Verse Logic

Eight Theorems about a Functional Logic Language

Failure, rollback, determinism and types,
proved in Lean for openverse

$$\frac{P \vdash \langle s,\ c\rangle \Downarrow \langle \mathsf{fail},\ s_1\rangle \quad P \vdash \langle s,\ e\rangle \Downarrow \langle r,\ s'\rangle}{P \vdash \langle s,\ \mathsf{if}\ c\ \mathsf{then}\ t\ \mathsf{else}\ e\rangle \Downarrow \langle r,\ s'\rangle}\ \textsc{Ite-Fail}$$

The openverse project

November 2026

Contents

Preface **1**
Who this book is for . . . 1
How the book is organized . . . 1
What is proved, and what is trusted . . . 2
The programs in this book are tested . . . 3
Acknowledgments . . . 3

1 Meaning as a Relation **4**
1.1 An opening problem . . . 4
1.2 Verse Core . . . 4
1.3 The judgment . . . 6
1.4 The interpreter . . . 7
1.5 The theorem . . . 8
1.6 The proof . . . 8
1.7 In code . . . 9
1.8 Limits and readings . . . 10

2 One Program, One Answer **12**
2.1 An opening problem . . . 12
2.2 The theorem . . . 12
2.3 Why it could fail . . . 13
2.4 The proof . . . 14
2.5 In code . . . 15
2.6 Limits and readings . . . 17

3 Failure Is Not False **19**
3.1 An opening problem . . . 19
3.2 The theorem . . . 20
3.3 The derived forms . . . 20
3.4 The proof . . . 21
3.5 In code . . . 22
3.6 Limits and readings . . . 24

4 Who May Fail **26**
4.1 An opening problem . . . 26
4.2 The analysis . . . 27

4.3 The theorem . . . 28
4.4 The proof . . . 28
4.5 In code . . . 29
4.6 Limits and readings . . . 31

5 Well-Typed Programs Do Not Go Wrong 32
5.1 An opening problem . . . 32
5.2 Types, values, heaps . . . 33
5.3 The typing judgment . . . 34
5.4 The theorem . . . 34
5.5 The proof . . . 36
5.6 In code . . . 37
5.7 Limits and readings . . . 38

6 Checked Once, Correct Everywhere 40
6.1 An opening problem . . . 40
6.2 Type variables and substitutions . . . 41
6.3 The theorem . . . 41
6.4 The proof . . . 42
6.5 How type safety uses it . . . 43
6.6 In code . . . 44
6.7 Limits and readings . . . 46

7 Look but Do Not Touch 48
7.1 An opening problem . . . 48
7.2 The analysis . . . 49
7.3 What "untouched" means . . . 49
7.4 The theorem . . . 50
7.5 The proof . . . 51
7.6 In code . . . 51
7.7 Limits and readings . . . 53

8 Everything Accepted Is Guaranteed 55
8.1 An opening problem . . . 55
8.2 The checks . . . 56
8.3 The theorem . . . 57
8.4 Why the front end need not be trusted . . . 58
8.5 The remaining errors, and the seam to C . . . 58
8.6 The whole argument . . . 60
8.7 Limits and readings . . . 60

A Notation and the Grammar of Verse Core 62
A.1 Values, outcomes, states (`Value.lean`, `Semantics.lean`) . . . 62
A.2 Expressions (`Syntax.lean`) . . . 62
A.3 Judgments . . . 63

A.4 Index of theorems 63

Preface

Verse is the programming language Epic Games designed for Unreal Editor for Fortnite and Unreal Engine. It is a *functional logic* language: besides computing values, an expression may **fail**, and a failure is caught by the nearest *failure context*, which rolls back everything the failed attempt did. That one idea gives conditionals, pattern matching, search and transactions a single meaning. It also invites the mathematician's question: what exactly does a Verse program mean, and which of the language's promises can be proved?

This book answers that question for **openverse**, a reference implementation of a deterministic subset of Verse whose semantics is defined in the Lean 4 proof assistant and whose central properties are Lean theorems. Each chapter takes one of those theorems, states it in the notation of logic, proves it in outline, reads the Lean statement line by line, and runs Verse programs that show it at work.

Who this book is for

The book is written for advanced undergraduates and graduate students in mathematics, logic and computer science. You should be at ease with sets, relations and functions; with induction on the natural numbers and on inductively defined relations; with propositional and first-order logic; and with inference rules of the form

$$\frac{\text{premise}_1 \quad \cdots \quad \text{premise}_n}{\text{conclusion}}\ \textsc{Name}$$

A first course on programming languages or type theory helps but is not required. No knowledge of Verse, Lean or games is assumed: Chapter 1 introduces the little of each that we need.

How the book is organized

Chapter	Theorem	The promise
1. Meaning as a Relation	`eval_iff`	the interpreter computes exactly the semantics
2. One Program, One Answer	`Eval.det`	evaluation is deterministic
3. Failure Is Not False	`ite_rollback`	a failed condition leaves no trace
4. Who May Fail	`Eval.no_fail`	only `<decides>` code fails
5. Well-Typed Programs Do Not Go Wrong	`Eval.typed`	no type errors, ever
6. Checked Once, Correct Everywhere	`Typed.subst`	generic code is checked once
7. Look but Do Not Touch	`Eval.no_writes`	`<reads>` code changes no object
8. Everything Accepted Is Guaranteed	`Prog.accepted`	the theorems, combined, for every program the compiler accepts

Every chapter follows the same plan: an opening problem; the theorem, first in mathematics and then in Lean; the definitions it rests on; the proof; code; limits; and exercises. Starred exercises (*) are harder. Appendix A collects the notation and the grammar of Verse Core.

