc{Sec-If}$$

$$\frac{\Gamma; \mathsf{pc} \vdash e : \tau^{\ell_e} \qquad (x : \mathsf{ref}\,\tau^{\ell_x}) \in \Gamma \qquad \mathsf{pc} \sqcup \ell_e \sqsubseteq \ell_x}{\Gamma; \mathsf{pc} \vdash x := e : \mathsf{Unit}^{\mathsf{pc}}}\ \textsc{Sec-Assign}$$

31.4 Implicit Flows

Definition 31.8 (Implicit Flow)**.** Information flows through control structure:

```
if secret then public := 1 else public := 0
```

The value of `public` reveals `secret`.

The `pc` (program counter) label tracks implicit flows. In Sec-If, the branches are checked under $\mathsf{pc} \sqcup \ell_c$, ensuring any assignment respects the guard's secrecy.

31.5 F* Mechanization

Information Flow Types in F*

```
module BrrrSemantics.InformationFlow

(* Security level *)
type sec_level =
  | Public : sec_level       (* ⊥ - anyone can read *)
  | Secret : sec_level       (* ⊤ - restricted *)

(* Security lattice operations *)
let sec_leq (l1 l2:sec_level) : bool =
  match l1, l2 with
  | Public, _ → true
  | Secret, Secret → true
  | Secret, Public → false

let sec_join (l1 l2:sec_level) : sec_level =
  match l1, l2 with
  | Secret, _ → Secret
  | _, Secret → Secret
  | Public, Public → Public

let sec_meet (l1 l2:sec_level) : sec_level =
  match l1, l2 with
  | Public, _ → Public
  | _, Public → Public
  | Secret, Secret → Secret

(* Labeled type *)
noeq type labeled_type = {
  base_type : brrr_type;
  label     : sec_level
}

(* Security typing context *)
type sec_ctx = list (string & labeled_type)

(* Program counter label *)
type pc_label = sec_level

(* Security type checking *)
let rec sec_typecheck (ctx:sec_ctx) (pc:pc_label) (e:expr)
    : option labeled_type =
  match e with
  | EVar x →
      List.assoc x ctx
  | ELit v →
      Some { base_type = type_of_val v; label = Public }

  | EApp f arg →
      (match sec_typecheck ctx pc f, sec_typecheck ctx pc arg with
       | Some {base_type = TFunc a r _; label = lf},
         Some {base_type = ta; label = la} →
           if subtype ta a && sec_leq la (get_arg_label a) then
             Some { base_type = r; label = sec_join lf (
                 get_result_label r) }
           else None
       | _, _ → None)

  | EIf cond e1 e2 →
      (match sec_typecheck ctx pc cond with
       | Some {base_type = TBool; label = lc} →
           let pc' = sec_join pc lc in
           (match sec_typecheck ctx pc' e1, sec_typecheck ctx pc' e2
              with
            | Some t1, Some t2 →
                if t1.base_type = t2.base_type then
                  Some { base_type = t1.base_type;
                         label = sec_join lc (sec_join t1.label t2.
                             label) }
                else None
            | _, _ → None)
       | _ → None)

  | ELet x e1 e2 →
      (match sec_typecheck ctx pc e1 with
       | Some t1 →
           sec_typecheck ((x, t1) :: ctx) pc e2
       | None → None)

  | _ → None

and get_arg_label (t:brrr_type) : sec_level = Public  (* Simplified *)
and get_result_label (t:brrr_type) : sec_level = Public
and type_of_val (v:base_val) : brrr_type =
  match v with
  | VBool _ → TBool
  | VInt _ → TInt {width=I64; sign=Signed}
  | _ → TUnit

(* Check assignment respects information flow *)
let check_assignment (ctx:sec_ctx) (pc:pc_label) (x:string) (e:expr) :
    bool =
  match List.assoc x ctx, sec_typecheck ctx pc e with
  | Some {label = lx}, Some {label = le} →
      sec_leq (sec_join pc le) lx  (* pc ⊔ le ⊑ lx *)
  | _, _ → false

(* Example: This should fail *)
let implicit_flow_example () : bool =
  let ctx = [("secret", {base_type = TBool; label = Secret});
             ("public", {base_type = TInt {width=I32; sign=Signed};
                 label = Public})] in
```

```
let pc = Public in
(* if secret then public := 1 else public := 0 *)
(* The assignment happens under pc = Public ⊔ Secret = Secret *)
(* But public has label Public, so Secret ⊑ Public fails *)
check_assignment ctx Secret "public" (ELit (VInt 1))
(* Returns false - correctly rejected! *)
```

32. Noninterference

32.1 Semantic Security

Definition 32.1 (Low-Equivalence)**.** Two states $\sigma_1 \approx_L \sigma_2$ are low-equivalent iff they agree on all Low-labeled values:

$$\sigma_1 \approx_L \sigma_2 \iff \forall x : \tau^{\mathsf{Low}}.\, \sigma_1(x) = \sigma_2(x)$$

Definition 32.2 (Noninterference)**.** A program P satisfies noninterference iff:

$$\sigma_1 \approx_L \sigma_2 \implies P(\sigma_1) \approx_L P(\sigma_2)$$

Low inputs determine low outputs—high inputs cannot influence low outputs.

Theorem 32.3 (Type Soundness Implies Noninterference)**.** *If $\Gamma; \mathsf{Low} \vdash P : \tau^{\ell}$ (well-typed with $\mathsf{pc} = \mathsf{Low}$), then P satisfies noninterference.*

Proof Sketch. By induction on typing derivations. The key insight is that:

1. High data can only flow to High locations
2. Branches on High data (raising pc) can only assign to High locations
3. Therefore, Low outputs depend only on Low inputs

□

32.2 Termination-Insensitive vs Termination-Sensitive

Definition 32.4 (Termination Channel)**.** Programs can leak information through termination:

```
if secret then loop_forever() else ()
```

Whether the program terminates reveals `secret`.

Definition 32.5 (TINI vs TSNI)**.** • **TINI** (Termination-Insensitive NI): Only considers terminating executions

• **TSNI** (Termination-Sensitive NI): Also considers termination behavior

32.3 Declassification

Definition 32.6 (Declassification)**.** Controlled release of secret information:

$$\mathsf{declassify}(e, \ell_{\mathsf{from}}, \ell_{\mathsf{to}}) : \tau^{\ell_{\mathsf{to}}}$$

where $\ell_{\mathsf{from}} \not\sqsubseteq \ell_{\mathsf{to}}$ (intentional downgrade).

Definition 32.7 (Robust Declassification)**.** Declassification must be:

- **Intentional**: Explicitly marked in code
- **Bounded**: Limited by policy (who, what, when, where)
- **Auditable**: Logged for review

32.4 F* Mechanization

Noninterference in F*

```
module BrrrSemantics.Noninterference

open BrrrSemantics.InformationFlow

(* State: mapping from variables to values with labels *)
type sec_state = list (string & (value & sec_level))

(* Low-equivalence: states agree on Low values *)
let low_equiv (s1 s2:sec_state) : bool =
  List.for_all (fun (x, (v1, l1)) →
    if l1 = Public then
      match List.assoc x s2 with
      | Some (v2, l2) → l2 = Public && v1 = v2
      | None → false
    else true) s1

(* Program execution *)
type exec_result =
  | Terminates : sec_state → exec_result
  | Diverges   : exec_result

(* Noninterference property *)
let noninterference (p:expr) : bool =
  (* For all s1, s2: s1 ≈_L s2⟹p(s1) ≈_L p(s2) *)
  (* This is a semantic property - would need to enumerate states *)
  admit()

(* Termination-insensitive noninterference *)
let tini (p:expr) (s1 s2:sec_state) : bool =
  if low_equiv s1 s2 then
    match run p s1, run p s2 with
    | Terminates s1', Terminates s2' → low_equiv s1' s2'
    | Diverges, _ → true  (* Don't care if one diverges *)
    | _, Diverges → true
  else true

and run (e:expr) (s:sec_state) : exec_result =
  (* Would need full interpreter *)
  Terminates s

(* Declassification policy *)
noeq type declass_policy = {
  allowed_from : sec_level;
  allowed_to   : sec_level;
  principal    : string;      (* Who can declassify *)
  purpose      : string       (* Why *)
}

(* Declassification with policy check *)
let declassify (#a:Type) (v:a) (from_:sec_level) (to_:sec_level)
               (policy:declass_policy) : option (a & sec_level) =
  if from_ = policy.allowed_from && to_ = policy.allowed_to then
    (* Log declassification for audit *)
    Some (v, to_)
  else None

(* Robust declassification: attacker can't influence what gets
    declassified *)
let robust_declassify (#a:Type) (v:a) (guard:bool) (policy:
    declass_policy)
    : option (a & sec_level) =
  (* Guard must be Low (attacker-controlled) *)
  (* Value must be computed independently of attacker *)
  if guard then
    declassify v Secret Public policy
  else None

(* Endorsement: dual of declassification for integrity *)
let endorse (#a:Type) (v:a) (from_:sec_level) (to_:sec_level) : a &
    sec_level =
  (* Raise integrity level (trust untrusted input after validation) *)
  (v, to_)

(* Example: Password check with declassification *)
let check_password (input:string) (stored_hash:string) : bool =
  let input_hash = hash input in  (* input_hash is Low *)
  let result = input_hash = stored_hash in  (* result is High (depends
      on stored) *)
  (* Declassify the boolean result (not the password!) *)
  match declassify result Secret Public
    { allowed_from = Secret; allowed_to = Public;
      principal = "auth_system"; purpose = "login_result" } with
  | Some (r, _) → r
  | None → false

and hash (s:string) : string = s  (* Simplified *)
```

33. Taint Analysis

33.1 Taint Sources and Sinks

Definition 33.1 (Taint Source)**.** A source introduces tainted data:

$$\mathsf{source}_{\mathsf{kind}} : () \to \tau^{\mathsf{Tainted[kind]}}$$

Examples: user input, network data, file contents.

Definition 33.2 (Taint Sink)**.** A sink is a security-sensitive operation:

$$\mathsf{sink}_{\mathsf{kind}} : \tau^{\mathsf{Untainted}} \to \sigma$$

Examples: SQL query, shell command, HTML output.

Definition 33.3 (Sanitizer)**.** A sanitizer removes taint:

$$\mathsf{sanitize}_{\mathsf{kind}} : \tau^{\mathsf{Tainted[kind]}} \to \tau^{\mathsf{Untainted}}$$

Examples: escaping, validation, encoding.

33.2 Taint Propagation

Definition 33.4 (Taint Propagation Rules).

$$\frac{\Gamma \vdash e_1 : \tau_1^{t_1} \qquad \Gamma \vdash e_2 : \tau_2^{t_2}}{\Gamma \vdash e_1 \oplus e_2 : \tau^{t_1 \sqcup t_2}} \text{ TAINT-OP}$$

$$\frac{\Gamma \vdash f : (\tau_1, \ldots, \tau_n) \to \sigma \qquad \exists i.\, e_i : \tau_i^{\mathsf{Tainted}}}{\Gamma \vdash f(e_1, \ldots, e_n) : \sigma^{\mathsf{Tainted}}} \text{ TAINT-CALL}$$

$$\frac{\Gamma \vdash e : \tau^{\mathsf{Tainted[kind]}} \qquad \mathsf{sink[kind]}}{\text{ERROR: tainted data reaches sink}} \text{ TAINT-SINK}$$

33.3 Taint Kinds

Definition 33.5 (Taint Taxonomy). Different vulnerability classes:

SQLi	(SQL injection)
XSS	(cross-site scripting)
CMDi	(command injection)
PathTraversal	(path traversal)
SSRF	(server-side request forgery)

33.4 F* Mechanization

Taint Analysis in F*

```
module BrrrSemantics.TaintAnalysis

(* Taint kind *)
type taint_kind =
  | TaintSQLi          : taint_kind
  | TaintXSS           : taint_kind
  | TaintCMDi          : taint_kind
  | TaintPathTraversal: taint_kind
  | TaintSSRF          : taint_kind

(* Taint status *)
type taint_status =
  | Untainted : taint_status
  | Tainted   : list taint_kind → taint_status

(* Join taint statuses *)
let taint_join (t1 t2:taint_status) : taint_status =
  match t1, t2 with
  | Untainted, t → t
  | t, Untainted → t
  | Tainted k1, Tainted k2 → Tainted (List.dedup (k1 @ k2))

(* Tainted value *)
noeq type tainted (a:Type) = {
  value : a;
  taint : taint_status
}

(* Source: mark data as tainted *)
let source (#a:Type) (kind:taint_kind) (v:a) : tainted a =
  { value = v; taint = Tainted [kind] }

(* Sink check: ensure untainted *)
let sink (#a:Type) (kind:taint_kind) (t:tainted a) : option a =
  match t.taint with
  | Untainted → Some t.value
  | Tainted kinds →
      if List.mem kind kinds then None  (* Vulnerability! *)
      else Some t.value  (* Different taint kind, may be ok *)

(* Sanitizer: remove specific taint *)
let sanitize (#a:Type) (kind:taint_kind) (t:tainted a) (sanitizer:a →
   a)
    : tainted a =
  let cleaned = sanitizer t.value in
  match t.taint with
  | Untainted → { value = cleaned; taint = Untainted }
  | Tainted kinds →
      let remaining = List.filter (fun k → k ≠ kind) kinds in
      { value = cleaned;
        taint = if remaining = [] then Untainted else Tainted
           remaining }

(* Propagate taint through operations *)
let taint_map (#a #b:Type) (f:a → b) (t:tainted a) : tainted b =
  { value = f t.value; taint = t.taint }

let taint_map2 (#a #b #c:Type) (f:a → b → c) (t1:tainted a) (t2:
   tainted b)
    : tainted c =
  { value = f t1.value t2.value; taint = taint_join t1.taint t2.taint
     }

(* Example: SQL query building *)
let user_input : tainted string =
  source TaintSQLi "Robert'; DROP TABLE students;--"

let build_query (name:tainted string) : tainted string =
  taint_map (fun n → "SELECT * FROM users WHERE name = '" ^ n ^ "'")
     name

let execute_sql (query:tainted string) : option (list string) =
  match sink TaintSQLi query with
  | Some q → Some ["result"]  (* Execute query *)
  | None → None  (* Blocked: tainted SQL *)

(* This correctly returns None - SQL injection prevented *)
let vulnerable_query = execute_sql (build_query user_input)

(* Safe version with sanitization *)
let escape_sql (s:string) : string =
  (* Replace ' with '' *)
  String.concat "''" (String.split_on_char '\'' s)

let safe_query =
  let sanitized = sanitize TaintSQLi user_input escape_sql in
  execute_sql (build_query sanitized)
```

```
(* This returns Some [...] - sanitized input is safe *)

(* Taint tracking through data structures *)
let taint_list (#a:Type) (l:list (tainted a)) : tainted (list a) =
  let values = List.map (fun t → t.value) l in
  let taint = List.fold_left (fun acc t → taint_join acc t.taint)
      Untainted l in
  { value = values; taint = taint }
```

34. Contracts and Verification

34.1 Function Contracts

Definition 34.1 (Contract)**.** A function contract specifies:

$$\begin{aligned} \mathsf{fn}\ f(x:\tau) \to \sigma \\ \mathsf{requires}\ P(x) \\ \mathsf{ensures}\ Q(x, \mathsf{result}) \end{aligned}$$

where P is the precondition and Q is the postcondition.

Definition 34.2 (Contract Typing)**.**

$$\frac{\Gamma, x:\tau, P(x) \vdash e : \sigma \qquad \Gamma, x:\tau, P(x) \vdash Q(x,e)}{\Gamma \vdash (\mathsf{fn}\ f(x)\ \mathsf{requires}\ P\ \mathsf{ensures}\ Q := e) : \{x:\tau \mid P\} \to \{r:\sigma \mid Q\}}\ \text{T-Contract}$$

34.2 Hoare Logic

Definition 34.3 (Hoare Triple)**.** $\{P\}\,c\,\{Q\}$ means: if P holds before c, and c terminates, then Q holds after.

Definition 34.4 (Core Hoare Rules)**.**

$$\frac{}{\{P\}\,\mathsf{skip}\,\{P\}}\ \text{H-Skip} \qquad \frac{}{\{P[x:=e]\}\,x := e\,\{P\}}\ \text{H-Assign}$$

$$\frac{\{P\}\,c_1\,\{R\} \qquad \{R\}\,c_2\,\{Q\}}{\{P\}\,c_1; c_2\,\{Q\}}\ \text{H-Seq} \qquad \frac{\{P \wedge b\}\,c_1\,\{Q\} \qquad \{P \wedge \neg b\}\,c_2\,\{Q\}}{\{P\}\,\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\,\{Q\}}\ \text{H-If}$$

$$\frac{\{I \wedge b\}\,c\,\{I\}}{\{I\}\,\mathsf{while}\ b\ \mathsf{do}\ c\,\{I \wedge \neg b\}}\ \text{H-While} \qquad \frac{P' \Rightarrow P \qquad \{P\}\,c\,\{Q\} \qquad Q \Rightarrow Q'}{\{P'\}\,c\,\{Q'\}}\ \text{H-Conseq}$$

34.3 Refinement Types

Definition 34.5 (Refinement Type)**.**

$$\{x : \tau \mid \phi(x)\}$$

The type of values of type τ satisfying predicate ϕ.