What is proved, and what is trusted

openverse compiles a Verse program in two steps. A *front end* (lexer, parser and elaborator) translates surface Verse into **Verse Core**, a small expression language with names resolved to slot and function numbers. The meaning of Verse Core is an inductively defined big-step relation, and every theorem in this book is a theorem about it.

The front end itself is not verified. Instead, each program it produces is *checked* by functions written in Lean (`Prog.check`, `Prog.checkTypes`, `Prog.checkControl`), and each check is proved to establish the hypotheses of the theorems. This is **translation validation**: a bug in the front end can make it reject a good program, but it cannot make an accepted program break a promise. Chapter 8 assembles the pieces.

What remains trusted is small and explicit: Lean's kernel; the *statement* of the semantics, which is the definition of what Verse Core means; and the reading that this definition captures the language described by the Book of Verse. openverse also contains a C runtime, which is checked against the Lean interpreter by differential testing; some

of its components (the rollback store for slots and output, the loader, and the integer arithmetic) are proved separately with the QuickProof tool, and the rest is tested. We touch on that boundary only in Chapter 8.

The programs in this book are tested

Every Verse program in this book was run by openverse when the book was built, and the output printed beneath it is the output openverse produced. Programs shown as *rejected* were rejected by the front end with the message quoted. Every Lean theorem statement was compared, symbol for symbol, with the statement Lean prints for it in the openverse repository, and every excerpt of Lean source with the source itself. If the book and openverse disagree, the book does not build. To run the programs yourself, paste them into the openverse playground, which runs openverse's own compiler and runtime in a web browser.

Acknowledgments

The language is Epic Games'. openverse is an independent implementation written from the public *Book of Verse* and from the paper on the Verse Calculus by Augustsson et al. (ICFP 2023). Where the Book of Verse is silent or contradicts itself, openverse takes a reading of its own, documented in its `docs/LANGUAGE.md`; when this book relies on one, it says so. Verse, Unreal Engine and Fortnite are trademarks of Epic Games.

Thank you to Epic Games for creating Verse and for publishing the Book of Verse for everyone to read, and to the authors of the Verse Calculus. openverse is not affiliated with, sponsored by or endorsed by Epic Games. Where this book and Epic's documentation differ about Verse, Epic's documentation describes the language; this book describes openverse.

CHAPTER 1

Meaning as a Relation

1.1 An opening problem

Here is a small Verse program. Read it before you read its output.

```
Main():void =
    var Coins:int = 10
    if (set Coins = Coins - 25, Coins >= 0):
        Print("bought the map")
    else:
        Print("cannot afford it: {Coins} coins")
```

Run in the playground

```
cannot afford it: 10 coins
```

The condition of the `if` *assigns* to `Coins`, setting it to -15, and then *fails*, because $-15 \geq 0$ is false and a comparison that does not hold fails in Verse. The `else` branch then sees `Coins` as 10: the assignment has been undone. Verse calls this **rollback**, and the `if` condition is a **failure context**.

What licenses that answer? A programmer would say "run it and see". A logician would ask for the *definition* of what the program means, and then derive the answer from it. openverse provides both, and the first theorem of this book says that they agree. The definition is an inductively defined relation, a big-step operational semantics. The procedure is an interpreter, a total Lean function that may run out of "fuel". The adequacy theorem `eval_iff` states that the relation holds exactly when the interpreter, given enough fuel, returns the same outcome. Every later chapter proves things about the relation, and gets for free that they hold of the interpreter, and so of `openverse run`.

1.2 Verse Core

openverse does not give meaning to surface Verse directly. Its front end *elaborates* the program into **Verse Core**: a small language whose names are resolved, so that a local variable is a slot number in the frame of the current call and a function is an index into the program. The grammar of Verse Core expressions is

$$
\begin{aligned}
e \quad ::= \quad & \mathsf{lit}\ v \mid \mathsf{local}\ i \mid \mathsf{assign}\ i\ e \mid \mathsf{seq}\ e_1\ e_2 \mid \mathsf{prim}\ p\ \vec{e} \mid \mathsf{call}\ f\ \vec{e} \\
\mid \quad & \mathsf{ite}\ c\ t\ e \mid \mathsf{for}\ i\ e_{it}\ e_{filt}\ e_{body} \mid \mathsf{loop}\ e \mid \mathsf{brk} \mid \mathsf{ret}\ e \mid \mathsf{print}\ e \mid \mathsf{heap}\ o\ \vec{e}
\end{aligned}
$$

where v ranges over values, i over slot numbers, f over function numbers, p over primitive operations (+, comparisons, indexing, ...) and o over heap operations (allocation, field reads and writes, class tests). Surface constructs such as `not`, `or`, `option{…}` and `and` are *not* primitive: they are derived forms, defined as Verse Core expressions in Chapter 3.