Definition 34.6 (Refinement Subtyping)**.**

$$\{x:\tau \mid \phi_1(x)\} <: \{x:\tau \mid \phi_2(x)\} \iff \forall x.\, \phi_1(x) \Rightarrow \phi_2(x)$$

Example 34.7 (Refined Types)**.**

$$\begin{aligned} \mathsf{Nat} &= \{x : \mathsf{Int} \mid x \geq 0\} \\ \mathsf{Pos} &= \{x : \mathsf{Int} \mid x > 0\} \\ \mathsf{NonEmpty}[A] &= \{xs : \mathsf{List}[A] \mid \mathsf{length}(xs) > 0\} \end{aligned}$$

34.4 F* Mechanization

Contracts in F*

```
module BrrrSemantics.Contracts

(* Refinement type in F* is native *)
type nat = x:int{x ≥ 0}
type pos = x:int{x > 0}

(* Non-empty list *)
type non_empty_list (a:Type) = l:list a{Cons? l}

(* Function with contract *)
let divide (x:int) (y:pos) : int =
  x / y
(* Precondition y > 0 is enforced by type *)
(* Division by zero is impossible! *)

(* Postcondition via refinement *)
let abs (x:int) : y:nat{y ≥ 0 ∧ (x ≥ 0 ⟹ y = x) ∧ (x < 0 ⟹ y = -x)}
   =
  if x ≥ 0 then x else -x

(* Loop invariant *)
let rec sum_to (n:nat) : Tot (r:nat{r = n * (n + 1) / 2}) (decreases n
   ) =
  if n = 0 then 0
  else n + sum_to (n - 1)

(* Hoare triple representation *)
noeq type hoare_triple (pre:Type0) (post:'a → Type0) =
  | HT : (unit → Pure 'a (requires pre) (ensures post)) → hoare_triple
      pre post

(* Sequential composition *)
let seq_hoare (#a #b:Type) (#p #q #r:Type0)
              (h1:hoare_triple p (fun _ → q))
              (h2:hoare_triple q (fun _ → r))
    : hoare_triple p (fun _ → r) =
  HT (fun () →
    let HT f1 = h1 in
    let HT f2 = h2 in
    let _ = f1 () in
    f2 ())

(* Array bounds checking via refinement *)
type bounded_index (len:nat) = i:nat{i < len}

let safe_array_access (#a:Type) (arr:list a) (i:bounded_index (List.
   length arr)) : a =
  List.nth arr i
(* Index out of bounds is impossible! *)

(* Sorted list invariant *)
let rec sorted (l:list int) : bool =
  match l with
  | [] → true
  | [_] → true
  | x :: y :: rest → x ≤ y && sorted (y :: rest)
```

```
type sorted_list = l:list int{sorted l}

(* Insert maintaining sortedness *)
let rec insert (x:int) (l:sorted_list) : sorted_list =
  match l with
  | [] → [x]
  | h :: t →
      if x ≤ h then x :: l
      else h :: insert x t

(* Verification condition generation *)
type vc = | VCTrue | VCImpl : vc → vc → vc | VCAnd : vc → vc → vc

let rec check_vc (v:vc) : bool =
  match v with
  | VCTrue → true
  | VCImpl p q → not (check_vc p) || check_vc q
  | VCAnd p q → check_vc p && check_vc q
```

35. Separation Logic Integration

35.1 Heap Assertions

Definition 35.1 (Separation Logic Assertions).

$$\begin{aligned} P, Q ::= {} & \mathsf{emp} && \text{(empty heap)} \\ & \mid e \mapsto v && \text{(points-to)} \\ & \mid P * Q && \text{(separating conjunction)} \\ & \mid P \mathrel{-\!\!*} Q && \text{(magic wand)} \\ & \mid \forall x.\, P \mid \exists x.\, P && \text{(quantifiers)} \end{aligned}$$

Definition 35.2 (Separating Conjunction Semantics). $P * Q$ holds on heap h iff $h = h_1 \uplus h_2$ where P holds on h_1 and Q holds on h_2 (disjoint heaps).

35.2 Frame Rule

Theorem 35.3 (Frame Rule).

$$\frac{\{P\}\, c\, \{Q\} \qquad \mathsf{mod}(c) \cap \mathsf{fv}(R) = \emptyset}{\{P * R\}\, c\, \{Q * R\}}\ \textsc{Frame}$$

Local reasoning: if c only touches heap described by P, then R is preserved.

35.3 Ownership Integration

Definition 35.4 (Ownership as Separation). Brrr-Lang ownership maps to separation logic:

$$\begin{aligned} \mathsf{own}\, x &\cong x \mapsto v * \mathsf{Freeable}(x) \\ \mathsf{ref}\, x &\cong x \mapsto^{1/n} v \\ \mathsf{ref\ mut}\, x &\cong x \mapsto v \end{aligned}$$

35.4 F* Mechanization

Separation Logic in F*

```
module BrrrSemantics.SeparationLogic

(* Heap location *)
type loc = nat

(* Heap: partial map from locations to values *)
type heap = loc → option value

(* Empty heap *)
let emp : heap = fun _ → None

(* Singleton heap *)
let singleton (l:loc) (v:value) : heap =
  fun l' → if l = l' then Some v else None

(* Heap disjointness *)
let disjoint (h1 h2:heap) : bool =
  ∀ l. not (Some? (h1 l) && Some? (h2 l))

(* Heap union (requires disjoint) *)
let heap_union (h1 h2:heap) : heap =
  fun l → match h1 l with Some v → Some v | None → h2 l

(* Separation logic assertion *)
noeq type sl_assertion =
  | SLEmp      : sl_assertion
  | SLPointsTo: loc → value → sl_assertion
  | SLStar     : sl_assertion → sl_assertion → sl_assertion
  | SLWand     : sl_assertion → sl_assertion → sl_assertion
  | SLForall   : (value → sl_assertion) → sl_assertion
  | SLExists   : (value → sl_assertion) → sl_assertion
  | SLPure     : bool → sl_assertion

(* Assertion satisfaction *)
let rec satisfies (h:heap) (p:sl_assertion) : bool =
  match p with
  | SLEmp → h = emp
  | SLPointsTo l v → h = singleton l v
  | SLStar p1 p2 →
      ∃ h1 h2. disjoint h1 h2 &&
                    heap_union h1 h2 = h &&
                    satisfies h1 p1 &&
                    satisfies h2 p2
  | SLWand p1 p2 →
      ∀ h'. disjoint h h' && satisfies h' p1 ⟹
                 satisfies (heap_union h h') p2
  | SLPure b → b && h = emp
  | _ → admit()

(* Separation logic Hoare triple *)
noeq type sl_triple = {
  pre  : sl_assertion;
  cmd  : expr;
  post : sl_assertion
}

(* Frame rule *)
let frame_rule (t:sl_triple) (frame:sl_assertion) : sl_triple =
  { pre = SLStar t.pre frame;
    cmd = t.cmd;
    post = SLStar t.post frame }

(* Allocation rule *)
(* {emp} x := alloc(v) {x ↦ v} *)
let alloc_rule (v:value) : sl_triple =
  { pre = SLEmp;
    cmd = (* alloc *) ELit VUnit;
    post = SLExists (fun l → SLPointsTo l v) }

(* Deallocation rule *)
(* {x ↦ v} free(x) {emp} *)
let free_rule (l:loc) (v:value) : sl_triple =
  { pre = SLPointsTo l v;
    cmd = (* free *) ELit VUnit;
    post = SLEmp }

(* Read rule *)
(* {x ↦ v} y := *x {x ↦ v ∧ y = v} *)
let read_rule (l:loc) (v:value) : sl_triple =
  { pre = SLPointsTo l v;
    cmd = (* read *) ELit VUnit;
    post = SLStar (SLPointsTo l v) (SLPure true) }

(* Write rule *)
(* {x ↦ _} *x := v {x ↦ v} *)
let write_rule (l:loc) (v:value) : sl_triple =
  { pre = SLExists (fun _ → SLPointsTo l (VBase VUnit));
    cmd = (* write *) ELit VUnit;
    post = SLPointsTo l v }

(* Ownership transfer via frame *)
let transfer_ownership (src dst:loc) (v:value) : sl_triple =
  (* {src ↦ v * dst ↦ _} move {src ↦ ⊥ * dst ↦ v} *)
  { pre = SLStar (SLPointsTo src v) (SLExists (fun _ → SLPointsTo dst
      (VBase VUnit)));
    cmd = ELit VUnit;
    post = SLStar SLEmp (SLPointsTo dst v) }
```

Part X

FFI & Interop

36. Foreign Function Interface

The FFI enables Brrr-Lang to interface with code written in other languages while maintaining type safety.

36.1 Foreign Type Declarations

Definition 36.1 (Foreign Type)**.** A foreign type represents an opaque type from another language:

$$\mathsf{foreign\ type}\ T$$

The type T has no Brrr-Lang representation; it exists only at FFI boundaries.

Definition 36.2 (Foreign Function Declaration)**.**

$$\begin{aligned}&\mathsf{extern}\ \text{“}\mathsf{C}'' \ \{\\ &\quad \mathsf{fn}\ f(x_1 : \tau_1, \ldots, x_n : \tau_n) \to \sigma;\\ &\}\end{aligned}$$

The calling convention (“C”, “Rust”, “System”) determines ABI.

Definition 36.3 (FFI Typing)**.**

$$\frac{\forall i.\ \tau_i \in \mathsf{FFISafe} \qquad \sigma \in \mathsf{FFISafe}}{\Gamma \vdash \mathsf{extern}\ f : (\tau_1, \ldots, \tau_n) \to \sigma\ [\mathsf{FFI}]}\ \text{T-Extern}$$

where FFISafe is the set of types with defined ABI representation.

36.2 FFI-Safe Types

Definition 36.4 (FFI-Safe Type Set)**.**

$$\begin{aligned}\mathsf{FFISafe} ::= {}& \mathsf{Int}[w, s] \text{ for } w \in \{8, 16, 32, 64\}\\ &\mid \mathsf{Float}[p] \text{ for } p \in \{32, 64\}\\ &\mid \mathsf{Ptr}[\tau] \text{ for } \tau \in \mathsf{FFISafe}\\ &\mid \mathsf{FnPtr}[\tau_1, \ldots, \tau_n \to \sigma]\\ &\mid \mathsf{CStr}\\ &\mid \mathsf{Struct}[f_1 : \tau_1, \ldots, f_n : \tau_n] \text{ with } \#[\mathsf{repr(C)}]\end{aligned}$$

Definition 36.5 (Type Representation Attribute)**.**

$\#[\mathsf{repr(C)}]$	C-compatible layout
$\#[\mathsf{repr(transparent)}]$	Same as inner type
$\#[\mathsf{repr(packed)}]$	No padding
$\#[\mathsf{repr(align}(n))]$	Alignment to n bytes

36.3 Safety and Unsafety

Definition 36.6 (Unsafe Block)**.** FFI calls require unsafe context:

$$\mathsf{unsafe}\ \{\ldots\}$$

Inside unsafe, the programmer asserts memory safety invariants.

Definition 36.7 (Unsafe Typing)**.**

$$\frac{\Gamma \vdash e : \tau\ [\langle \mathsf{Unsafe} \mid \varepsilon\rangle]}{\Gamma \vdash \mathsf{unsafe}\ \{e\} : \tau\ [\varepsilon]}\ \text{T-Unsafe}$$

Unsafe blocks discharge the Unsafe effect.

36.4 F* Mechanization

FFI in F*

```
module BrrrSemantics.FFI

(* Calling convention *)
type calling_convention =
  | CC_C          : calling_convention
  | CC_Rust       : calling_convention
  | CC_System     : calling_convention
  | CC_Fastcall   : calling_convention

(* FFI-safe types *)
noeq type ffi_safe_type =
  | FFIInt    : int_width → signedness → ffi_safe_type
  | FFIFloat  : float_prec → ffi_safe_type
  | FFIPtr    : ffi_safe_type → ffi_safe_type
  | FFIFnPtr  : list ffi_safe_type → ffi_safe_type → ffi_safe_type
  | FFICStr   : ffi_safe_type
  | FFIVoid   : ffi_safe_type
  | FFIStruct : list (string & ffi_safe_type) → ffi_safe_type

(* Check if type is FFI-safe *)
let rec is_ffi_safe (t:brrr_type) : option ffi_safe_type =
  match t with
  | TInt {width; sign} →
      (match width with
       | I8 | I16 | I32 | I64 → Some (FFIInt width sign)
       | _ → None)  (* I128, IBig not FFI-safe *)
  | TFloat p →
      (match p with
       | F32 | F64 → Some (FFIFloat p)
       | F16 → None)
  | TUnit → Some FFIVoid
  | _ → None  (* Would need more cases *)

(* Foreign function declaration *)
noeq type extern_fn = {
  name        : string;
  convention  : calling_convention;
  params      : list (string & ffi_safe_type);
  return_type : ffi_safe_type;
  is_variadic : bool
}

(* Type representation *)
type repr_attr =
  | ReprC            : repr_attr
  | ReprTransparent  : repr_attr
  | ReprPacked       : repr_attr
  | ReprAlign        : nat → repr_attr

(* Struct with repr *)
noeq type ffi_struct = {
  name   : string;
  repr   : repr_attr;
  fields : list (string & ffi_safe_type)
}
```

```
(* Compute struct layout for repr(C) *)
let rec compute_c_layout (fields:list (string & ffi_safe_type))
    : list (string & nat (* offset *) & nat (* size *)) =
  let rec go fields offset acc =
    match fields with
    | [] → List.rev acc
    | (name, ty) :: rest →
        let size = ffi_type_size ty in
        let align = ffi_type_align ty in
        let padded_offset = align_up offset align in
        go rest (padded_offset + size) ((name, padded_offset, size) ::
            acc)
  in go fields 0 []

and ffi_type_size (t:ffi_safe_type) : nat =
  match t with
  | FFIInt I8 _ → 1
  | FFIInt I16 _ → 2
  | FFIInt I32 _ → 4
  | FFIInt I64 _ → 8
  | FFIFloat F32 → 4
  | FFIFloat F64 → 8
  | FFIPtr _ → 8  (* Assuming 64-bit *)
  | FFIFnPtr _ _ → 8
  | FFICStr → 8  (* Pointer *)
  | FFIVoid → 0
  | FFIStruct fields → admit()  (* Sum of field sizes + padding *)
  | _ → 0

and ffi_type_align (t:ffi_safe_type) : nat =
  ffi_type_size t  (* Simplified: align = size *)

and align_up (n align:nat) : nat =
  if align = 0 then n
  else ((n + align - 1) / align) * align

(* Unsafe effect *)
type unsafe_effect = | UnsafeFFI | UnsafePtr | UnsafeTransmute

(* Unsafe context tracking *)
let in_unsafe_block : ref bool = ref false

(* Check if we can call FFI *)
let check_ffi_call (fn:extern_fn) : bool =
  !in_unsafe_block

(* Unsafe block execution *)
let with_unsafe (#a:Type) (body:unit → a) : a =
  in_unsafe_block := true;
  let result = body () in
  in_unsafe_block := false;
  result
```

37. Memory Layout

37.1 Data Layout Specification

Definition 37.1 (Layout)**.** A layout specifies size, alignment, and field positions:

$$\mathsf{Layout} = \{\mathsf{size} : \mathbb{N}, \mathsf{align} : \mathbb{N}, \mathsf{fields} : \mathsf{List}[(\mathsf{Name}, \mathsf{Offset}, \mathsf{Layout})]\}$$

Definition 37.2 (Primitive Layouts)**.**

Type	Size (bytes)	Alignment
$\mathsf{Int}[8, \_]$	1	1
$\mathsf{Int}[16, \_]$	2	2
$\mathsf{Int}[32, \_]$	4	4
$\mathsf{Int}[64, \_]$	8	8
$\mathsf{Float}[32]$	4	4
$\mathsf{Float}[64]$	8	8
$\mathsf{Ptr}[\_]$	8	8

37.2 Struct Layout Rules

Definition 37.3 (C Layout Algorithm)**.** For #[repr(C)] structs:

1. Start at offset 0
2. For each field in declaration order:
   (a) Pad to field's alignment
   (b) Place field at current offset
   (c) Advance offset by field size
3. Final size is padded to struct alignment
4. Struct alignment is max of field alignments

Definition 37.4 (Rust Layout)**.** Without repr attribute, the compiler may:

- Reorder fields to minimize padding
- Use niche optimization for enums
- Apply other optimizations

37.3 Pointer Operations

Definition 37.5 (Pointer Arithmetic)**.**

$$\mathsf{offset} : \mathsf{Ptr}[\tau] \times \mathsf{Int} \to \mathsf{Ptr}[\tau]$$
$$\mathsf{offset}(p, n) = p + n \cdot \mathsf{sizeof}(\tau)$$

Definition 37.6 (Pointer Cast)**.**

$$\frac{\Gamma \vdash p : \mathsf{Ptr}[\tau_1]\ [\varepsilon]}{\Gamma \vdash p \ \mathsf{as}\ \mathsf{Ptr}[\tau_2] : \mathsf{Ptr}[\tau_2]\ [\langle \mathsf{Unsafe} \mid \varepsilon \rangle]}\ \textsc{T-PtrCast}$$

Pointer casts require unsafe and may violate aliasing rules.