The opening program becomes, after elaboration (abbreviating string literals and the initialization of `Coins`),

$$
\begin{aligned}
&\mathsf{seq}\ (\mathsf{assign}\ 0\ (\mathsf{lit}\ 10))\ (\mathsf{ite}\ c\ e_{then}\ e_{else}) \quad \text{where} \\
&\quad c \ = \ \mathsf{seq}\ (\mathsf{assign}\ 0\ (\mathsf{prim} - [\mathsf{local}\ 0, \mathsf{lit}\ 25]))\ (\mathsf{prim} \geq [\mathsf{local}\ 0, \mathsf{lit}\ 0])
\end{aligned}
$$

The command `openverse core FILE` prints this term for any program.

An evaluation has one of five **outcomes**:

$$
r \ ::= \ \mathsf{ok}\ v \mid \mathsf{fail} \mid \mathsf{brk} \mid \mathsf{ret}\ v \mid \mathsf{err}\ k
$$

Success with a value; Verse failure; a `break` on its way to the nearest `loop`; a `return` on its way to the function boundary; and a runtime error of kind k (overflow, a type error, a slot or call error, a control error, or an explicit `Err()`). In Lean:

```
inductive Res where
  /-- Success with a value. -/
  | ok   : Value → Res
  /-- Verse failure: caught by the nearest failure context, which rolls
      back. -/
  | fail : Res
  /-- `break` travelling to the nearest `loop`. -/
  | brk  : Res
  /-- `return v` travelling to the function boundary. -/
  | ret  : Value → Res
  /-- A runtime error: ends the program. -/
  | err  : ErrKind → Res
```

A **state** s has two parts: the *frame* $s.fr$, the list of values in the slots of the current call, and the *global* part $s.g$, which holds the output printed so far and the heap of objects. Both parts are restored on rollback; that the output is too is openverse's *transactional* reading of `Print`.

1.3 The judgment

The meaning of Verse Core is the relation

$$P \vdash \langle s,\, e\rangle \Downarrow \langle r,\, s'\rangle$$

read: *in program P, from state s, expression e has outcome r and leaves state s'.* It is written `Eval P s e r s'` in Lean, and it is the least relation closed under a set of inference rules, one per Lean constructor. Here are the rules for literals, sequencing and `if`; the others follow the same pattern.

$$\frac{}{P \vdash \langle s,\, \mathsf{lit}\ v\rangle \Downarrow \langle \mathsf{ok}\, v,\, s\rangle}\ \textsc{Lit} \qquad \frac{P \vdash \langle s,\, a\rangle \Downarrow \langle \mathsf{ok}\, v,\, s_1\rangle \quad P \vdash \langle s_1,\, b\rangle \Downarrow \langle r,\, s_2\rangle}{P \vdash \langle s,\, \mathsf{seq}\ a\ b\rangle \Downarrow \langle r,\, s_2\rangle}\ \textsc{Seq-Ok}$$

$$\frac{P \vdash \langle s,\, a\rangle \Downarrow \langle r,\, s_1\rangle \quad r \neq \mathsf{ok}\ _}{P \vdash \langle s,\, \mathsf{seq}\ a\ b\rangle \Downarrow \langle r,\, s_1\rangle}\ \textsc{Seq-Stop}$$

$$\frac{P \vdash \langle s,\, c\rangle \Downarrow \langle \mathsf{ok}\, v,\, s_1\rangle \quad P \vdash \langle s_1,\, t\rangle \Downarrow \langle r,\, s_2\rangle}{P \vdash \langle s,\, \mathsf{ite}\ c\ t\ e\rangle \Downarrow \langle r,\, s_2\rangle}\ \textsc{Ite-Ok}$$

$$\frac{P \vdash \langle s,\, c\rangle \Downarrow \langle \mathsf{fail},\, s_1\rangle \quad P \vdash \langle s,\, e\rangle \Downarrow \langle r,\, s_2\rangle}{P \vdash \langle s,\, \mathsf{ite}\ c\ t\ e\rangle \Downarrow \langle r,\, s_2\rangle}\ \textsc{Ite-Fail}$$

$$\frac{P \vdash \langle s,\, c\rangle \Downarrow \langle r,\, s_1\rangle \quad r \neq \mathsf{ok}\ _ \quad r \neq \mathsf{fail}}{P \vdash \langle s,\, \mathsf{ite}\ c\ t\ e\rangle \Downarrow \langle r,\, s_1\rangle}\ \textsc{Ite-Stop}$$

Three features deserve attention.

Abrupt outcomes propagate. A "stop" rule such as Seq-Stop passes on any outcome other than success: failure, `break`, `return` or an error stops the sequence and becomes its outcome. There is one such rule for every construct with subexpressions, and they are what make failure travel outward until something catches it.

Failure contexts catch failure, and roll back. In Ite-Fail the condition fails in state s_1, but the `else` branch starts from s, the state *before* the condition ran. The intermediate state s_1 is simply dropped. That is the whole of rollback, at the level of the definition: no undo log, no snapshot, just a premise that mentions s and not s_1. (The interpreter and the C runtime implement the same thing with copy-on-write and an undo journal; that they agree with the definition is the point of this chapter and of openverse's tests.) Note that Ite-Stop does *not* catch a runtime error or a `return`: only fail is caught.

The relation is the definition. There is nothing behind it. A Verse Core program

means what the rules derive, and the Lean source is the authoritative statement of the rules:

```
| ite_ok {s s₁ s₂ c t e v r} :
    Eval P s c (.ok v) s₁ → Eval P s₁ t r s₂ →
    Eval P s (.ite c t e) r s₂
/-- Rollback: the `else` branch starts from `s`, not from where `c`
    failed. -/
| ite_fail {s s₁ s₂ c t e r} :
    Eval P s c .fail s₁ → Eval P s e r s₂ →
    Eval P s (.ite c t e) r s₂
```

Each constructor is an inference rule: the arrows separate premises, the last line is the conclusion, and the variables in braces are universally quantified. The relation is defined mutually with `EvalList`, for argument lists evaluated left to right, and `EvalFor`, for the iterations of a `for`.

A whole program **runs** to outcome r with output o when calling `Main` from the initial state (an empty frame, no output, an empty heap) derives r and leaves output o:

$$\text{Runs } P\ r\ o \quad :\Longleftrightarrow \quad \exists fr\ h.\ P \vdash \langle s_0, \mathsf{call}\ \mathit{main}\ [] \rangle \Downarrow \langle r, \langle fr, \langle o, h \rangle \rangle \rangle$$

1.4 The interpreter

A relation is a definition, not a program. To *run* Verse Core, openverse has a reference interpreter, `eval`, a Lean function of type

$$\text{eval} : \text{Prog} \to \mathbb{N} \to \text{St} \to \text{Expr} \to \text{Option}(\text{Res} \times \text{St}).$$

The natural number is **fuel**. Every recursive call spends one unit, so `eval` is defined by structural recursion on the fuel and Lean accepts it as a total function, even though Verse programs need not terminate. When the fuel runs out, `eval` returns `none`; otherwise it returns the outcome and the final state. Its `if` case mirrors the three rules:

```
| n+1, s, .ite c t e => match eval P n s c with
    | none => none
    | some (.ok _, s₁) => eval P n s₁ t
    | some (.fail, _) => eval P n s e
```

On failure the interpreter, like Ite-Fail, discards the state the condition left and continues from `s`. This is the function that `openverse run` executes, and the C runtime is a transcription of it.

1.5 The theorem

Adequacy (`eval_iff`) and fuel irrelevance (`eval_fuel_irrelevant`)

For every program P, states s, s', expression e and outcome r:

$$P \vdash \langle s,\, e\rangle \Downarrow \langle r,\, s'\rangle \iff \exists n \in \mathbb{N}.\ \mathrm{eval}\ P\ n\ s\ e = \mathsf{some}\ (r, s').$$

Moreover, for all fuels n_1, n_2 and results x_1, x_2:

$$\mathrm{eval}\ P\ n_1\ s\ e = \mathsf{some}\ x_1 \ \wedge\ \mathrm{eval}\ P\ n_2\ s\ e = \mathsf{some}\ x_2 \implies x_1 = x_2.$$

In Lean, as printed by `#check`:

```
@eval_iff : ∀ {P : Prog} {s : St} {e : Expr} {r : Res} {s' : St},
  Eval P s e r s' ↔ ∃ n, eval P n s e = some (r, s')
```

```
@eval_fuel_irrelevant : ∀ {P : Prog} {n₁ n₂ : Nat} {s : St} {e : Expr}
    {x₁ x₂ : Res × St},
  eval P n₁ s e = some x₁ → eval P n₂ s e = some x₂ → x₁ = x₂
```

Read the first line left to right. The `@` and the braces mark *implicit* arguments, which Lean infers at each use; logically they are universal quantifiers, $\forall P\, s\, e\, r\, s'$. `Eval P s e r s'` is our judgment. `↔` is $\iff$, and `∃ n, eval P n s e = some (r, s')` is the right-hand side, with `some` the constructor of `Option`. The second statement uses right-nested implications, `A → B → C`, which is Lean's curried spelling of $A \wedge B \Rightarrow C$. The axioms Lean reports for both proofs are `propext` and `Quot.sound`, the standard ones; there is no `sorry`.

Fuel irrelevance says that fuel decides *whether* the interpreter finishes, never *what* it returns. Adequacy says that what it returns is exactly what the rules derive.

1.6 The proof

The two directions are proved separately, as `eval_sound` and `eval_complete`, each mutually with its counterparts for argument lists and `for` loops.

Soundness ($\Leftarrow$): *if* $\mathrm{eval}\ P\ n\ s\ e = \mathsf{some}\ (r, s')$ *then* $P \vdash \langle s,\, e\rangle \Downarrow \langle r,\, s'\rangle$. By induction on n. With $n = 0$ the interpreter returns `none`, so there is nothing to prove. With $n + 1$, case on e: each clause of `eval` makes recursive calls with fuel n, whose results the induction hypothesis turns into derivations, and these are exactly the premises of the rule that matches the clause. For example, if the condition of an `ite` returns $\mathsf{some}(\mathsf{fail}, s_1)$ and the else branch $\mathsf{some}(r, s_2)$, the hypothesis gives $P \vdash \langle s,\, c\rangle \Downarrow \langle \mathsf{fail},\, s_1\rangle$ and $P \vdash \langle s,\, e\rangle \Downarrow \langle r,\, s_2\rangle$, and Ite-Fail concludes. Small lemmas about the interpreter's "bind" combinators (`bindR_some`, `bindR_stop`) do the bookkeeping of the stop rules.