37.4 F* Mechanization

Memory Layout in F*

```
module BrrrSemantics.Layout

(* Layout descriptor *)
```

```
noeq type layout = {
  size  : nat;
  align : nat
}

(* Field layout *)
noeq type field_layout = {
  name   : string;
  offset : nat;
  layout : layout
}

(* Struct layout *)
noeq type struct_layout = {
  total_size  : nat;
  total_align : nat;
  fields      : list field_layout
}

(* Primitive layouts *)
let layout_i8  : layout = { size = 1; align = 1 }
let layout_i16 : layout = { size = 2; align = 2 }
let layout_i32 : layout = { size = 4; align = 4 }
let layout_i64 : layout = { size = 8; align = 8 }
let layout_f32 : layout = { size = 4; align = 4 }
let layout_f64 : layout = { size = 8; align = 8 }
let layout_ptr : layout = { size = 8; align = 8 }

(* Get layout for FFI type *)
let ffi_layout (t:ffi_safe_type) : layout =
  match t with
  | FFIInt I8 _ → layout_i8
  | FFIInt I16 _ → layout_i16
  | FFIInt I32 _ → layout_i32
  | FFIInt I64 _ → layout_i64
  | FFIFloat F32 → layout_f32
  | FFIFloat F64 → layout_f64
  | FFIPtr _ → layout_ptr
  | FFIFnPtr _ _ → layout_ptr
  | FFICStr → layout_ptr
  | FFIVoid → { size = 0; align = 1 }
  | FFIStruct _ → admit()  (* Recursive *)
  | _ → { size = 0; align = 1 }

(* Align offset *)
let align_offset (offset align:nat) : nat =
  if align = 0 then offset
  else
    let rem = offset % align in
    if rem = 0 then offset else offset + (align - rem)

(* Compute C struct layout *)
let compute_struct_layout (fields:list (string & ffi_safe_type)) :
   struct_layout =
  let rec go fields offset max_align acc =
    match fields with
    | [] →
```

```
        let final_size = align_offset offset max_align in
        { total_size = final_size; total_align = max_align; fields =
            List.rev acc }
    | (name, ty) :: rest →
        let ly = ffi_layout ty in
        let aligned_offset = align_offset offset ly.align in
        let field_ly = { name = name; offset = aligned_offset; layout
            = ly } in
        go rest (aligned_offset + ly.size) (max max_align ly.align) (
            field_ly :: acc)
  in go fields 0 1 []

(* Pointer type *)
noeq type ptr (a:Type) = {
  addr : nat;
  phantom : a → unit  (* Type witness *)
}

(* Null pointer *)
let null (#a:Type) : ptr a = { addr = 0; phantom = fun _ → () }

(* Pointer arithmetic *)
let ptr_offset (#a:Type) (p:ptr a) (n:int) (elem_size:nat) : ptr a =
  { p with addr = p.addr + n * elem_size }

(* Check pointer validity *)
let is_valid_ptr (#a:Type) (p:ptr a) (heap_size:nat) (elem_size:nat) :
    bool =
  p.addr > 0 &&
  p.addr + elem_size ≤ heap_size &&
  p.addr % elem_size = 0  (* Aligned *)

(* Pointer cast (unsafe) *)
let ptr_cast (#a #b:Type) (p:ptr a) : ptr b =
  { addr = p.addr; phantom = fun _ → () }

(* Read through pointer *)
let ptr_read (#a:Type) (p:ptr a) (heap:nat → option a) : option a =
  if p.addr = 0 then None
  else heap p.addr

(* Write through pointer *)
let ptr_write (#a:Type) (p:ptr a) (v:a) (heap:nat → option a) : nat →
    option a =
  if p.addr = 0 then heap
  else fun addr → if addr = p.addr then Some v else heap addr
```

38. Cross-Language Semantics

38.1 Translation Functors

Definition 38.1 (Translation Functor). A translation from source language L to Brrr-Lang is a functor:

$$\mathcal{T}_L : \mathsf{Prog}_L \to \mathsf{Prog}_{\mathsf{Brrr}}$$

preserving typing: if $\Gamma \vdash_L e : \tau$, then $\mathcal{T}_L(\Gamma) \vdash_{\mathsf{Brrr}} \mathcal{T}_L(e) : \mathcal{T}_L(\tau)$.

Definition 38.2 (Soundness)**.** Translation $\mathcal{T}_L$ is sound iff:

$$[\![e]\!]_L = [\![\mathcal{T}_L(e)]\!]_{\mathsf{Brrr}}$$

The Brrr-Lang program has the same behavior as the source.

38.2 Language Mode Configuration

Definition 38.3 (Language Mode)**.** Each source language is characterized by a 5-tuple of modes:

$$\mathsf{Mode}_L = (\mathsf{Memory}, \mathsf{Types}, \mathsf{Null}, \mathsf{Effects}, \mathsf{Concurrency})$$

Definition 38.4 (Mode Axes)**.**

Axis	Values
Memory	$\{\mathsf{GC}, \mathsf{RC}, \mathsf{Manual}, \mathsf{Ownership}\}$
Types	$\{\mathsf{Static}, \mathsf{Gradual}, \mathsf{Dynamic}\}$
Null	$\{\mathsf{Nullable}, \mathsf{Optional}, \mathsf{NonNull}\}$
Effects	$\{\mathsf{Pure}, \mathsf{Tracked}, \mathsf{Untracked}\}$
Concurrency	$\{\mathsf{None}, \mathsf{Threads}, \mathsf{Async}, \mathsf{Actors}\}$

Example 38.5 (Language Configurations)**.**

Language	Memory	Types	Null	Effects	Concurrency
Rust	Ownership	Static	Optional	Tracked	Async
Python	GC	Dynamic	Nullable	Untracked	Async
TypeScript	GC	Gradual	Nullable	Untracked	Async
Go	GC	Static	Nullable	Untracked	Async
Swift	RC	Static	Optional	Untracked	Async
Java	GC	Static	Nullable	Untracked	Threads

38.3 Boundary Guards

Definition 38.6 (Boundary)**.** A language boundary $L_1 \leftrightarrow L_2$ is where code from L_1 calls into L_2.

Definition 38.7 (Guard)**.** A guard ensures properties hold at boundaries:

$$\mathsf{guard}_\tau^{L_1 \to L_2} : \mathcal{T}_{L_1}(\tau) \to \mathcal{T}_{L_2}(\tau)$$
$$\mathsf{guard}_{\mathsf{Int}}^{\mathsf{Py} \to \mathsf{Rust}} = \lambda x.\,\mathsf{if\ is\_int}(x)\ \mathsf{then}\ x\ \mathsf{else\ panic}$$

38.4 F* Mechanization

Cross-Language Semantics in F*

```
module BrrrSemantics.CrossLanguage

(* Memory mode *)
type memory_mode = | MemGC | MemRC | MemManual | MemOwnership

(* Type mode *)
type type_mode = | TypeStatic | TypeGradual | TypeDynamic

(* Null mode *)
type null_mode = | NullNullable | NullOptional | NullNonNull

(* Effect mode *)
type effect_mode = | EffPure | EffTracked | EffUntracked
```

```
(* Concurrency mode *)
type conc_mode = | ConcNone | ConcThreads | ConcAsync | ConcActors

(* Language mode configuration *)
noeq type lang_mode = {
  memory      : memory_mode;
  types       : type_mode;
  null_safety : null_mode;
  effects     : effect_mode;
  concurrency : conc_mode
}

(* Standard language configurations *)
let rust_mode : lang_mode = {
  memory = MemOwnership;
  types = TypeStatic;
  null_safety = NullOptional;
  effects = EffTracked;
  concurrency = ConcAsync
}

let python_mode : lang_mode = {
  memory = MemGC;
  types = TypeDynamic;
  null_safety = NullNullable;
  effects = EffUntracked;
  concurrency = ConcAsync
}

let typescript_mode : lang_mode = {
  memory = MemGC;
  types = TypeGradual;
  null_safety = NullNullable;
  effects = EffUntracked;
  concurrency = ConcAsync
}

let go_mode : lang_mode = {
  memory = MemGC;
  types = TypeStatic;
  null_safety = NullNullable;
  effects = EffUntracked;
  concurrency = ConcAsync
}

(* Translation context *)
noeq type trans_ctx = {
  source_mode : lang_mode;
  target_mode : lang_mode   (* Always Brrr *)
}

(* Type translation *)
let rec translate_type (ctx:trans_ctx) (t:brrr_type) : brrr_type =
  match ctx.source_mode.null_safety, t with
  | NullNullable, t → TOption t  (* Lift nullable to Option *)
  | NullOptional, t → t          (* Already Option *)
  | NullNonNull, t → t           (* No wrapping *)
```

```
(* Guard generation *)
type guard_result (a:Type) =
  | GuardOk  : a → guard_result a
  | GuardErr : string → guard_result a

let generate_guard (from_mode to_mode:lang_mode) (t:brrr_type)
    : value → guard_result value =
  match from_mode.types, to_mode.types with
  | TypeDynamic, TypeStatic →
      (* Need runtime type check *)
      fun v → match runtime_typecheck v t with
        | true → GuardOk v
        | false → GuardErr "Type mismatch at boundary"
  | TypeStatic, TypeDynamic →
      (* Can always widen to dynamic *)
      fun v → GuardOk v
  | _, _ →
      fun v → GuardOk v

and runtime_typecheck (v:value) (t:brrr_type) : bool =
  match v, t with
  | VBase (VInt _), TInt _ → true
  | VBase (VBool _), TBool → true
  | VBase (VString _), TString → true
  | _, _ → false

(* Memory mode translation *)
let translate_ownership (from_mode:lang_mode) (t:brrr_type) :
   moded_type =
  match from_mode.memory with
  | MemOwnership → { ty = t; mode = MOne }        (* Linear *)
  | MemRC → { ty = t; mode = MOmega }             (* Shared via RC *)
  | MemGC → { ty = t; mode = MOmega }             (* Shared via GC *)
  | MemManual → { ty = t; mode = MOne }           (* Must track *)

(* Effect translation *)
let translate_effects (from_mode:lang_mode) (eff:effect_row) :
   effect_row =
  match from_mode.effects with
  | EffPure → eff                      (* Keep precise effects *)
  | EffTracked → eff                   (* Keep precise effects *)
  | EffUntracked → Var "ε"            (* Abstract over effects *)

(* Full translation *)
let translate_expr (ctx:trans_ctx) (e:expr) : expr =
  (* Would traverse and apply translations *)
  e
```

Part XI

Physical Representation

39. Content-Addressed Nodes

This chapter specifies how Brrr-Lang IR is stored for efficient analysis.

39.1 Merkle Hashing

Definition 39.1 (Content Address)**.** Every IR node has a content address (hash):

$$\mathsf{hash} : \mathsf{Node} \to \mathsf{Hash}_{256}$$

where Hash_{256} is a 256-bit BLAKE3 hash.

Definition 39.2 (Node Hash Computation)**.**

$$\begin{aligned}\mathsf{hash}(\mathsf{Leaf}(v)) &= \mathsf{BLAKE3}(\mathsf{tag}\|\mathsf{encode}(v))\\ \mathsf{hash}(\mathsf{Node}(k, c_1, \ldots, c_n)) &= \mathsf{BLAKE3}(\mathsf{tag}\|k\|\mathsf{hash}(c_1)\| \cdots \|\mathsf{hash}(c_n))\end{aligned}$$

where tag distinguishes node types.

Theorem 39.3 (Structural Sharing)**.** *If* $\mathsf{hash}(n_1) = \mathsf{hash}(n_2)$*, then* $n_1 = n_2$ *(with overwhelming probability).*

This enables:

- *Deduplication of identical subtrees*
- *O(1) equality checking*
- *Efficient caching and memoization*

39.2 Node Index

Definition 39.4 (Node ID)**.** A node ID is a 32-bit index into the node array:

$$\mathsf{NodeId} = \mathbb{N}_{32}$$

Definition 39.5 (DFS Ordering)**.** Nodes are stored in depth-first order:

- Parent before children
- Left-to-right child ordering
- Enables subtree as contiguous range

39.3 F* Mechanization

Content Addressing in F*

```
module BrrrSemantics.ContentAddressing

(* 256-bit hash *)
type hash256 = {
  bytes : list nat (* 32 bytes *)
}

(* Node ID *)
type node_id = nat

(* Node kind tag *)
type node_tag =
  | TagExpr    : nat → node_tag  (* Expression kind *)
  | TagStmt    : nat → node_tag  (* Statement kind *)
  | TagType    : nat → node_tag  (* Type kind *)
  | TagPattern : nat → node_tag  (* Pattern kind *)
```

```
(* IR Node *)
noeq type ir_node =
  | IRLeaf : node_tag → list nat (* encoded data *) → ir_node
  | IRNode : node_tag → list node_id (* children *) → ir_node

(* BLAKE3 hash (simplified) *)
let blake3 (data:list nat) : hash256 =
  (* Real impl would use BLAKE3 algorithm *)
  { bytes = List.take 32 (data @ List.init 32 (fun _ → 0)) }

(* Encode value to bytes *)
let encode_value (v:value) : list nat =
  match v with
  | VBase (VInt n) → [0; n % 256; (n / 256) % 256; (n / 65536) % 256;
     n / 16777216]
  | VBase (VBool b) → [1; if b then 1 else 0]
  | VBase (VString s) → [2] @ List.map Char.code (String.to_list s)
  | _ → [255]

(* Hash computation *)
let rec compute_hash (node:ir_node) (get_hash:node_id → hash256) :
    hash256 =
  match node with
  | IRLeaf tag data →
      let tag_bytes = match tag with
        | TagExpr n → [0; n]
        | TagStmt n → [1; n]
        | TagType n → [2; n]
        | TagPattern n → [3; n]
      in
      blake3 (tag_bytes @ data)

  | IRNode tag children →
      let tag_bytes = match tag with
        | TagExpr n → [0; n]
        | TagStmt n → [1; n]
        | TagType n → [2; n]
        | TagPattern n → [3; n]
      in
      let child_hashes = List.concat_map (fun cid →
        (get_hash cid).bytes) children in
      blake3 (tag_bytes @ child_hashes)

(* Hash table for deduplication *)
type hash_table = list (hash256 & node_id)

let lookup_hash (h:hash256) (table:hash_table) : option node_id =
  match List.find (fun (h', _) → h.bytes = h'.bytes) table with
  | Some (_, id) → Some id
  | None → None

(* Insert with deduplication *)
let insert_node (node:ir_node) (nodes:list ir_node) (table:hash_table)
    : node_id & list ir_node & hash_table =
  let h = compute_hash node (fun id →
    compute_hash (List.nth nodes id) (fun _ → { bytes = [] })) in
  match lookup_hash h table with
  | Some existing_id → (existing_id, nodes, table)  (* Deduplicated!
     *)
  | None →
      let new_id = List.length nodes in
      (new_id, nodes @ [node], (h, new_id) :: table)

(* DFS traversal order *)
let rec dfs_order (root:ir_node) (get_children:ir_node → list ir_node)
    : list ir_node =
  root :: List.concat_map (fun c → dfs_order c get_children) (
     get_children root)

(* Check if range is contiguous subtree *)
let is_subtree_range (start_id end_id:node_id) (nodes:list ir_node) :
    bool =
  (* In DFS order, subtree is always contiguous *)
  start_id ≤ end_id && end_id < List.length nodes
```

40. Columnar Storage

40.1 Structure of Arrays

Definition 40.1 (SoA Layout)**.** Instead of array of structs (AoS), use structure of arrays (SoA):

$$\begin{aligned}\text{AoS:}\quad & [(\mathsf{kind}_1, \mathsf{span}_1), (\mathsf{kind}_2, \mathsf{span}_2), \ldots] \\ \text{SoA:}\quad & ([\mathsf{kind}_1, \mathsf{kind}_2, \ldots], [\mathsf{span}_1, \mathsf{span}_2, \ldots])\end{aligned}$$

Theorem 40.2 (SoA Benefits)**.** • *Cache efficiency: accessing one field hits sequential memory*
- *SIMD: can process 8/16/32 elements per instruction*
- *Compression: similar values cluster together*

40.2 Column Types

Definition 40.3 (Node Columns)**.**

$$\begin{aligned}\mathsf{kinds} : \mathsf{Array}[\mathsf{NodeKind}] &\qquad \text{(1 byte each)} \\ \mathsf{spans} : \mathsf{Array}[\mathsf{Span}] &\qquad \text{(8 bytes each)} \\ \mathsf{types} : \mathsf{Array}[\mathsf{TypeId}] &\qquad \text{(4 bytes each)} \\ \mathsf{parents} : \mathsf{Array}[\mathsf{NodeId}] &\qquad \text{(4 bytes each)} \\ \mathsf{first\_child} : \mathsf{Array}[\mathsf{NodeId}] &\qquad \text{(4 bytes each)} \\ \mathsf{next\_sibling} : \mathsf{Array}[\mathsf{NodeId}] &\qquad \text{(4 bytes each)}\end{aligned}$$

40.3 Edge Storage (CSR)

Definition 40.4 (Compressed Sparse Row)**.** Edges stored in CSR format:

$$\begin{aligned}\mathsf{row\_ptr} : \mathsf{Array}[\mathsf{EdgeIdx}] &\qquad \text{(start of edges for node } i\text{)} \\ \mathsf{col\_idx} : \mathsf{Array}[\mathsf{NodeId}] &\qquad \text{(target nodes)} \\ \mathsf{edge\_data} : \mathsf{Array}[\mathsf{EdgeKind}] &\qquad \text{(optional edge labels)}\end{aligned}$$

Edges from node i are at indices $[\mathsf{row\_ptr}[i], \mathsf{row\_ptr}[i+1])$ in $\mathsf{col\_idx}$.

Theorem 40.5 (CSR Complexity)**.** • *Space: $O(V+E)$ instead of $O(V^2)$ for adjacency matrix*
- *Iteration over successors: $O(\text{out-degree})$*
- *SIMD-friendly for sparse operations*

40.4 F* Mechanization

Columnar Storage in F*

```
module BrrrSemantics.ColumnarStorage

(* Node kind enum (fits in 1 byte) *)
type node_kind = nat (* 0-255 *)

(* Source span *)
noeq type span = {
  file_id : nat;
  start   : nat;
  end_    : nat
}

(* Type ID (index into type table) *)
type type_id = nat

(* Columnar node storage *)
noeq type node_columns = {
  kinds       : list node_kind;     (* Node kinds *)
  spans       : list span;          (* Source locations *)
  types       : list type_id;       (* Type annotations *)
  parents     : list node_id;       (* Parent pointers *)
  first_child : list node_id;       (* First child (0 = none) *)
  next_sibling: list node_id        (* Next sibling (0 = none) *)
}

(* Create empty columns *)
let empty_columns : node_columns = {
  kinds = [];
  spans = [];
  types = [];
  parents = [];
  first_child = [];
  next_sibling = []
}

(* Add node to columns *)
let add_node (cols:node_columns) (kind:node_kind) (sp:span) (ty:
   type_id)
             (parent:node_id) : node_columns & node_id =
  let id = List.length cols.kinds in
  let cols' = {
    kinds = cols.kinds @ [kind];
    spans = cols.spans @ [sp];
    types = cols.types @ [ty];
    parents = cols.parents @ [parent];
    first_child = cols.first_child @ [0];  (* No children yet *)
    next_sibling = cols.next_sibling @ [0]
  } in
  (cols', id)

(* Get node kind *)
let get_kind (cols:node_columns) (id:node_id) : option node_kind =
  if id < List.length cols.kinds then Some (List.nth cols.kinds id)
  else None
```

```
(* CSR edge storage *)
noeq type csr_edges = {
  row_ptr   : list nat;        (* Start index for each node's edges *)
  col_idx   : list node_id;    (* Target nodes *)
  edge_kind : list nat         (* Edge labels *)
}

(* Create CSR from edge list *)
let build_csr (num_nodes:nat) (edges:list (node_id & node_id & nat)) :
    csr_edges =
  (* Sort edges by source *)
  let sorted = List.sortBy (fun (s, _, _) → s) edges in

  (* Build row pointers *)
  let rec build_row_ptr (node:nat) (edges:list (node_id & node_id &
      nat)) (idx:nat) (acc:list nat) =
    if node > num_nodes then List.rev acc
    else
      (* Count edges from this node *)
      let from_node = List.filter (fun (s, _, _) → s = node) edges in
      build_row_ptr (node + 1) edges (idx + List.length from_node) (
         idx :: acc)
  in

  let row_ptr = build_row_ptr 0 sorted 0 [] in
  let col_idx = List.map (fun (_, t, _) → t) sorted in
  let edge_kind = List.map (fun (_, _, k) → k) sorted in

  { row_ptr = row_ptr; col_idx = col_idx; edge_kind = edge_kind }

(* Get successors of node *)
let get_successors (csr:csr_edges) (node:node_id) : list node_id =
  if node + 1 ≥ List.length csr.row_ptr then []
  else
    let start = List.nth csr.row_ptr node in
    let end_ = List.nth csr.row_ptr (node + 1) in
    List.slice csr.col_idx start end_

(* Iterate over edges (SIMD-friendly) *)
let fold_edges (#a:Type) (csr:csr_edges) (f:node_id → node_id → nat →
    a → a) (init:a) : a =
  let rec go (i:nat) (acc:a) =
    if i ≥ List.length csr.col_idx then acc
    else
      (* Find source node *)
      let src = List.find_index (fun p → p > i) csr.row_ptr in
      match src with
      | Some s → go (i + 1) (f (s - 1) (List.nth csr.col_idx i) (List.
         nth csr.edge_kind i) acc)
      | None → acc
  in go 0 init

(* Memory layout statistics *)
let column_memory_usage (cols:node_columns) : nat =
  let n = List.length cols.kinds in
  n * 1 +       (* kinds: 1 byte *)
  n * 12 +      (* spans: 12 bytes *)
  n * 4 +       (* types: 4 bytes *)
  n * 4 +       (* parents: 4 bytes *)
  n * 4 +       (* first_child: 4 bytes *)
  n * 4         (* next_sibling: 4 bytes *)
  (* Total: 33 bytes per node *)
```

41. String Interning

41.1 String Table

Definition 41.1 (Interned String)**.** All strings are stored once in a global table:

$$\mathsf{StringId} = \mathbb{N}_{32}$$

$$\mathsf{StringTable} : \mathsf{StringId} \rightharpoonup \mathsf{String}$$

Definition 41.2 (Interning Operation)**.**

$$\mathsf{intern} : \mathsf{String} \to \mathsf{StringId}$$

$$\mathsf{intern}(s) = \begin{cases} \mathsf{id} & \text{if } s \in \mathsf{table} \\ \mathsf{fresh}() & \text{otherwise, adding } s \text{ to table} \end{cases}$$

Theorem 41.3 (Interning Benefits)**.** • *O(1) string equality via ID comparison*

- *Reduced memory for repeated identifiers*
- *Better cache locality (IDs are 4 bytes)*

41.2 Type Hash-Consing

Definition 41.4 (Type Table)**.** Types are hash-consed similarly:

$$\mathsf{TypeId} = \mathbb{N}_{32}$$

$$\mathsf{TypeTable} : \mathsf{TypeId} \rightharpoonup \mathsf{Type}$$

Definition 41.5 (Type Hash-Consing)**.**

$$\mathsf{mk\_func}(\tau_1, \tau_2, \varepsilon) = \mathsf{intern\_type}(\mathsf{Func}(\tau_1, \tau_2, \varepsilon))$$
$$\mathsf{mk\_array}(\tau) = \mathsf{intern\_type}(\mathsf{Array}(\tau))$$

Structurally equal types have equal IDs.

41.3 F* Mechanization

String Interning in F*

```
module BrrrSemantics.Interning

(* String ID *)
type string_id = nat

(* String table *)
noeq type string_table = {
  strings    : list string;            (* ID → String *)
  lookup_map : list (string & string_id)   (* String → ID *)
}

(* Empty table with builtin strings *)
let empty_string_table : string_table = {
  strings = [""];   (* ID 0 = empty string *)
  lookup_map = [("", 0)]
}

(* Intern a string *)
let intern_string (table:string_table) (s:string) : string_table &
    string_id =
  match List.assoc s table.lookup_map with
  | Some id → (table, id)
  | None →
      let id = List.length table.strings in
      let table' = {
        strings = table.strings @ [s];
        lookup_map = (s, id) :: table.lookup_map
      } in
      (table', id)

(* Get string by ID *)
let get_string (table:string_table) (id:string_id) : option string =
  if id < List.length table.strings then Some (List.nth table.strings
      id)
  else None

(* String equality via ID *)
let string_eq (id1 id2:string_id) : bool = id1 = id2

(* Type ID *)
type type_id = nat

(* Type representation for hashing *)
noeq type type_repr =
  | TRPrim    : nat → type_repr                            (* Primitive
      type code *)
  | TRFunc    : type_id → type_id → effect_id → type_repr
  | TRArray   : type_id → type_repr
  | TROption  : type_id → type_repr
  | TRTuple   : list type_id → type_repr
  | TRNamed   : string_id → list type_id → type_repr     (* Named type
      with args *)

and effect_id = nat

(* Type table *)
noeq type type_table = {
  types      : list type_repr;
  lookup_map : list (type_repr & type_id)
}

(* Empty type table with primitives *)
let empty_type_table : type_table = {
  types = [
    TRPrim 0;   (* Unit *)
    TRPrim 1;   (* Bool *)
    TRPrim 2;   (* Int32 *)
    TRPrim 3;   (* Int64 *)
    TRPrim 4;   (* Float32 *)
    TRPrim 5;   (* Float64 *)
    TRPrim 6;   (* String *)
    TRPrim 7   (* Never *)
  ];
  lookup_map = []  (* Primitives found by code *)
}

(* Type ID constants *)
let type_unit   : type_id = 0
let type_bool   : type_id = 1
let type_i32    : type_id = 2
let type_i64    : type_id = 3
let type_f32    : type_id = 4
let type_f64    : type_id = 5
let type_string : type_id = 6
let type_never  : type_id = 7

(* Hash-cons a type *)
let intern_type (table:type_table) (t:type_repr) : type_table &
   type_id =
  match List.assoc t table.lookup_map with
  | Some id → (table, id)
  | None →
      let id = List.length table.types in
      let table' = {
        types = table.types @ [t];
        lookup_map = (t, id) :: table.lookup_map
      } in
      (table', id)

(* Make function type *)
let mk_func_type (table:type_table) (arg ret:type_id) (eff:effect_id)
    : type_table & type_id =
  intern_type table (TRFunc arg ret eff)

(* Make array type *)
let mk_array_type (table:type_table) (elem:type_id) : type_table &
   type_id =
  intern_type table (TRArray elem)

(* Type equality via ID *)
let type_eq (id1 id2:type_id) : bool = id1 = id2

(* Get type by ID *)
let get_type (table:type_table) (id:type_id) : option type_repr =
  if id < List.length table.types then Some (List.nth table.types id)
  else None
```

Part XII
Compute Architecture

42. Parallelization Strategy

42.1 Parallelism Levels

Definition 42.1 (Four-Level Parallelism). 1. **File-level**: Different files analyzed in parallel
2. **Function-level**: Independent functions analyzed in parallel
3. **Worklist-level**: Multiple worklist items processed in parallel
4. **SIMD-level**: Data-parallel operations on vectors

Definition 42.2 (Dependency Graph).

$$G_{\mathsf{dep}} = (V, E) \quad \text{where } (u, v) \in E \iff \text{analysis of } v \text{ requires } u$$

Theorem 42.3 (Parallelization Theorem). *Tasks $t_1, \ldots, t_n$ can execute in parallel iff there is no path between any t_i and t_j in G_{dep}.*

42.2 Work Stealing

Definition 42.4 (Work-Stealing Queue). Each worker has a deque of tasks:

- Push/pop from bottom (local operations)
- Steal from top (remote operations)

Definition 42.5 (Work-Stealing Algorithm). 1. Worker attempts to pop from own deque
2. If empty, randomly select victim and steal
3. If all empty, worker goes idle
4. New work wakes idle workers

42.3 F* Mechanization

Parallelization in F*

```
module BrrrSemantics.Parallel

(* Task ID *)
type task_id = nat

(* Task state *)
type task_state = | Pending | Running | Completed

(* Task with dependencies *)
noeq type task = {
  id           : task_id;
  dependencies : list task_id;
  state        : ref task_state;
  work         : unit → unit
}

(* Check if task is ready *)
let is_ready (t:task) (completed:list task_id) : bool =
  List.for_all (fun dep → List.mem dep completed) t.dependencies

(* Work-stealing deque *)
noeq type ws_deque (a:Type) = {
  items : ref (list a);
  lock  : ref bool   (* Simplified mutex *)
}
```

```
let new_deque (#a:Type) () : ws_deque a =
  { items = ref []; lock = ref false }

(* Push to bottom (local) *)
let push_bottom (#a:Type) (d:ws_deque a) (x:a) : unit =
  d.items := x :: !(d.items)

(* Pop from bottom (local) *)
let pop_bottom (#a:Type) (d:ws_deque a) : option a =
  match !(d.items) with
  | [] → None
  | x :: rest → d.items := rest; Some x

(* Steal from top (remote, needs sync) *)
let steal_top (#a:Type) (d:ws_deque a) : option a =
  (* Acquire lock *)
  if !(d.lock) then None
  else begin
    d.lock := true;
    let result = match List.rev !(d.items) with
      | [] → None
      | x :: rest → d.items := List.rev rest; Some x
    in
    d.lock := false;
    result
  end

(* Worker *)
noeq type worker = {
  id    : nat;
  deque : ws_deque task
}

(* Scheduler *)
noeq type scheduler = {
  workers   : list worker;
  completed : ref (list task_id)
}

(* Run worker *)
let rec worker_loop (w:worker) (all_workers:list worker) (completed:
   ref (list task_id)) : unit =
  (* Try own deque *)
  match pop_bottom w.deque with
  | Some t →
      if is_ready t !completed then begin
        t.state := Running;
        t.work ();
        t.state := Completed;
        completed := t.id :: !completed;
        worker_loop w all_workers completed
      end else begin
        (* Put back if not ready *)
        push_bottom w.deque t;
        worker_loop w all_workers completed
      end
  | None →
```

```
      (* Try stealing *)
      let victims = List.filter (fun w' → w'.id ≠ w.id) all_workers in
      match try_steal victims with
      | Some t → push_bottom w.deque t; worker_loop w all_workers
         completed
      | None → ()  (* Done or waiting *)

and try_steal (victims:list worker) : option task =
  match victims with
  | [] → None
  | v :: rest →
      match steal_top v.deque with
      | Some t → Some t
      | None → try_steal rest

(* Dependency graph analysis *)
let topological_levels (tasks:list task) : list (list task_id) =
  (* Group tasks by dependency depth *)
  let rec depth (t:task) (memo:list (task_id & nat)) : nat =
    match List.assoc t.id memo with
    | Some d → d
    | None →
        if t.dependencies = [] then 0
        else 1 + List.fold_left max 0
          (List.map (fun dep →
            match List.find (fun t' → t'.id = dep) tasks with
            | Some t' → depth t' memo
            | None → 0) t.dependencies)
  in
  let depths = List.map (fun t → (t.id, depth t [])) tasks in
  let max_depth = List.fold_left (fun m (_, d) → max m d) 0 depths in
  List.init (max_depth + 1) (fun level →
    List.filter_map (fun (id, d) → if d = level then Some id else None
       ) depths)

(* Parallel for each level *)
let parallel_execute (tasks:list task) : unit =
  let levels = topological_levels tasks in
  List.iter (fun level →
    (* Tasks in same level can run in parallel *)
    List.iter (fun id →
      match List.find (fun t → t.id = id) tasks with
      | Some t → t.work ()
      | None → ()
    ) level
  ) levels
```

43. SIMD Optimization

43.1 Vector Operations

Definition 43.1 (SIMD Width).

$$\begin{aligned} \mathsf{SSE} &: 128 \text{ bits} = 4 \times \mathsf{Float}[32] = 2 \times \mathsf{Float}[64] \\ \mathsf{AVX2} &: 256 \text{ bits} = 8 \times \mathsf{Float}[32] = 4 \times \mathsf{Float}[64] \\ \mathsf{AVX512} &: 512 \text{ bits} = 16 \times \mathsf{Float}[32] = 8 \times \mathsf{Float}[64] \end{aligned}$$

Definition 43.2 (Vector Type).

$$\mathsf{Vec}[n, \tau] \quad \text{where } n \cdot \mathsf{sizeof}(\tau) \leq \mathsf{SIMD\_WIDTH}$$

43.2 Bitset Operations

Definition 43.3 (SIMD Bitset). For dataflow facts represented as bitsets:

$$\mathsf{facts} : \mathsf{Vec}[512, \mathsf{Bool}] \cong \mathbb{Z}_{2^{512}}$$

Definition 43.4 (Vectorized Set Operations).

$$\begin{aligned}
\mathsf{union}(a, b) &= a \vee b && \text{(bitwise OR)} \\
\mathsf{intersect}(a, b) &= a \wedge b && \text{(bitwise AND)} \\
\mathsf{diff}(a, b) &= a \wedge \neg b && \text{(bitwise ANDNOT)} \\
\mathsf{is\_subset}(a, b) &= (a \wedge \neg b) = 0
\end{aligned}$$

Single instruction per 512 bits with AVX-512.