Completeness ($\Rightarrow$) is the subtler direction, and the proof strengthens the statement before proving it. The naive induction fails in the Seq-Ok case: the hypotheses give fuel n_1 that suffices for a and fuel n_2 that suffices for b, but the interpreter runs both a and b with the *same* fuel. One needs to know that more fuel does not hurt. Rather than proving monotonicity separately, openverse builds it into the induction:

Eventually

$$\text{Eventually}\ f\ y \quad :\Longleftrightarrow \quad \exists N.\ \forall m \geq N.\ f\ m = \text{some}\ y$$

```
def Eventually (f : Nat → Option α) (y : α) : Prop :=
  ∃ n, ∀ m, n ≤ m → f m = some y
```

The completeness lemma proves, by induction on the *derivation*, that

$$P \vdash \langle s,\ e\rangle \Downarrow \langle r,\ s'\rangle \implies \text{Eventually}\ (\lambda n.\ \text{eval}\ P\ n\ s\ e)\ (r, s').$$

Now the Seq-Ok case is easy. The hypotheses give thresholds N_1 for a and N_2 for b; take $N = \max(N_1, N_2) + 1$. For any $m \geq N$ write $m = k + 1$; then $k \geq N_1$ and $k \geq N_2$, so the interpreter's two recursive calls with fuel k return the right results, and the `seq` clause combines them. Every case has this shape: the threshold of a rule is one more than the maximum of its premises' thresholds. The `call_ok` case, where the derivation of the callee's body is a premise, is no different, which is why recursive functions need no special treatment: a derivation is a finite tree, even when the program it describes calls itself.

Adequacy follows by taking $m = N$, and fuel irrelevance is now a corollary rather than a lemma. Suppose the interpreter returns x_1 with fuel n_1 and x_2 with fuel n_2. By soundness both are derivable; by completeness, each is *eventually* the interpreter's answer, from thresholds N_1 and N_2. At fuel $\max(N_1, N_2)$ the interpreter returns both, so $\text{some}\ x_1 = \text{some}\ x_2$ and $x_1 = x_2$. The same three lines will prove determinism in Chapter 2.

1.7 In code

The opening program is a derivation. Its `if` condition, a sequence, derives fail by Seq-Ok (the assignment succeeds) and then the comparison's own failure; Ite-Fail discards the state in which `Coins` is -15 and evaluates the `else` branch from the state in which it is 10.

The output is part of the state, so it is rolled back too:

```
Main():void =
    if (Print("checking the vault"), 1 > 2):
        Print("impossible")
    Print("done")
```

Run in the playground

```
done
```

The first `Print` succeeded inside the condition, but the condition as a whole failed, so its output belongs to the dropped state s_1.

What does adequacy say about a program that does not terminate? Nothing directly: such a program has no derivation, because a derivation is finite, and by adequacy the interpreter returns `none` at every fuel. openverse reports this as a resource limit (exit code 3) when its fuel runs out:

```
Main():void =
    Print("searching")
    loop:
        Print("still searching")
```

Run in the playground

```
(no output)
```

Notice that *nothing* is printed, not even the first line. The reference interpreter computes a final state, output included; when the fuel runs out, there is no final state, and so there is no output. A run either has an outcome and an output, or neither.

When a program *does* terminate, the reference interpreter and the definition give the same answer by `eval_iff`, and the C runtime gives the same bytes again by openverse's differential tests. Here is a program whose answer needs some fuel, a recursion of depth 20:

```
Fib(N:int):int =
    if (N < 2) then N else Fib(N - 1) + Fib(N - 2)

Main():void =
    Print("Fib(20) = {Fib(20)}")
```

Run in the playground

```
Fib(20) = 6765
```

1.8 Limits and readings

- **Big-step semantics cannot see divergence.** A non-terminating program has no derivation. By adequacy, a configuration has no derivation exactly when the interpreter returns `none` at every fuel, that is, when the interpreter runs forever: there are no *stuck* configurations. Ill-shaped ones (a slot out of range, a

call with the wrong number of arguments, an operation on values of the wrong type) derive an *error* outcome rather than no outcome, and Chapters 5 and 8 show that checked programs never produce those errors.

- **The interpreter is total only because of fuel.** `eval` is a function in Lean's logic, so it must terminate on every input; fuel is the price. The theorem shows that the price is paid only in time, never in meaning.
- **The definition is trusted.** Adequacy relates two Lean objects. That the rules capture the language described by the Book of Verse is not a theorem; it is the reading openverse documents and tests.

Exercises

1. Write the rules Assign-Ok and Assign-Stop for assign $i\ e$ in the notation of this chapter, using writeSlot $s\ i\ v\ =$ some s' for the state with slot i set to v. Compare with `Eval.assign_ok` in `Semantics.lean`.
2. Derive the outcome of the opening program step by step, naming each rule. How many uses of Seq-Ok does the derivation contain?
3. Using only the two theorems of this chapter, prove: if eval $P\ n\ s\ e =$ some x, $n \le m$, and eval $P\ m\ s\ e \neq$ none, then eval $P\ m\ s\ e =$ some x (this is openverse's `eval_mono`). Then (*) drop the third hypothesis: prove, by induction on n and directly from the definition of `eval`, that more fuel never turns a result into `none`. Why do the theorems alone not suffice?
4. State in Lean, without proof, the claim that the program `Main():void = loop: Print("x")` has no run:

   ```
   theorem spin_no_run (P : Prog) (hP : SpinsForever P) :
       ∀ r out, ¬ Runs P r out := sorry
   ```

   What would `SpinsForever` have to say about `P`? Why is this statement about `Runs` and not about the interpreter?
5. (*) Suppose the rule Ite-Fail were changed so that the `else` branch starts from s_1 instead of s. Which programs of this chapter change their output? Is adequacy still provable, if the interpreter is changed to match?