Theorem 43.5 (Dataflow Speedup). *For bitvector dataflow with n facts:*

- *Scalar:* $O(n)$ *operations per join*
- *SIMD (512-bit):* $O(n/512)$ *operations per join*
- *Speedup: up to* $512\times$ *for large fact sets*

43.3 F* Mechanization

SIMD Optimization in F*

```
module BrrrSemantics.SIMD

(* SIMD width in bits *)
type simd_width = | SSE128 | AVX256 | AVX512

let width_bits (w:simd_width) : nat =
  match w with
  | SSE128 → 128
  | AVX256 → 256
  | AVX512 → 512

(* Vector type *)
noeq type vec (n:nat) (a:Type) = {
  elements : l:list a{List.length l = n}
}

(* SIMD vector types *)
type vec4f32  = vec 4 float    (* SSE *)
type vec8f32  = vec 8 float    (* AVX2 *)
type vec16f32 = vec 16 float   (* AVX-512 *)

type vec2f64  = vec 2 float
type vec4f64  = vec 4 float
type vec8f64  = vec 8 float

(* Bitset as vector of 64-bit words *)
type bitset512 = vec 8 nat  (* 8 x 64 bits = 512 bits *)

(* Create empty bitset *)
let empty_bitset : bitset512 = { elements = [0;0;0;0;0;0;0;0] }
```

```
(* Create full bitset *)
let full_bitset : bitset512 = {
  elements = [0xFFFFFFFFFFFFFFFF; 0xFFFFFFFFFFFFFFFF;
              0xFFFFFFFFFFFFFFFF; 0xFFFFFFFFFFFFFFFF;
              0xFFFFFFFFFFFFFFFF; 0xFFFFFFFFFFFFFFFF;
              0xFFFFFFFFFFFFFFFF; 0xFFFFFFFFFFFFFFFF]
}

(* SIMD bitwise OR (union) *)
let simd_or (a b:bitset512) : bitset512 =
  { elements = List.map2 (fun x y → x lor y) a.elements b.elements }

(* SIMD bitwise AND (intersect) *)
let simd_and (a b:bitset512) : bitset512 =
  { elements = List.map2 (fun x y → x land y) a.elements b.elements }

(* SIMD bitwise ANDNOT (diff) *)
let simd_andnot (a b:bitset512) : bitset512 =
  { elements = List.map2 (fun x y → x land (lnot y)) a.elements b.
      elements }

(* SIMD bitwise XOR *)
let simd_xor (a b:bitset512) : bitset512 =
  { elements = List.map2 (fun x y → x lxor y) a.elements b.elements }

(* Check if zero (all bits unset) *)
let simd_is_zero (a:bitset512) : bool =
  List.for_all ((=) 0) a.elements

(* Check subset: a ⊆ b iff (a AND NOT b) = 0 *)
let simd_is_subset (a b:bitset512) : bool =
  simd_is_zero (simd_andnot a b)

(* Check equality *)
let simd_eq (a b:bitset512) : bool =
  simd_is_zero (simd_xor a b)

(* Count set bits (popcount) *)
let popcount64 (x:nat) : nat =
  (* Simplified - real impl uses hardware popcount *)
  let rec count x acc =
    if x = 0 then acc
    else count (x / 2) (acc + x mod 2)
  in count x 0

let simd_popcount (a:bitset512) : nat =
  List.fold_left (+) 0 (List.map popcount64 a.elements)

(* Set bit at position *)
let set_bit (a:bitset512) (pos:nat) : bitset512 =
  if pos ≥ 512 then a
  else
    let word_idx = pos / 64 in
    let bit_idx = pos mod 64 in
    { elements = List.mapi (fun i w →
        if i = word_idx then w lor (1 lsl bit_idx) else w) a.elements
          }
```

```
(* Test bit at position *)
let test_bit (a:bitset512) (pos:nat) : bool =
  if pos ≥ 512 then false
  else
    let word_idx = pos / 64 in
    let bit_idx = pos mod 64 in
    (List.nth a.elements word_idx land (1 lsl bit_idx)) ≠ 0

(* Vectorized dataflow iteration *)
let dataflow_step (in_facts:list bitset512) (gen kill:bitset512) :
   bitset512 =
  (* out = gen ∪ (in - kill) *)
  let combined_in = List.fold_left simd_or empty_bitset in_facts in
  simd_or gen (simd_andnot combined_in kill)
```

44. Worklist Algorithms

44.1 RPO Worklist

Definition 44.1 (Reverse Postorder (RPO))**.** Nodes in reverse postorder of the CFG. Processing in RPO ensures:

- Predecessors processed before successors (mostly)
- Faster convergence for forward analyses

Definition 44.2 (Worklist Algorithm)**.** 1. Initialize: worklist ← all nodes in RPO

2. While worklist non-empty:
   (a) Remove node n with smallest RPO number
   (b) Compute new facts for n
   (c) If facts changed, add successors to worklist
3. Return fixpoint

44.2 Lock-Free Fact Propagation

Definition 44.3 (Atomic Fact Update)**.** For concurrent worklist processing:

$$\mathsf{update}(\mathsf{facts}[n], \mathsf{new}) = \mathsf{CAS}(\mathsf{facts}[n], \mathsf{old}, \mathsf{old} \sqcup \mathsf{new})$$

where CAS is atomic compare-and-swap.

Theorem 44.4 (Monotonicity Ensures Correctness)**.** *If the transfer function is monotonic and the lattice has finite height, lock-free propagation converges to the same fixpoint as sequential.*

44.3 F* Mechanization

Worklist Algorithms in F*

```
module BrrrSemantics.Worklist

(* Priority queue based on RPO number *)
noeq type worklist = {
  items    : list (nat (* rpo *) & node_id);
  in_queue : node_id → bool
}

let empty_worklist : worklist = {
  items = [];
  in_queue = fun _ → false
}

(* Add node to worklist *)
let add_to_worklist (wl:worklist) (rpo:nat) (node:node_id) : worklist
    =
  if wl.in_queue node then wl
  else {
    items = insert_sorted (rpo, node) wl.items;
    in_queue = fun n → n = node || wl.in_queue n
  }

and insert_sorted (item:nat & node_id) (items:list (nat & node_id))
    : list (nat & node_id) =
  match items with
  | [] → [item]
  | (r, n) :: rest →
      if fst item ≤ r then item :: items
      else (r, n) :: insert_sorted item rest

(* Remove minimum from worklist *)
let pop_worklist (wl:worklist) : option (node_id & worklist) =
  match wl.items with
  | [] → None
  | (_, node) :: rest →
      Some (node, {
        items = rest;
        in_queue = fun n → n ≠ node && wl.in_queue n
      })

(* Dataflow lattice *)
type lattice (a:Type) = {
  bottom : a;
  join   : a → a → a;
  leq    : a → a → bool
}

(* Transfer function *)
type transfer (a:Type) = node_id → a → a

(* Dataflow analysis *)
let analyze (#a:Type) (lat:lattice a) (transfer:transfer a)
            (cfg:csr_edges) (rpo_order:list node_id) : node_id → a =
  let n_nodes = List.length rpo_order in
  let facts : ref (node_id → a) = ref (fun _ → lat.bottom) in
  let rpo_num = List.mapi (fun i n → (n, i)) rpo_order in

  (* Initialize worklist with all nodes *)
  let init_wl = List.fold_left (fun wl (node, rpo) →
    add_to_worklist wl rpo node) empty_worklist rpo_num in

  (* Main loop *)
  let rec iterate (wl:worklist) : unit =
    match pop_worklist wl with
    | None → ()  (* Converged *)
    | Some (node, wl') →
        let old_fact = !facts node in
        (* Gather input from predecessors *)
        let pred_facts = List.map (fun p → !facts p) (get_predecessors
            cfg node) in
        let in_fact = List.fold_left lat.join lat.bottom pred_facts in
        let new_fact = transfer node in_fact in

        if lat.leq new_fact old_fact then
          (* No change, continue *)
          iterate wl'
        else begin
          (* Update and propagate *)
          facts := (fun n → if n = node then lat.join old_fact
             new_fact else !facts n);
          let succs = get_successors cfg node in
          let wl'' = List.fold_left (fun w s →
            let rpo = match List.assoc s rpo_num with Some r → r |
               None → n_nodes in
            add_to_worklist w rpo s) wl' succs in
          iterate wl''
        end
  in

  iterate init_wl;
  !facts

and get_predecessors (cfg:csr_edges) (node:node_id) : list node_id =
  (* Would need reverse edges *)
  []

(* Lock-free fact update *)
type atomic_facts (a:Type) = {
  data : ref (node_id → a);
  (* In real impl: array of atomic cells *)
}

let atomic_update (#a:Type) (lat:lattice a) (facts:atomic_facts a)
                  (node:node_id) (new_fact:a) : bool (* changed *) =
  let old = !facts.data node in
  let joined = lat.join old new_fact in
  if lat.leq joined old then false
  else begin
    (* CAS: compare-and-swap *)
    facts.data := (fun n → if n = node then joined else !facts.data n)
       ;
    true
  end

(* Parallel worklist with lock-free facts *)
let parallel_analyze (#a:Type) (lat:lattice a) (transfer:transfer a)
                     (cfg:csr_edges) (rpo_order:list node_id) (
                        n_workers:nat)
                     : node_id → a =
  let facts : atomic_facts a = { data = ref (fun _ → lat.bottom) } in

  (* Each worker processes nodes from shared worklist *)
  (* Simplified: actual impl would use concurrent data structures *)

  !facts.data
```

45. Memory Hierarchy Optimization

45.1 Cache-Aware Layout

Definition 45.1 (Cache Line)**.** Modern CPUs fetch 64-byte cache lines. Data layout should:

- Align hot data to cache line boundaries
- Group frequently co-accessed data
- Separate read-only from read-write data

Definition 45.2 (Hot/Warm/Cold Separation)**.**

$$\begin{aligned}\mathsf{Hot} &: \text{Accessed in inner loops (node kinds, parents)}\\ \mathsf{Warm} &: \text{Accessed occasionally (types, spans)}\\ \mathsf{Cold} &: \text{Rarely accessed (debug info, comments)}\end{aligned}$$

45.2 Memory Budget

Definition 45.3 (Per-Node Memory)**.** Target: fit working set in L3 cache.

Component	Bytes per node
Node kind	1
Parent pointer	4
First child	4
Next sibling	4
Type ID	4
Span	12
Flags	1
Hot total	**30**
Source text offset	4
Hash (cached)	4
Warm total	**8**

Theorem 45.4 (L3 Capacity)**.** *With 30 bytes/node hot data and 32 MB L3 cache:*

$$\textit{Max hot nodes} = \frac{32 \times 10^6}{30} \approx 1{,}066{,}666 \textit{ nodes}$$

Sufficient for most single-file analyses.

45.3 Prefetching

Definition 45.5 (Software Prefetch)**.** For predictable access patterns:

$$\mathsf{prefetch}(\mathsf{addr} + k \cdot \mathsf{stride})$$

Fetch data k iterations ahead.

45.4 F* Mechanization

Memory Hierarchy in F*

```
module BrrrSemantics.MemoryHierarchy

(* Cache line size *)
let cache_line_bytes : nat = 64
```

```
(* L1/L2/L3 cache sizes (typical) *)
let l1_cache_bytes : nat = 32 * 1024       (* 32 KB *)
let l2_cache_bytes : nat = 256 * 1024      (* 256 KB *)
let l3_cache_bytes : nat = 32 * 1024 * 1024   (* 32 MB *)

(* Temperature classification *)
type temperature = | Hot | Warm | Cold

(* Hot node data (fits in 32 bytes, aligned to 32) *)
noeq type hot_node = {
  kind         : nat;       (* 1 byte, padded *)
  flags        : nat;       (* 1 byte *)
  _pad1        : nat;       (* 2 bytes padding *)
  parent       : node_id;   (* 4 bytes *)
  first_child  : node_id;   (* 4 bytes *)
  next_sibling: node_id;    (* 4 bytes *)
  type_id      : type_id;   (* 4 bytes *)
  span_start   : nat;       (* 4 bytes *)
  span_end     : nat;       (* 4 bytes *)
  span_file    : nat        (* 4 bytes *)
  (* Total: 32 bytes *)
}

(* Warm node data *)
noeq type warm_node = {
  source_offset : nat;      (* 4 bytes *)
  hash_cache    : nat       (* 4 bytes *)
  (* Total: 8 bytes *)
}

(* Cold node data *)
noeq type cold_node = {
  debug_name    : string_id;
  comment       : string_id;
  original_span : span
}

(* Separated storage *)
noeq type node_storage = {
  hot_nodes  : list hot_node;     (* Contiguous, cache-friendly *)
  warm_nodes : list warm_node;
  cold_nodes : list cold_node
}

(* Memory usage calculation *)
let hot_bytes_per_node : nat = 32
let warm_bytes_per_node : nat = 8
let cold_bytes_per_node : nat = 20  (* Approximate *)

let calculate_memory (n_nodes:nat) : nat & nat & nat =
  (n_nodes * hot_bytes_per_node,
   n_nodes * warm_bytes_per_node,
   n_nodes * cold_bytes_per_node)

(* Can fit in L3? *)
let fits_in_l3 (n_nodes:nat) : bool =
  n_nodes * hot_bytes_per_node ≤ l3_cache_bytes

(* Max nodes for L3 *)
let max_l3_nodes : nat = l3_cache_bytes / hot_bytes_per_node

(* Prefetch hint *)
type prefetch_hint = | PrefetchRead | PrefetchWrite | PrefetchNTA

let prefetch_distance : nat = 8  (* Prefetch 8 iterations ahead *)

(* Prefetch in traversal *)
let traverse_with_prefetch (nodes:list hot_node) (f:hot_node → unit) :
    unit =
  let arr = nodes in  (* Assume array-like access *)
  let len = List.length arr in
  List.iteri (fun i node →
    (* Prefetch ahead *)
    if i + prefetch_distance < len then
      let _ = List.nth arr (i + prefetch_distance) in ()  (* Touch to
          prefetch *)
    else ();
    f node
  ) arr

(* Cache line alignment *)
let align_to_cache_line (addr:nat) : nat =
  let rem = addr mod cache_line_bytes in
  if rem = 0 then addr else addr + (cache_line_bytes - rem)

(* Pack multiple small items into cache line *)
let items_per_cache_line (item_size:nat) : nat =
  cache_line_bytes / item_size

(* For 32-byte hot nodes: 2 per cache line *)
let hot_nodes_per_cache_line : nat = items_per_cache_line
   hot_bytes_per_node
```

A. Notation Summary

Symbol	Meaning	Definition
$[\![e]\!]$	Denotation of e	Chapter 1
$\tau@m$	Type with mode	Chapter 7
$\sqcup$	Effect join	Chapter 4
$\sqsubseteq$	Effect subtyping	Chapter 4
$\langle E \mid \varepsilon\rangle$	Effect row extension	Chapter 4
$\bot$	Pure effect (bottom)	Chapter 4
$\otimes$	Tensor product (pair)	Chapter 10
$\multimap$	Linear function	Chapter 10
$!\tau$	Exponential (unrestricted)	Chapter 8
$\Box\tau$	Box (shared borrow)	Chapter 9
$\Diamond\tau$	Diamond (exclusive)	Chapter 9
ρ	Region/lifetime	Chapter 10

B. Complete Typing Rules

This appendix collects all typing rules for reference.

B.1 Core Rules

$$\frac{(x : \tau@m) \in \Gamma \qquad m > 0}{\Gamma \vdash x : \tau\ [\bot]}\ \textsc{T-Var} \qquad \frac{\Gamma, x : \tau_1@m \vdash e : \tau_2\ [\varepsilon]}{\Gamma \vdash \lambda x.\, e : (\tau_1@m) \xrightarrow{\varepsilon} \tau_2\ [\bot]}\ \textsc{T-Abs}$$

$$\frac{\Gamma_1 \vdash e_1 : (\tau_1@m) \xrightarrow{\varepsilon_f} \tau_2\ [\varepsilon_1] \qquad \Gamma_2 \vdash e_2 : \tau_1\ [\varepsilon_2]}{\Gamma_1 + \Gamma_2 \vdash e_1\ e_2 : \tau_2\ [\varepsilon_1 \sqcup \varepsilon_2 \sqcup \varepsilon_f]}\ \textsc{T-App}$$

$$\frac{\Gamma_1 \vdash e_1 : \tau_1\ [\varepsilon_1] \qquad \Gamma_2, x : \tau_1@m \vdash e_2 : \tau_2\ [\varepsilon_2]}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\, x = e_1\ \mathsf{in}\ e_2 : \tau_2\ [\varepsilon_1 \sqcup \varepsilon_2]}\ \textsc{T-Let}$$

$$\frac{\Gamma_0 \vdash e_c : \mathsf{Bool}\ [\varepsilon_0] \qquad \Gamma_1 \vdash e_1 : \tau\ [\varepsilon_1] \qquad \Gamma_2 \vdash e_2 : \tau\ [\varepsilon_2]}{\Gamma_0 + \Gamma_1 + \Gamma_2 \vdash \mathsf{if}\, e_c\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : \tau\ [\varepsilon_0 \sqcup \varepsilon_1 \sqcup \varepsilon_2]}\ \textsc{T-If}$$

B.2 Subtyping Rules

$$\frac{\Gamma \vdash e : \tau_1\ [\varepsilon_1] \qquad \tau_1 <: \tau_2 \qquad \varepsilon_1 \sqsubseteq \varepsilon_2}{\Gamma \vdash e : \tau_2\ [\varepsilon_2]}\ \textsc{T-Sub}$$

B.3 Effect Rules

$$\frac{\Gamma \vdash e : \tau\ [\langle E \mid \varepsilon\rangle] \qquad \Gamma, x : \tau \vdash e_r : \sigma\ [\varepsilon'] \qquad \forall i.\ \Gamma, x_i : \tau_i, k : \sigma_i \to \sigma\ [\varepsilon'] \vdash e_i : \sigma\ [\varepsilon']}{\Gamma \vdash \mathsf{handle}\, e\, \mathsf{with}\, H : \sigma\ [\varepsilon \sqcup \varepsilon']}\ \textsc{T-Handle}$$

B.4 Ownership Rules

$$\frac{\Gamma, x : \tau@1 \vdash e : \sigma\ [\varepsilon]}{\Gamma \vdash \mathsf{move}(x) : \tau\ [\varepsilon]}\ \textsc{T-Move} \qquad \frac{\Gamma \vdash e : \tau@1\ [\varepsilon]}{\Gamma \vdash \&e : \Box\tau\ [\varepsilon]}\ \textsc{T-Borrow}$$

$$\frac{\Gamma \vdash e : \tau@1\ [\varepsilon] \qquad \text{no active borrows}}{\Gamma \vdash \&\mathsf{mut}\, e : \Diamond\tau\ [\varepsilon]}\ \textsc{T-BorrowMut}$$

Part XIII
Incremental Analysis

C. Merkle-Based Change Detection

Incremental analysis requires efficient detection of what has changed. We use content-addressed hashing to create Merkle trees over the program structure.

C.1 Content Hashing

Definition C.1 (Node Hash)**.** For any AST node n, the **content hash** is:

$$\mathsf{hash}(n) = H(\mathsf{kind}(n) \parallel \mathsf{hash}(\mathsf{children}(n)) \parallel \mathsf{content}(n))$$

where H is BLAKE3 and $\parallel$ denotes concatenation.

Definition C.2 (Merkle Root)**.** The **Merkle root** of a module M is:

$$\mathsf{root}(M) = H\left(\bigoplus_{n \in \mathsf{toplevel}(M)} \mathsf{hash}(n)\right)$$

Theorem C.3 (Change Detection)**.** *For modules M_1, M_2:*

$$\mathsf{root}(M_1) = \mathsf{root}(M_2) \iff M_1 \equiv_\alpha M_2$$

where $\equiv_\alpha$ is α-equivalence (up to variable renaming).

Merkle Tree Implementation

```
module Merkle

open FStar.Bytes
open FStar.Seq

type hash = bytes_32

val hash_node : node_kind → seq hash → bytes → hash
let hash_node kind children content =
  let kind_bytes = encode_kind kind in
  let children_hash = fold_left xor_hash zero_hash children in
  blake3 (kind_bytes @| children_hash @| content)

type merkle_tree = {
  root: hash;
  nodes: map hash node;
  children: map hash (seq hash);
}

val diff : merkle_tree → merkle_tree → set hash
let diff old_tree new_tree =
  let rec find_changed h =
    if not (Map.contains old_tree.nodes h) then
      singleton h
    else if Map.find old_tree.nodes h ≠ Map.find new_tree.nodes h then
      singleton h
    else
      fold_union find_changed (Map.find new_tree.children h)
  in
  find_changed new_tree.root

lemma diff_sound (t1 t2 : merkle_tree) (h : hash) :
  Lemma (requires h `Set.mem` diff t1 t2)
```

```
        (ensures node_changed t1 t2 h)
```

C.2 Differential Algorithm

Definition C.4 (Minimal Edit Script). Given trees T_1, T_2, the **minimal edit script** is:

$$\mathsf{edit}(T_1, T_2) =_{s \in \mathsf{Scripts}} |\{e \in s \mid e = \mathsf{Modify}(\cdot)\}|$$

subject to $\mathsf{apply}(s, T_1) = T_2$.

Theorem C.5 (Edit Distance Bound). *For trees with n_1, n_2 nodes:*

$$|\mathsf{edit}(T_1, T_2)| \leq \min(n_1, n_2) + |n_1 - n_2|$$

The algorithm proceeds top-down:

1. Compare roots: if hashes match, subtrees are identical
2. If hashes differ, recursively compare children
3. Use Hungarian algorithm for optimal child matching
4. Generate Insert/Delete/Modify operations

Tree Diff Algorithm

```
type edit_op =
  | Insert of hash * node
  | Delete of hash
  | Modify of hash * node * node
  | Move of hash * hash * hash   (* node, old_parent, new_parent *)

val tree_diff : merkle_tree → merkle_tree → list edit_op
let rec tree_diff t1 t2 =
  if t1.root = t2.root then []
  else
    let c1 = Map.find t1.children t1.root in
    let c2 = Map.find t2.children t2.root in
    let matching = hungarian_match c1 c2 hash_similarity in
    let ops = concat_map (fun (h1, h2) →
      if h1 = h2 then []
      else if is_none h1 then [Insert (h2, Map.find t2.nodes h2)]
      else if is_none h2 then [Delete h1]
      else tree_diff (subtree t1 h1) (subtree t2 h2)
    ) matching in
    ops
```

D. Dependency Tracking

Analysis results depend on program fragments. We track these dependencies to enable precise invalidation.

D.1 Dependency Graph

Definition D.1 (Dependency Relation). The **dependency relation** is:

$$fg \iff \text{analysis of } f \text{ reads result of } g$$

Definition D.2 (Transitive Closure). The **affected set** for change Δ is:

$$\mathsf{affected}(\Delta) = \{f \mid \exists g \in \Delta.\ f^*g\}$$

Theorem D.3 (Soundness of Invalidation). *If* $\mathsf{invalidate}(\Delta) \supseteq \mathsf{affected}(\Delta)$, *then recomputing invalidated results produces correct analysis.*

Dependency Tracking
```
module Dependencies

type dep_kind =
  | TypeDep       (* Type of symbol *)
  | CallDep       (* Function called *)
  | FieldDep      (* Field accessed *)
  | ImportDep     (* Module imported *)
  | InheritDep    (* Class inherited *)

type dependency = {
  source: symbol_id;
  target: symbol_id;
  kind: dep_kind;
  strength: float;  (* For prioritization *)
}

type dep_graph = {
  forward: map symbol_id (set dependency);
  backward: map symbol_id (set dependency);
}

val add_dependency : dep_graph → dependency → dep_graph
let add_dependency g d =
  { forward = Map.update g.forward d.source (Set.add d);
    backward = Map.update g.backward d.target (Set.add d) }

val affected_by : dep_graph → set symbol_id → set symbol_id
let rec affected_by g changed =
  let direct = fold_union (fun s →
    Set.map (fun d → d.source) (Map.find_default g.backward s Set.
       empty)
  ) changed in
  if Set.is_empty direct then changed
  else Set.union changed (affected_by g direct)

lemma affected_complete (g : dep_graph) (c : set symbol_id) (s :
   symbol_id) :
  Lemma (requires depends_transitively g s c)
        (ensures s `Set.mem` affected_by g c)
```

D.2 Fine-Grained Dependencies

We track dependencies at multiple granularities:

Definition D.4 (Granularity Levels)**.**

$$\begin{aligned}\textsf{Module} &: \text{Entire module changed}\\ \textsf{Symbol} &: \text{Specific function/class changed}\\ \textsf{Signature} &: \text{Only type signature changed}\\ \textsf{Body} &: \text{Only implementation changed}\end{aligned}$$

Theorem D.5 (Signature Stability)**.** *If only the body of f changes (not its signature), then:*

$$\forall g.\ g_{\textsf{Signature}} f \implies g \textit{ need not be reanalyzed}$$

Fine-Grained Invalidation
```
type change_kind =
  | SignatureChange of symbol_id * typ * typ
  | BodyChange of symbol_id
  | AddSymbol of symbol_id
  | RemoveSymbol of symbol_id

val minimal_invalidation : dep_graph → change_kind → set symbol_id
let minimal_invalidation g change =
  match change with
  | SignatureChange (s, old_ty, new_ty) →
      (* All dependents must be reanalyzed *)
      affected_by g (singleton s)
  | BodyChange s →
      (* Only callers with body-level deps *)
      let deps = Map.find_default g.backward s Set.empty in
      Set.filter (fun d → d.kind = CallDep && is_inlined d) deps
      |> Set.map (fun d → d.source)
  | AddSymbol s →
      (* New symbol, check for shadowing *)
      potential_shadowees g s
  | RemoveSymbol s →
      (* All references now invalid *)
      affected_by g (singleton s)
```

E. Memoization and Caching

Analysis results are cached and reused when inputs haven't changed.

E.1 Content-Addressed Cache

Definition E.1 (Cache Key)**.** For analysis A on input x:

$$\mathsf{key}(A, x) = H(A.\mathsf{version} \,\|\, \mathsf{hash}(x) \,\|\, \mathsf{hash}(\mathsf{deps}(A, x)))$$

Definition E.2 (Cache Validity)**.** Cache entry (k, v) is **valid** iff:

$$\forall x.\ \mathsf{key}(A, x) = k \implies A(x) = v$$

Theorem E.3 (Cache Soundness)**.** *If the cache is content-addressed and the hash function is collision-resistant, then cache lookups always return correct results.*

Memoization Cache

```
module Cache

type cache_entry (a : Type) = {
  key: hash;
  value: a;
  deps: set hash;
  created: timestamp;
  access_count: nat;
}

type cache (a : Type) = {
  entries: map hash (cache_entry a);
  max_size: nat;
  eviction_policy: eviction_policy;
}

val lookup : cache a → hash → option a
let lookup c k =
  match Map.find c.entries k with
  | Some e →
```

```
      (* Validate deps still match *)
      if all_deps_valid e.deps then Some e.value
      else None
  | None → None

val insert : cache a → hash → a → set hash → cache a
let insert c k v deps =
  let entry = { key = k; value = v; deps = deps;
                created = now(); access_count = 0 } in
  let c' = maybe_evict c in
  { c' with entries = Map.add c'.entries k entry }

val invalidate : cache a → set hash → cache a
let invalidate c changed =
  let dominated = find_dominated c.entries changed in
  { c with entries = Map.remove_all c.entries dominated }

(* LRU eviction with dependency awareness *)
val maybe_evict : cache a → cache a
let maybe_evict c =
  if Map.size c.entries < c.max_size then c
  else
    let victim = find_lru_no_dependents c in
    { c with entries = Map.remove c.entries victim.key }
```

E.2 Incremental Fixpoint

For analyses that compute fixpoints (e.g., dataflow), we support incremental recomputation.

Definition E.4 (Incremental Worklist)**.** Given change set Δ, the initial worklist is:

$$W_0 = \mathsf{affected}(\Delta)$$

Theorem E.5 (Incremental Correctness)**.** *Let F be a monotone analysis. If:*

1. $W_0 \supseteq \mathsf{affected}(\Delta)$

2. Each iteration adds newly-affected nodes to worklist

Then the incremental fixpoint equals the from-scratch fixpoint.

Incremental Fixpoint

```
val incremental_fixpoint :
  analysis_fn → dep_graph → cache result → set symbol_id →
  (cache result * map symbol_id result)
let incremental_fixpoint analyze deps cache changed =
  let worklist = ref (affected_by deps changed) in
  let results = ref (cache_to_map cache) in

  while not (Set.is_empty !worklist) do
    let node = Set.choose !worklist in
    worklist := Set.remove !worklist node;

    let old_result = Map.find_default !results node bottom in
    let new_result = analyze node !results in

    if new_result ≠ old_result then begin
      results := Map.add !results node new_result;
      (* Add dependents to worklist *)
      let dependents = Map.find_default deps.backward node Set.empty
        in
      worklist := Set.union !worklist (Set.map (fun d → d.source)
         dependents)
    end
  done;

  (map_to_cache !results, !results)

lemma incremental_correct (f : analysis_fn) (d : dep_graph) (c : cache
   ) (ch : set symbol_id) :
  Lemma (let (_, r) = incremental_fixpoint f d c ch in
         r = from_scratch_fixpoint f)
```

Part XIV

Source Language Mapping

This part defines translation functors from source languages to Brrr-Lang, with soundness guarantees.

F. Translation Functor Framework

F.1 Categorical Foundation

Each source language L forms a category:

Definition F.1 (Language Category)**.**

$$\mathbf{Cat}_L = (\mathsf{Obj}_L, \mathsf{Mor}_L, \mathsf{id}, \circ)$$

where:

- Obj_L = types in L
- Mor_L = functions $f : A \to B$ in L
- id_A = identity function
- $\circ$ = function composition

Definition F.2 (Translation Functor)**.** A **translation functor** $T : \mathbf{Cat}_L \to \mathbf{Cat}_{\mathsf{Brrr}}$ consists of:

- Object mapping: $T(\tau) \in \mathsf{Types}_{\mathsf{Brrr}}$
- Morphism mapping: $T(f : A \to B) : T(A) \to T(B)$

satisfying:

$$T(\mathsf{id}_A) = \mathsf{id}_{T(A)} \qquad \text{(Identity)}$$
$$T(g \circ f) = T(g) \circ T(f) \qquad \text{(Composition)}$$

Theorem F.3 (Soundness)**.** *A translation T is* ***sound*** *iff:*

$$\forall e, \rho.\ [\![e]\!]_L(\rho) = v \implies [\![T(e)]\!]_{\mathsf{Brrr}}(T(\rho)) = T(v)$$

Translation Functor Interface

```
module TranslationFunctor

type functor (l : language) = {
  (* Object mapping *)
  translate_type : l.typ → brrr_typ;

  (* Morphism mapping *)
  translate_expr : l.expr → brrr_expr;
  translate_stmt : l.stmt → brrr_stmt;

  (* Environment mapping *)
  translate_env : l.env → brrr_env;

  (* Value mapping *)
  translate_value : l.value → brrr_value;
}

val functor_laws : functor l → prop
let functor_laws f =
  (* Identity preservation *)
  (∀ a. f.translate_expr (l.id a) = brrr.id (f.translate_type a)) ∧
  (* Composition preservation *)
  (∀ g h. f.translate_expr (l.compose g h) =
              brrr.compose (f.translate_expr g) (f.translate_expr h))
```

```
val soundness : functor l → prop
let soundness f =
  ∀ e ρ v.
    l.eval e ρ = v ⟹
    brrr.eval (f.translate_expr e) (f.translate_env ρ) = f.
       translate_value v
```

G. Python Translation

G.1 Type Mapping

Definition G.1 (Python Type Translation)**.**

$$
\begin{aligned}
T_{\mathsf{Py}}(\texttt{None}) &= \mathsf{Unit} \\
T_{\mathsf{Py}}(\texttt{bool}) &= \mathsf{Bool} \\
T_{\mathsf{Py}}(\texttt{int}) &= \mathsf{Int}[\mathsf{BigInt}, \mathsf{Signed}] \\
T_{\mathsf{Py}}(\texttt{float}) &= \mathsf{Float}[\mathsf{F64}] \\
T_{\mathsf{Py}}(\texttt{str}) &= \mathsf{String} \\
T_{\mathsf{Py}}(\texttt{list}[A]) &= \mathsf{gc}\,\mathsf{Array}[[]T_{\mathsf{Py}}(A)] \\
T_{\mathsf{Py}}(\texttt{dict}[K, V]) &= \mathsf{gc}\,[T_{\mathsf{Py}}(K), T_{\mathsf{Py}}(V)] \\
T_{\mathsf{Py}}(\texttt{Callable}[[A_1, \ldots, A_n], R]) &= (T_{\mathsf{Py}}(A_1), \ldots) \xrightarrow{\varepsilon_{\mathsf{Py}}} T_{\mathsf{Py}}(R) \\
T_{\mathsf{Py}}(\texttt{Any}) &= \mathsf{Any}
\end{aligned}
$$

where $\varepsilon_{\mathsf{Py}} = \langle |\, \langle | \rangle \rangle$ (Python's default effects).