Chapter 2

One Program, One Answer

2.1 An opening problem

Logic programming usually means search. A Prolog query may have many answers, produced in an order that depends on the search strategy, and the Verse Calculus, the core calculus behind Verse, has an operator of *choice*, $e_1 \mid e_2$, under which one expression has several values. Yet a Verse program run by openverse prints the same thing every time:

```
Main():void =
    Pairs := for (X := 1..3, Y := 1..3, X < Y) { X * 10 + Y }
    for (P : Pairs) { Print("{P}") }
```

Run in the playground

```
12
13
23
```

The `for` explores a search space (all pairs with $X < Y$), but in a fixed order: generators left to right, each in ascending order, filters pruning as they go. Is this an accident of the implementation, or a property of the language that openverse implements? For the programmer, the question is whether a test that passed once will pass again, and whether the Lean interpreter and the C runtime *can* disagree on a correct program. For the logician, it is whether the relation of Chapter 1 is the graph of a partial function. The theorem of this chapter says that it is.

2.2 The theorem

Determinism (`Eval.det`, `Runs.det`)

For every program P, state s and expression e:

$$P \vdash \langle s, e\rangle \Downarrow \langle r_1, s_1\rangle \ \wedge \ P \vdash \langle s, e\rangle \Downarrow \langle r_2, s_2\rangle \implies r_1 = r_2 \ \wedge \ s_1 = s_2.$$

Consequently a program has at most one run:

$$\text{Runs}\ P\ r_1\ o_1\ \wedge\ \text{Runs}\ P\ r_2\ o_2\ \Longrightarrow\ r_1 = r_2\ \wedge\ o_1 = o_2.$$

Equivalently, with the uniqueness quantifier $\exists^{\leq 1}$ ("at most one"):

$$\forall P\, s\, e.\ \exists^{\leq 1}(r, s').\ P \vdash \langle s, e\rangle \Downarrow \langle r, s'\rangle.$$

Note "at most": a program that runs forever has *no* outcome, and the theorem says nothing about it.

```
@Eval.det : ∀ {P : Prog} {s : St} {e : Expr} {r₁ : Res} {s₁ : St} {r₂ : Res}
    {s₂ : St},
  Eval P s e r₁ s₁ → Eval P s e r₂ s₂ → r₁ = r₂ ∧ s₁ = s₂
```

```
@Runs.det : ∀ {P : Prog} {r₁ : Res} {o₁ : List Nat} {r₂ : Res}
    {o₂ : List Nat},
  Runs P r₁ o₁ → Runs P r₂ o₂ → r₁ = r₂ ∧ o₁ = o₂
```

The output o is a `List Nat`: the bytes printed (in a typed program, each a number below 256; the type `Nat` does not enforce it). The states s_1, s_2 in `Eval.det` include the frame *and* the heap, so determinism covers more than printed output: the values of all variables, every object, and even the addresses at which objects were allocated.

2.3 Why it could fail

A relation defined by inference rules is deterministic only if the rules are designed for it. Two things could go wrong.

Overlapping rules. If two rules can conclude about the same expression, with compatible premises, there may be two derivations with different outcomes. Look again at the three `ite` rules of Chapter 1. Their first premises are about the same condition c, from the same state s, with outcomes $\mathsf{ok}\, v$, fail, and "anything else":

$$\frac{P \vdash \langle s, c\rangle \Downarrow \langle r, s_1\rangle \quad r \neq \mathsf{ok}_ \quad r \neq \mathsf{fail}}{P \vdash \langle s, \mathsf{ite}\ c\ t\ e\rangle \Downarrow \langle r, s_1\rangle}\ \textsc{Ite-Stop}$$

The side conditions $r \neq \mathsf{ok}_$ and $r \neq \mathsf{fail}$ are what keep Ite-Stop from overlapping the other two. Every construct of Verse Core follows the same discipline: its rules are told apart by the outcome of the subexpression evaluated first (a success, a failure where that matters, or anything else), and, where a successful value can still be ill-shaped, by a side condition on that value (Assign-Bad for a slot outside the frame, Call-Bad for a missing function, ...). (In Lean the side condition "not a success" is written `r.isOk = false`.)

Nondeterministic primitives. A rule whose conclusion is not a function of its premises would break determinism even without overlaps: a rule for choice, say, with conclusions $\mathsf{ok}\, v_1$ and $\mathsf{ok}\, v_2$ for $e_1 \mid e_2$, or an allocation rule that picks *some* fresh address. openverse has neither. `choice` is not in the supported subset. Concurrency is (`spawn`, `race`, `sync`, `Sleep`), but it adds no rule to Verse Core: the front end compiles it into ordinary Core code, as the next section shows. And allocation takes the *next* address, the current size of the heap:

```
  | .new t, vs, ⟨out, heap⟩ =>
      (.ok (.ref heap.size), ⟨out, heap.push ⟨t, vs⟩⟩)
```

Primitive operations are functions too (`Prim.apply`, `HeapOp.apply`), so each rule's conclusion is determined by its premises.