G.2 Expression Translation

Definition G.2 (Python Expression Translation)**.**

$$
\begin{aligned}
T_{\mathsf{Py}}(x) &= x \\
T_{\mathsf{Py}}(\texttt{lambda}\, x : e) &= \lambda x. T_{\mathsf{Py}}(e) \\
T_{\mathsf{Py}}(e_1(e_2)) &= T_{\mathsf{Py}}(e_1)\, T_{\mathsf{Py}}(e_2) \\
T_{\mathsf{Py}}(e_1 + e_2) &= T_{\mathsf{Py}}(e_1) + T_{\mathsf{Py}}(e_2) \\
T_{\mathsf{Py}}([e_1, \ldots, e_n]) &= \mathsf{gc\_alloc}([T_{\mathsf{Py}}(e_1), \ldots, T_{\mathsf{Py}}(e_n)]) \\
T_{\mathsf{Py}}(e.attr) &= \mathsf{attr\_get}(T_{\mathsf{Py}}(e), attr) \\
T_{\mathsf{Py}}(\texttt{raise}\, e) &= (T_{\mathsf{Py}}(e))
\end{aligned}
$$

Python Translation

```
module Python

let translate_type (t : py_type) : brrr_type =
  match t with
  | PyNone → TUnit
  | PyBool → TBool
  | PyInt → TInt BigInt Signed
  | PyFloat → TFloat F64
  | PyStr → TString
  | PyList a → TArray (translate_type a) GC
  | PyDict (k, v) → TDict (translate_type k) (translate_type v) GC
  | PyCallable (args, ret) →
      TFunc (List.map translate_type args) (translate_type ret)
         py_effects
  | PyAny → TAny
  | PyClass name → TClass name GC
```

```
let py_effects = RowExtend Throw (RowExtend IO RowVar)

let rec translate_expr (e : py_expr) : brrr_expr =
  match e with
  | PyVar x → EVar x
  | PyLambda (x, body) → ELambda (x, translate_expr body)
  | PyCall (f, args) → EApp (translate_expr f) (List.map
     translate_expr args)
  | PyBinOp (op, l, r) → EBinOp (translate_op op) (translate_expr l) (
     translate_expr r)
  | PyList elems → EAlloc GC (EArray (List.map translate_expr elems))
  | PyDict pairs → EAlloc GC (EDict (List.map (fun (k,v) →
      (translate_expr k, translate_expr v)) pairs))
  | PyAttr (obj, attr) → EFieldGet (translate_expr obj) attr
  | PyRaise exc → EThrow (translate_expr exc)
  | PyTry (body, handlers, finally_) →
      let brrr_body = translate_expr body in
      let brrr_handlers = List.map (fun (exc_ty, var, handler) →
        (translate_type exc_ty, var, translate_expr handler)
      ) handlers in
      let brrr_finally = Option.map translate_expr finally_ in
      ETryCatch brrr_body brrr_handlers brrr_finally

lemma python_translation_sound (e : py_expr) (env : py_env) :
  Lemma (ensures
    ∀ v. py_eval e env = v ⟹
    brrr_eval (translate_expr e) (translate_env env) = translate_value
        v)
```

H. TypeScript Translation

H.1 Type Mapping

Definition H.1 (TypeScript Type Translation).

$$\begin{aligned}
T_{\mathsf{TS}}(\texttt{undefined}) &= \mathsf{Unit}\\
T_{\mathsf{TS}}(\texttt{null}) &= \mathsf{Option}[[]\mathsf{Any}]\\
T_{\mathsf{TS}}(\texttt{boolean}) &= \mathsf{Bool}\\
T_{\mathsf{TS}}(\texttt{number}) &= \mathsf{Float}[\mathsf{F64}]\\
T_{\mathsf{TS}}(\texttt{bigint}) &= \mathsf{Int}[\mathsf{BigInt}, \mathsf{Signed}]\\
T_{\mathsf{TS}}(\texttt{string}) &= \mathsf{String}\\
T_{\mathsf{TS}}(A[]) &= \mathsf{gc}\,\mathsf{Array}[[]T_{\mathsf{TS}}(A)]\\
T_{\mathsf{TS}}(\texttt{Promise}\langle A\rangle) &= [T_{\mathsf{TS}}(A), \mathsf{Hot}]\\
T_{\mathsf{TS}}(A \mid B) &= T_{\mathsf{TS}}(A) \sqcup T_{\mathsf{TS}}(B)\\
T_{\mathsf{TS}}(A \,\&\, B) &= T_{\mathsf{TS}}(A) \sqcap T_{\mathsf{TS}}(B)
\end{aligned}$$

H.2 Async/Await Translation

Definition H.2 (Async Translation).

$$\begin{aligned}
T_{\mathsf{TS}}(\mathsf{async}\, f) &= f : \tau \to [\sigma]\\
T_{\mathsf{TS}}(\mathsf{await}\, e) &= \mathsf{await}(T_{\mathsf{TS}}(e))
\end{aligned}$$

TypeScript Translation

```
module TypeScript

let rec translate_type (t : ts_type) : brrr_type =
  match t with
  | TSUndefined → TUnit
  | TSNull → TOption TAny
  | TSBoolean → TBool
  | TSNumber → TFloat F64
  | TSBigInt → TInt BigInt Signed
  | TSString → TString
  | TSArray a → TArray (translate_type a) GC
  | TSPromise a → TFuture (translate_type a) Hot
  | TSUnion (a, b) → TUnion (translate_type a) (translate_type b)
  | TSIntersection (a, b) → TIntersection (translate_type a) (
     translate_type b)
  | TSObject fields → TStruct (List.map (fun (n, t) → (n,
     translate_type t)) fields) GC
  | TSFunction (params, ret) →
      TFunc (List.map translate_type params) (translate_type ret)
         ts_effects

let ts_effects = RowExtend Throw (RowExtend Async RowVar)

let rec translate_expr (e : ts_expr) : brrr_expr =
  match e with
  | TSVar x → EVar x
  | TSArrow (params, body) →
      ELambda (List.map fst params) (translate_expr body)
  | TSCall (f, args) →
      EApp (translate_expr f) (List.map translate_expr args)
  | TSAwait e → EAwait (translate_expr e)
  | TSAsync body →
      EAsync (translate_expr body)
  | TSOptionalChain (obj, prop) →
      EMatch (translate_expr obj) [
        (PNone, EConst CUnit);
        (PSome "v", EFieldGet (EVar "v") prop)
      ]
  | TSNullishCoalesce (l, r) →
      EMatch (translate_expr l) [
        (PNone, translate_expr r);
        (PSome "v", EVar "v")
      ]
  | TSTypeAssertion (e, ty) →
      ECast (translate_expr e) (translate_type ty)
```

I. Rust Translation

I.1 Ownership Mapping

Definition I.1 (Rust Ownership Translation).

$$\begin{aligned}
T_{\mathsf{Rs}}(T) &= \mathsf{own}\, T_{\mathsf{Rs}}^{\mathsf{base}}(T) \\
T_{\mathsf{Rs}}(\&'aT) &= T_{\mathsf{Rs}}^{\mathsf{base}}(T)\,['a] \\
T_{\mathsf{Rs}}(\&'a\mathtt{mut}\,T) &= T_{\mathsf{Rs}}^{\mathsf{base}}(T)\,['a] \\
T_{\mathsf{Rs}}(\mathtt{Box}\langle T\rangle) &= \mathsf{own}\,[T_{\mathsf{Rs}}^{\mathsf{base}}(T)] \\
T_{\mathsf{Rs}}(\mathtt{Rc}\langle T\rangle) &= \mathsf{rc}\, T_{\mathsf{Rs}}^{\mathsf{base}}(T) \\
T_{\mathsf{Rs}}(\mathtt{Arc}\langle T\rangle) &= \mathsf{arc}\, T_{\mathsf{Rs}}^{\mathsf{base}}(T)
\end{aligned}$$

I.2 Move Semantics

Definition I.2 (Move Translation).

$$\begin{aligned}
T_{\mathsf{Rs}}(\mathtt{let}\,y = x) &= \mathtt{let}\,y = \mathsf{move}(x) \quad (\text{if } x : T,\ T \text{ not Copy}) \\
T_{\mathsf{Rs}}(f(x)) &= f(\mathsf{move}(x)) \quad (\text{by value}) \\
T_{\mathsf{Rs}}(f(\&x)) &= f(\&x) \quad (\text{shared borrow}) \\
T_{\mathsf{Rs}}(f(\&\mathtt{mut}\,x)) &= f(\&\mathtt{mut}\,x) \quad (\text{exclusive borrow})
\end{aligned}$$

Theorem I.3 (Ownership Preservation). *If Rust program P is ownership-safe, then $T_{\mathsf{Rs}}(P)$ is ownership-safe in Brrr.*

Rust Translation

```
module Rust

type ownership_mode = Own | Ref of lifetime | RefMut of lifetime | Rc
    | Arc

let rec translate_type (t : rust_type) : brrr_type * ownership_mode =
  match t with
  | RsOwned base → (translate_base base, Own)
  | RsRef (lt, base) → (translate_base base, Ref lt)
  | RsRefMut (lt, base) → (translate_base base, RefMut lt)
  | RsBox base → (TBox (translate_base base), Own)
  | RsRc base → (translate_base base, Rc)
  | RsArc base → (translate_base base, Arc)
  | RsOption t →
      let (bt, mode) = translate_type t in
      (TOption bt, mode)
  | RsResult (t, e) →
      let (bt, _) = translate_type t in
      let (be, _) = translate_type e in
      (TResult bt be, Own)

and translate_base (b : rust_base_type) : brrr_type =
  match b with
  | RsBool → TBool
  | RsI8 → TInt I8 Signed
  | RsI16 → TInt I16 Signed
  | RsI32 → TInt I32 Signed
  | RsI64 → TInt I64 Signed
  | RsU8 → TInt I8 Unsigned
  | RsU64 → TInt I64 Unsigned
  | RsF32 → TFloat F32
  | RsF64 → TFloat F64
  | RsStr → TString
  | RsVec t → TArray (fst (translate_type t))
  | RsStruct (name, fields) →
      TStruct (List.map (fun (n, t) → (n, fst (translate_type t)))
         fields)

let rec translate_expr (e : rust_expr) (ctx : borrow_context) :
    brrr_expr =
  match e with
  | RsVar x →
      if is_copy ctx x then EVar x
      else if is_moved ctx x then error "use after move"
      else EMove (EVar x)
  | RsBorrow x → EBorrow (EVar x) Shared
  | RsBorrowMut x → EBorrow (EVar x) Exclusive
  | RsDeref e → EDeref (translate_expr e ctx)
  | RsMatch (scrut, arms) →
      EMatch (translate_expr scrut ctx)
        (List.map (fun (pat, body) →
          (translate_pattern pat, translate_expr body (extend_ctx ctx
             pat))
        ) arms)
  | RsBlock stmts →
      let (ctx', brrr_stmts) = translate_stmts stmts ctx in
      EBlock brrr_stmts
  | RsDrop x →
      ESeq (ECall (EVar "drop") [EMove (EVar x)]) (EConst CUnit)

lemma rust_ownership_preserved (e : rust_expr) :
  Lemma (requires rust_borrow_check e)
        (ensures brrr_ownership_safe (translate_expr e empty_ctx))
```

J. Go Translation

J.1 Type Mapping

Definition J.1 (Go Type Translation).

$$\begin{aligned}
T_{\mathsf{Go}}(\texttt{bool}) &= \mathsf{Bool} \\
T_{\mathsf{Go}}(\texttt{int}) &= \mathsf{Int}[\mathsf{I64}, \mathsf{Signed}] \\
T_{\mathsf{Go}}(\texttt{int32}) &= \mathsf{Int}[\mathsf{I32}, \mathsf{Signed}] \\
T_{\mathsf{Go}}(\texttt{float64}) &= \mathsf{Float}[\mathsf{F64}] \\
T_{\mathsf{Go}}(\texttt{string}) &= \mathsf{String} \\
T_{\mathsf{Go}}([]A) &= \mathsf{gc}\,[T_{\mathsf{Go}}(A)] \\
T_{\mathsf{Go}}(\texttt{map}[K]V) &= \mathsf{gc}\,[T_{\mathsf{Go}}(K), T_{\mathsf{Go}}(V)] \\
T_{\mathsf{Go}}(\texttt{chan}\ A) &= [T_{\mathsf{Go}}(A)] \\
T_{\mathsf{Go}}(*A) &= \mathsf{gc}\,[T_{\mathsf{Go}}(A)] \\
T_{\mathsf{Go}}(\texttt{interface}\{\ldots\}) &= [\mathsf{methods}]
\end{aligned}$$

J.2 Goroutine Translation

Definition J.2 (Concurrency Translation).

$$T_{\mathsf{Go}}(\mathsf{go}\ f(x)) = \mathsf{spawn}(T_{\mathsf{Go}}(f)(T_{\mathsf{Go}}(x)))$$
$$T_{\mathsf{Go}}(ch \texttt{ <- } v) = \mathsf{chan\_send}(ch, T_{\mathsf{Go}}(v))$$
$$T_{\mathsf{Go}}(\texttt{<-}ch) = \mathsf{chan\_recv}(ch)$$

Go Translation

```
module Go

let rec translate_type (t : go_type) : brrr_type =
  match t with
  | GoBool → TBool
  | GoInt → TInt I64 Signed
  | GoInt32 → TInt I32 Signed
  | GoInt64 → TInt I64 Signed
  | GoUint64 → TInt I64 Unsigned
  | GoFloat64 → TFloat F64
  | GoString → TString
  | GoSlice a → TSlice (translate_type a) GC
  | GoMap (k, v) → TDict (translate_type k) (translate_type v) GC
  | GoChan a → TChannel (translate_type a)
  | GoPtr a → TPtr (translate_type a) GC
  | GoInterface methods → TDyn (translate_methods methods)
  | GoStruct fields → TStruct (translate_fields fields) GC
  | GoFunc (params, results) →
      TFunc (List.map translate_type params)
            (tuple_or_single (List.map translate_type results))
            go_effects

let go_effects = RowExtend Panic (RowExtend Spawn RowVar)

let rec translate_expr (e : go_expr) : brrr_expr =
  match e with
  | GoVar x → EVar x
  | GoGo f args → ESpawn (EApp (translate_expr f) (List.map
     translate_expr args))
  | GoChanSend (ch, v) → EChanSend (translate_expr ch) (translate_expr
      v)
  | GoChanRecv ch → EChanRecv (translate_expr ch)
  | GoSelect cases →
      ESelect (List.map (fun (ch, dir, body) →
        match dir with
        | Send v → (SelectSend (translate_expr ch) (translate_expr v),
            translate_expr body)
        | Recv x → (SelectRecv (translate_expr ch) x, translate_expr
           body)
      ) cases)
  | GoDefer body → EDefer (translate_expr body)
  | GoPanic v → EPanic (translate_expr v)
  | GoRecover → ERecover
```

K. Swift Translation

K.1 Type Mapping

Definition K.1 (Swift Type Translation)**.**

$$\begin{aligned} T_{\mathsf{Sw}}(\texttt{Bool}) &= \mathsf{Bool} \\ T_{\mathsf{Sw}}(\texttt{Int}) &= \mathsf{Int}[\mathsf{I64}, \mathsf{Signed}] \\ T_{\mathsf{Sw}}(\texttt{Double}) &= \mathsf{Float}[\mathsf{F64}] \\ T_{\mathsf{Sw}}(\texttt{String}) &= \mathsf{String} \\ T_{\mathsf{Sw}}([A]) &= \mathsf{Array}[[]T_{\mathsf{Sw}}(A)] \quad \text{(CoW)} \\ T_{\mathsf{Sw}}(A?) &= \mathsf{Option}[[]T_{\mathsf{Sw}}(A)] \\ T_{\mathsf{Sw}}(\texttt{class}\, C) &= \mathsf{arc}\, T_{\mathsf{Sw}}^{\mathsf{fields}}(C) \\ T_{\mathsf{Sw}}(\texttt{struct}\, S) &= \mathsf{own}\, T_{\mathsf{Sw}}^{\mathsf{fields}}(S) \\ T_{\mathsf{Sw}}(\texttt{actor}\, A) &= [T_{\mathsf{Sw}}^{\mathsf{fields}}(A)] \end{aligned}$$

K.2 Value vs Reference Semantics

Definition K.2 (Swift Semantics Translation)**.** Swift structs have value semantics with copy-on-write:

$$\begin{aligned} T_{\mathsf{Sw}}(\texttt{var}\, x = s) &= \texttt{let}\, x = \mathsf{cow\_copy}(T_{\mathsf{Sw}}(s)) \\ T_{\mathsf{Sw}}(x.\mathit{field} = v) &= \mathsf{cow\_write}(x, \mathit{field}, T_{\mathsf{Sw}}(v)) \end{aligned}$$

Swift Translation

```
module Swift

type swift_semantics = Value | Reference | Actor

let rec translate_type (t : swift_type) : brrr_type * swift_semantics
    =
  match t with
  | SwBool → (TBool, Value)
  | SwInt → (TInt I64 Signed, Value)
  | SwDouble → (TFloat F64, Value)
  | SwString → (TString, Value)  (* CoW optimized *)
  | SwArray a →
      let (bt, _) = translate_type a in
      (TArray bt CoW, Value)
  | SwOptional a →
      let (bt, sem) = translate_type a in
      (TOption bt, sem)
  | SwClass (name, fields) →
      (TStruct (translate_fields fields) Arc, Reference)
  | SwStruct (name, fields) →
      (TStruct (translate_fields fields) Own, Value)
  | SwActor (name, fields) →
      (TActor (TStruct (translate_fields fields)), Actor)
  | SwProtocol methods →
      (TDyn (translate_protocol methods), Reference)

let rec translate_expr (e : swift_expr) : brrr_expr =
  match e with
  | SwVar x → EVar x
  | SwOptionalChain (base, prop) →
      EMatch (translate_expr base) [
        (PNone, ENone);
        (PSome "v", ESome (EFieldGet (EVar "v") prop))
      ]
  | SwForceUnwrap e →
      EMatch (translate_expr e) [
        (PNone, EPanic (EString "force unwrap of nil"));
        (PSome "v", EVar "v")
      ]
  | SwAsync body → EAsync (translate_expr body)
  | SwAwait e → EAwait (translate_expr e)
  | SwActorIsolated (actor, method, args) →
      EActorCall (translate_expr actor) method (List.map
         translate_expr args)
```

L. Java Translation

L.1 Type Mapping

Definition L.1 (Java Type Translation).

$$\begin{aligned} T_{\mathsf{Jv}}(\texttt{boolean}) &= \mathsf{Bool} \\ T_{\mathsf{Jv}}(\texttt{int}) &= \mathsf{Int}[\mathsf{I32}, \mathsf{Signed}] \\ T_{\mathsf{Jv}}(\texttt{long}) &= \mathsf{Int}[\mathsf{I64}, \mathsf{Signed}] \\ T_{\mathsf{Jv}}(\texttt{double}) &= \mathsf{Float}[\mathsf{F64}] \\ T_{\mathsf{Jv}}(\texttt{String}) &= \mathsf{String} \\ T_{\mathsf{Jv}}(A[]) &= \mathsf{gc}\,\mathsf{Array}[[]T_{\mathsf{Jv}}(A)] \\ T_{\mathsf{Jv}}(\texttt{List}\langle A\rangle) &= \mathsf{gc}\,[T_{\mathsf{Jv}}(A)] \\ T_{\mathsf{Jv}}(\texttt{Map}\langle K, V\rangle) &= \mathsf{gc}\,[T_{\mathsf{Jv}}(K), T_{\mathsf{Jv}}(V)] \\ T_{\mathsf{Jv}}(C) &= \mathsf{gc}\,T_{\mathsf{Jv}}^{\mathsf{class}}(C) \quad \text{(reference type)} \end{aligned}$$