2.4 The proof

There are two proofs, and it is instructive to see both.

The textbook proof is by induction on the first derivation, inverting the second. Suppose $\mathcal{D}_1$ derives $P \vdash \langle s, e\rangle \Downarrow \langle r_1, s_1\rangle$ and $\mathcal{D}_2$ derives $P \vdash \langle s, e\rangle \Downarrow \langle r_2, s_2\rangle$; we show $r_1 = r_2$ and $s_1 = s_2$ by induction on $\mathcal{D}_1$. The last rule of $\mathcal{D}_2$ must be a rule for the same construct. Take $e = \mathsf{ite}\ c\ t\ e'$ and $\mathcal{D}_1$ ending in Ite-Ok, with premises $P \vdash \langle s, c\rangle \Downarrow \langle \mathsf{ok}\, v, s_1'\rangle$ and $P \vdash \langle s_1', t\rangle \Downarrow \langle r_1, s_1\rangle$.

- If $\mathcal{D}_2$ ends in Ite-Fail, its first premise is $P \vdash \langle s, c\rangle \Downarrow \langle \mathsf{fail}, s''\rangle$. The induction hypothesis for the first premise of $\mathcal{D}_1$ gives $\mathsf{ok}\, v = \mathsf{fail}$: impossible.
- If $\mathcal{D}_2$ ends in Ite-Stop, its first premise gives an outcome $r \neq \mathsf{ok}\ _$; again the hypothesis gives $\mathsf{ok}\, v = r$, a contradiction.
- If $\mathcal{D}_2$ ends in Ite-Ok, with premises $P \vdash \langle s, c\rangle \Downarrow \langle \mathsf{ok}\, v', s''\rangle$ and $P \vdash \langle s'', t\rangle \Downarrow \langle r_2, s_2\rangle$, the hypothesis gives $v = v'$ and $s_1' = s''$; so the second premises start in the same state, and the hypothesis for them gives $r_1 = r_2$ and $s_1 = s_2$.

Every case of every construct is like one of these: an "ok" rule against a "stop" rule is a contradiction through the side condition, and two rules of the same kind agree by the hypotheses applied left to right, each premise starting where the previous one ended. For `call`, the hypothesis for the argument list gives the same arguments, hence the same callee and frame, and the hypothesis for the body gives the same result.

openverse's proof is three lines long, because Chapter 1 already did the work:

```
theorem Eval.det {P : Prog} {s e r₁ s₁ r₂ s₂}
    (h₁ : Eval P s e r₁ s₁) (h₂ : Eval P s e r₂ s₂) :
    r₁ = r₂ ∧ s₁ = s₂ := by
  obtain ⟨n₁, H₁⟩ := eval_iff.mp h₁
  obtain ⟨n₂, H₂⟩ := eval_iff.mp h₂
  have := eval_fuel_irrelevant H₁ H₂
```

```
  simp only [Prod.mk.injEq] at this
  exact this
```

In words: by adequacy, the interpreter returns (r_1, s_1) at some fuel n_1 and (r_2, s_2) at some fuel n_2; by fuel irrelevance these results are equal; equality of pairs is equality of components. The `obtain` lines destructure the existentials that `eval_iff.mp` (the left-to-right direction of the biconditional) produces.

The logic behind the short proof is worth stating as a principle.

Determinism for free

If a relation $R \subseteq A \times B$ satisfies $R\,a\,b \iff \exists n.\, f\ n\ a = \text{some}\ b$ for a family of partial functions $f\ n$ that agree wherever they are defined, then R is the graph of a partial function, hence deterministic. Defining an interpreter and proving it adequate yields determinism without a separate induction.

The textbook induction is not needed at all, but its work is done elsewhere. Fuel irrelevance came from the `Eventually` form of completeness, an induction on a *single* derivation; it never compares two rules. The exclusivity that the textbook proof checks case by case is built into `eval`: a Lean function defined by pattern matching has one clause for each shape of input, so it cannot return two answers. Adequacy transfers that property from the function to the relation.

`Runs.det` follows from `Eval.det`: two runs are two derivations from the same initial state for the same expression, call *main* `[]`, so their final states are equal, hence so are their outputs (`congrArg (fun s : St => s.g.out)`).

2.5 In code

Determinism is what makes the order of a `for` meaningful. Generators run left to right, the leftmost slowest, like nested loops; the result array lists the body's values in that order, as in the opening program.

Maps iterate in insertion order. Updating a key keeps its place; a new key goes at the end:

```
Main():void =
    var Loot:[string]int = map{"ruby" => 3, "coin" => 50}
    set Loot["amber"] = 7
    set Loot["ruby"] = 4
    for (Name -> Count : Loot) { Print("{Name}: {Count}") }
```

Run in the playground

```
ruby: 4
coin: 50
amber: 7
```

In Verse Core a map is simply the array of its key-value pairs, so map iteration inherits the determinism of array iteration. (This is also why openverse's map equality is order-sensitive, a reading documented in its `LANGUAGE.md`.)