L.2 Null Safety

Definition L.2 (Nullable Translation).

$$\begin{aligned} T_{\mathsf{Jv}}(T) &= \mathsf{Option}[[]T_{\mathsf{Jv}}^{\mathsf{base}}(T)] \quad \text{(for reference } T\text{)} \\ T_{\mathsf{Jv}}(\texttt{@NonNull}\,T) &= T_{\mathsf{Jv}}^{\mathsf{base}}(T) \end{aligned}$$

Java Translation

```
module Java

let rec translate_type (t : java_type) (nullability : nullable_annot)
    : brrr_type =
  match t with
  | JvPrimitive p → translate_primitive p
  | JvArray elem →
      let bt = translate_type elem Nullable in
      wrap_nullable (TArray bt GC) nullability
  | JvClass name →
      wrap_nullable (TClass name GC) nullability
  | JvGeneric (name, args) →
      let bargs = List.map (fun a → translate_type a Nullable) args in
      wrap_nullable (TGeneric name bargs GC) nullability
  | JvWildcard (bound, variance) →
      TExistential variance (translate_type bound Nullable)
```

```
and translate_primitive (p : java_primitive) : brrr_type =
  match p with
  | JvBoolean → TBool
  | JvByte → TInt I8 Signed
  | JvShort → TInt I16 Signed
  | JvInt → TInt I32 Signed
  | JvLong → TInt I64 Signed
  | JvFloat → TFloat F32
  | JvDouble → TFloat F64
  | JvChar → TChar

and wrap_nullable (t : brrr_type) (n : nullable_annot) : brrr_type =
  match n with
  | Nullable → TOption t
  | NonNull → t

let rec translate_expr (e : java_expr) : brrr_expr =
  match e with
  | JvVar x → EVar x
  | JvNull → ENone
  | JvNew (cls, args) →
      EAlloc GC (EConstruct cls (List.map translate_expr args))
  | JvMethodCall (obj, method, args) →
      (* Insert null check for nullable receiver *)
      let bobj = translate_expr obj in
      EMatch bobj [
        (PNone, EThrow (ENullPointerException));
        (PSome "recv", EMethodCall (EVar "recv") method (List.map
           translate_expr args))
      ]
  | JvFieldAccess (obj, field) →
      let bobj = translate_expr obj in
      EMatch bobj [
        (PNone, EThrow (ENullPointerException));
        (PSome "recv", EFieldGet (EVar "recv") field)
      ]
  | JvInstanceOf (e, ty) →
      ETypeTest (translate_expr e) (translate_type ty NonNull)
  | JvCast (e, ty) →
      ECast (translate_expr e) (translate_type ty NonNull)
  | JvTryCatch (body, catches, finally_) →
      ETryCatch (translate_expr body)
        (List.map (fun (exc_ty, var, handler) →
          (translate_type exc_ty NonNull, var, translate_expr handler)
        ) catches)
        (Option.map translate_expr finally_)
  | JvSynchronized (lock, body) →
      ESynchronized (translate_expr lock) (translate_expr body)
```

M. Cross-Language Boundaries

M.1 Axiom Lattice

Different languages provide different guarantees. We model this as a lattice:

Definition M.1 (Axiom Lattice).

$$\begin{tikzcd}
& \mathsf{AxAll} \arrow[dl] \arrow[d] \arrow[dr] & \\
\mathsf{AxMemSafe} \arrow[dr] & \mathsf{AxTypeSafe} \arrow[d] & \mathsf{AxNullSafe} \arrow[dl] \\
& \mathsf{AxPartial} \arrow[dl] \arrow[d] \arrow[dr] & \\
\mathsf{AxLeakFree} & \mathsf{AxRaceFree} & \mathsf{AxDetDrop} \\
& \mathsf{AxNone} \arrow[ul] \arrow[u] \arrow[ur] &
\end{tikzcd}$$

Definition M.2 (Language Axioms).

$$\begin{aligned}
\mathsf{axioms}(\mathsf{Python}) &= \{\mathsf{AxMemSafe}, \mathsf{AxLeakFree}\} \\
\mathsf{axioms}(\mathsf{TypeScript}) &= \{\mathsf{AxMemSafe}, \mathsf{AxTypeSafe}\} \\
\mathsf{axioms}(\mathsf{Rust}) &= \{\mathsf{AxMemSafe}, \mathsf{AxTypeSafe}, \mathsf{AxRaceFree}, \mathsf{AxLeakFree}\} \\
\mathsf{axioms}(\mathsf{Go}) &= \{\mathsf{AxMemSafe}, \mathsf{AxTypeSafe}, \mathsf{AxLeakFree}\} \\
\mathsf{axioms}(\mathsf{Swift}) &= \{\mathsf{AxMemSafe}, \mathsf{AxTypeSafe}, \mathsf{AxNullSafe}\} \\
\mathsf{axioms}(\mathsf{Java}) &= \{\mathsf{AxMemSafe}, \mathsf{AxTypeSafe}, \mathsf{AxLeakFree}\}
\end{aligned}$$

M.2 Boundary Guards

Definition M.3 (Guard Generation). When crossing from L_1 to L_2:

$$\mathsf{guard}(L_1, L_2, v : \tau) = \begin{cases} \mathsf{type\_check}(v, \tau) & \text{if } \mathsf{AxTypeSafe} \in \mathsf{axioms}(L_2) \setminus \mathsf{axioms}(L_1) \\ \mathsf{null\_check}(v) & \text{if } \mathsf{AxNullSafe} \in \mathsf{axioms}(L_2) \setminus \mathsf{axioms}(L_1) \\ \mathsf{pin}(v) & \text{if } \mathsf{AxMemSafe} \in \mathsf{axioms}(L_1) \setminus \mathsf{axioms}(L_2) \\ v & \text{otherwise} \end{cases}$$

Theorem M.4 (Boundary Soundness). *If P is safe under L_1's axioms and* $\mathsf{boundary}(L_1, L_2)$ *inserts appropriate guards, then cross-language calls preserve safety.*

Boundary Guards

```
module Boundary

type axiom =
  | AxMemSafe | AxTypeSafe | AxNullSafe
  | AxLeakFree | AxRaceFree | AxDetDrop

let language_axioms (l : language) : set axiom =
  match l with
  | Python → set_of_list [AxMemSafe; AxLeakFree]
  | TypeScript → set_of_list [AxMemSafe; AxTypeSafe]
  | Rust → set_of_list [AxMemSafe; AxTypeSafe; AxRaceFree; AxLeakFree]
  | Go → set_of_list [AxMemSafe; AxTypeSafe; AxLeakFree]
  | Swift → set_of_list [AxMemSafe; AxTypeSafe; AxNullSafe]
  | Java → set_of_list [AxMemSafe; AxTypeSafe; AxLeakFree]

let boundary_risks (from_lang : language) (to_lang : language) : set
    axiom =
  Set.diff (language_axioms from_lang) (language_axioms to_lang)

let generate_guard (from_lang : language) (to_lang : language)
```

```
                      (value : brrr_value) (ty : brrr_type) : brrr_expr =
  let risks = boundary_risks from_lang to_lang in
  let guarded = value in

  (* Insert type check if crossing into typed language *)
  let guarded =
    if Set.mem AxTypeSafe risks && not (Set.mem AxTypeSafe (
       language_axioms from_lang))
    then ETypeCheck guarded ty
    else guarded in

  (* Insert null check if crossing into null-safe language *)
  let guarded =
    if Set.mem AxNullSafe risks
    then EMatch guarded [
      (PNone, EThrow ENullPointerException);
      (PSome "v", EVar "v")
    ]
    else guarded in

  (* Pin GC objects if crossing into unsafe language *)
  let guarded =
    if not (Set.mem AxMemSafe (language_axioms to_lang)) &&
       Set.mem AxMemSafe (language_axioms from_lang)
    then EGCPin guarded
    else guarded in

  guarded

val boundary_call : language → language → brrr_expr → list brrr_expr →
    brrr_expr
let boundary_call from_lang to_lang callee args =
  (* Guard arguments (contravariant - from caller to callee) *)
  let guarded_args = List.map (fun (arg, ty) →
    generate_guard from_lang to_lang arg ty
  ) args in

  (* Make the call *)
  let result = EApp callee guarded_args in

  (* Guard result (covariant - from callee to caller) *)
  generate_guard to_lang from_lang result (return_type callee)

lemma boundary_soundness (l1 l2 : language) (f : brrr_expr) (args :
    list brrr_expr) :
  Lemma (requires safe_in_language l1 f)
        (ensures safe_in_language l2 (boundary_call l1 l2 f args) ∨
                 raises_boundary_exception (boundary_call l1 l2 f args
                    ))
```

A. Complete Formal Grammar

A.1 Lexical Structure

```
ident       ::= [a-zA-Z_][a-zA-Z0-9_]*
int_lit     ::= [0-9]+ | '0x' [0-9a-fA-F]+ | '0b' [01]+
float_lit   ::= [0-9]+ '.' [0-9]+ ([eE] [+-]? [0-9]+)?
string_lit  ::= '"' (escape | [^"\])* '"'
```

```
escape      ::= '\' [nrt\"\\] | '\x' [0-9a-fA-F]{2} | '\u{' [0-9a-fA-F]+ '}'
```

A.2 Type Grammar

```
type        ::= base_type ('[' effect_row ']')?
base_type   ::= 'Bool' | 'Int' | 'Float' | 'String' | 'Unit'
              | 'Option' '[' type ']'
              | 'Result' '[' type ',' type ']'
              | 'Array' '[' type ']'
              | 'Dict' '[' type ',' type ']'
              | 'Future' '[' type ']'
              | 'Channel' '[' type ']'
              | '(' type (',' type)* ')' '->' type
              | '{' (ident ':' type ';')* '}'
              | ident ('[' type (',' type)* ']')?
              | refine_type

refine_type ::= '{' ident ':' type '|' formula '}'

effect_row  ::= effect ('|' effect)* ('|' row_var)?
effect      ::= 'Pure' | 'Throw' | 'IO' | 'Async' | 'Alloc' | 'Panic'
              | ident '[' type (',' type)* ']'
row_var     ::= ident
```

A.3 Expression Grammar

```
expr        ::= literal
              | ident
              | expr '.' ident
              | expr '(' (expr (',' expr)*)? ')'
              | 'fn' '(' params ')' '->' type '{' expr '}'
              | 'if' expr 'then' expr 'else' expr
              | 'match' expr '{' (pattern '=>' expr ';')* '}'
              | 'let' ident (':' type)? '=' expr 'in' expr
              | expr binop expr
              | unop expr
              | 'throw' expr
              | 'try' expr 'catch' '{' handlers '}' ('finally' expr)?
              | 'async' expr
              | 'await' expr
              | 'spawn' expr
              | 'handle' expr 'with' '{' handlers '}'
              | 'perform' ident '(' expr ')'
              | '&' expr | '&mut' expr
              | 'move' '(' expr ')'
              | 'reset' '<' ident '>' expr
              | 'shift' '<' ident '>' expr

literal     ::= int_lit | float_lit | string_lit | 'true' | 'false' | '()'
pattern     ::= ident | literal | '_' | '(' pattern (',' pattern)* ')'
              | ident '(' pattern (',' pattern)* ')'
binop       ::= '+' | '-' | '*' | '/' | '%' | '==' | '!=' | '<' | '>'
```

```
             | '<=' | '>=' | '&&' | '||' | '|>' | '>>'
unop        ::= '-' | '!' | '*'
```

A.4 Declaration Grammar

```
decl        ::= fn_decl | type_decl | effect_decl | module_decl

fn_decl     ::= 'fn' ident type_params? '(' params ')' '->' type
                ('[' effect_row ']')? '{' expr '}'

type_decl   ::= 'type' ident type_params? '=' type
              | 'struct' ident type_params? '{' fields '}'
              | 'enum' ident type_params? '{' variants '}'

effect_decl ::= 'effect' ident type_params? '{' operations '}'

module_decl ::= 'module' ident '{' decl* '}'
              | 'signature' ident '=' sig_body
              | 'functor' ident '(' ident ':' sig_body ')' '=' module_expr

type_params ::= '[' ident (':' kind)? (',' ident (':' kind)?)* ']'
params      ::= (ident ':' type ('at' mode)?) (',' ...)*
fields      ::= (ident ':' type ';')*
variants    ::= ident ('(' type (',' type)* ')')? ('|' ...)*
operations  ::= (ident ':' type ';')*
```

B. Node Kind Enumeration

```
enum NodeKind {
    // Literals
    NLitBool, NLitInt, NLitFloat, NLitString, NLitUnit,

    // Expressions
    NVar, NApp, NLambda, NLet, NIf, NMatch, NBlock,
    NBinOp, NUnOp, NFieldGet, NFieldSet, NIndex,
    NThrow, NTryCatch, NAsync, NAwait, NSpawn,
    NHandle, NPerform, NResume,
    NBorrow, NBorrowMut, NMove, NDeref,
    NReset, NShift,

    // Patterns
    PWild, PVar, PLit, PTuple, PConstruct, POr, PGuard,

    // Types
    TBool, TInt, TFloat, TString, TUnit,
    TOption, TResult, TArray, TDict, TFuture, TChannel,
    TFunc, TStruct, TEnum, TRef, TBox,
    TForall, TExists, TRefinement,

    // Effects
    EPure, EThrow, EIO, EAsync, EAlloc, EPanic,
    ERowEmpty, ERowExtend, ERowVar,
    // Declarations
    DFn, DType, DStruct, DEnum, DEffect, DModule,
    DSignature, DFunctor,

    // Modules
    MStruct, MFunctorApp, MSeal, MPath,
}
```

C. Effect Kind Enumeration

```
enum EffectKind {
    // Core effects
    Pure,           // No effects
    Diverge,        // Non-termination

    // Exception effects
    Throw(Type),    // May throw exception of type
    Panic,          // May panic (unrecoverable)

    // I/O effects
    Read,           // File/network read
    Write,          // File/network write
    IO,             // General I/O (Read | Write)

    // Concurrency effects
    Async,          // Asynchronous computation
    Spawn,          // Thread/task spawning
    Sync,           // Synchronization primitives

    // Memory effects
    Alloc,          // Heap allocation
    Free,           // Deallocation

    // State effects
    State(Type),    // Stateful computation
    STRef,          // Mutable reference

    // Control effects
    Shift(Label),   // Delimited continuation capture
    Abort(Type),    // Early exit with value

    // Resource effects
    Resource(Name), // Named resource usage

    // Security effects
    DynamicCode,    // eval/exec
    FFI,            // Foreign function call
}
```

D. Memory Budget Calculations

Definition D.1 (Node Size Formula)**.** For a node with c children and s bytes of content:

$$\mathsf{size}(n) = 16 + 8c + s + \mathsf{padding}(s)$$

where 16 bytes is the header (kind + hash prefix).

Definition D.2 (Total Memory Budget)**.**

$$M_{\mathsf{total}} = M_{\mathsf{nodes}} + M_{\mathsf{edges}} + M_{\mathsf{strings}} + M_{\mathsf{cache}}$$

with typical ratios:

$$\begin{aligned} M_{\mathsf{nodes}} &\approx 0.4 \cdot M_{\mathsf{total}} \\ M_{\mathsf{edges}} &\approx 0.2 \cdot M_{\mathsf{total}} \\ M_{\mathsf{strings}} &\approx 0.2 \cdot M_{\mathsf{total}} \\ M_{\mathsf{cache}} &\approx 0.2 \cdot M_{\mathsf{total}} \end{aligned}$$

Definition D.3 (Tokens per Node)**.** For LLM context efficiency:

$$\mathsf{tokens}(n) \approx \frac{\mathsf{size}(n)}{4} \cdot \mathsf{compression}$$

where $\mathsf{compression} \approx 0.3$ for typical Brrr output.

E. Benchmark Reference

Metric	Small	Medium	Large	XLarge	Unit
Lines of Code	10K	100K	1M	10M	lines
Nodes	50K	500K	5M	50M	nodes
Edges	100K	1M	10M	100M	edges
Parse Time	50	500	5K	50K	ms
Type Check	100	1K	10K	100K	ms
Callgraph	20	200	2K	20K	ms
Taint Analysis	50	500	5K	50K	ms
Memory (Peak)	100	1G	10G	100G	MB

Definition E.1 (Complexity Classes)**.**

$$\begin{aligned} \text{Parsing:}&\quad O(n) \\ \text{Type Checking:}&\quad O(n \log n) \text{ typical}, O(n^2) \text{ worst} \\ \text{Callgraph:}&\quad O(n + e) \text{ flow-insensitive} \\ \text{Points-to:}&\quad O(n^3) \text{ Andersen}, O(n) \text{ Steensgaard} \\ \text{Taint:}&\quad O(n \cdot k) \text{ where } k = |\text{taint kinds}| \end{aligned}$$