Allocation is deterministic too, and identity is observable. Instances of a `<unique>` class compare by identity, so two objects with equal fields differ:

```
gem := class<unique>:
    Name:string

Main():void =
    A := gem{Name := "opal"}
    B := gem{Name := "opal"}
    if (A = B) then Print("same gem") else Print("two gems")
    if (A = A) then Print("A is A") else Print("?")
```

Run in the playground

```
two gems
A is A
```

Here `A` and `B` are references ref a and ref b to the objects allocated at addresses a and $b = a + 1$, the sizes of the heap at the two allocations. A semantics that allocated at an arbitrary fresh address would still give `two gems` (the addresses differ either way) but would not be deterministic in the sense of `Eval.det`, because the final states would differ in their addresses. Such a semantics would need determinism *up to renaming of addresses*, a weaker and much harder theorem.

Concurrent programs are deterministic for the same reason. Here two calls of `Worker` run as the arms of a `sync`:

```
Worker(Name:string, Steps:int)<suspends>:void =
    var I:int = 0
    loop:
        if (I >= Steps) then break
        Print("{Name} {I}")
        Sleep(0.0)
        set I += 1

Main()<suspends>:void =
    sync:
        Worker("a", 2)
        Worker("b", 3)
    Print("both done")
```

Run in the playground

```
a 0
b 0
a 1
b 1
```

```
b 2
both done
```

Each arm runs until it waits. `Sleep(0.0)` waits for the next tick of a virtual clock, and in each tick the arms go on in source order, so the lines interleave the same way on every run. None of this is new to Verse Core: the front end compiles `Worker` into a frame object and a function that resumes it from one wait to the next, and the scheduler is a helper written in Verse (`docs/DESIGN.md`). The whole program is an ordinary Core program, and `Eval.det` applies to it as it is. (That the lowering keeps the meaning of the surface program is tested, not proved, like the rest of the front end: Chapter 8.)

Finally, determinism is what makes openverse's testing strategy sound. Its C interpreter is compared with the Lean interpreter on every test program and on random programs, byte for byte, output and exit code. Without determinism, a disagreement would not show a bug: both could be right.

2.6 Limits and readings

- **The subset is deterministic; Verse is not.** Full Verse has choice. The theorem is about openverse's subset, and it is one reason for choosing that subset: the Verse Calculus paper itself studies how to keep choice deterministic by fixing the order of results, a more delicate property than ours. openverse's concurrency is deterministic by construction: it fixes the order in which tasks go on and counts time in ticks (`docs/LANGUAGE.md`).
- **At most one, not exactly one.** `Eval.det` gives uniqueness, not existence. A diverging program has no outcome; Chapter 1 explained why that is the only way to have none.
- **The C runtime is deterministic by testing.** Lean proves determinism of the definition and the reference interpreter. The C interpreter is a transcription checked by differential testing, under sanitizers. Its rollback store and loader have separate machine-checked proofs (Chapter 8).
- **Readings.** Several of the orders above are openverse's choices where the Book of Verse is silent: output inside a failed failure context is rolled back, a failing `for` body makes the whole `for` fail (as in the Verse Calculus), and maps compare in order. Determinism holds *given* these readings, which the rules fix.

Exercises

1. Remove the side condition $r \neq \mathsf{fail}$ from Ite-Stop. Give a Verse Core expression with two derivations and different outcomes.
2. Prove `Runs.det` from `Eval.det` on paper, using the definition of Runs from Chapter 1. Where is the output in the final state?

3. Write the textbook proof's case for seq in full: Seq-Ok against Seq-Ok, and Seq-Ok against Seq-Stop.
4. Predict the output, then run it:

```
Main():void =
    for (X := 1..2, Y := 1..2) { Print("{X}{Y}") }
```

Run in the playground

Which generator varies fastest? How would you get the other order?

5. (*) Suppose allocation chose an arbitrary address not in use. State a determinism theorem "up to renaming of heap addresses" in the notation of this chapter. Which rules would its proof have to treat specially?
6. Prove the principle *Determinism for free* in Lean:

```
theorem det_of_graph {α β} (R : α → β → Prop) (f : Nat → α → Option β)
    (hR : ∀ a b, R a b ↔ ∃ n, f n a = some b)
    (hf : ∀ n₁ n₂ a b₁ b₂, f n₁ a = some b₁ → f n₂ a = some b₂ → b₁ = b₂) :
    ∀ a b₁ b₂, R a b₁ → R a b₂ → b₁ = b₂ := by
  sorry
```

CHAPTER 3

Failure Is Not False

3.1 An opening problem

In most languages a condition is a Boolean expression: it evaluates to `true` or `false`, and evaluating it may have side effects that persist whichever way it goes. In Verse a condition is an expression that either *succeeds* or *fails*, and when it fails, its side effects disappear:

```
Main():void =
    var X:int = 1
    if (set X = 2, X = 2, set X = 3, X = 4):
        Print("never")
    Print("X = {X}")
    if (set X = 2, X = 2, set X = 3):
        Print("committed: X = {X}")
```

Run in the playground

```
X = 1
committed: X = 3
```

The first condition makes two assignments and passes one test before the last test fails, and then it is as if none of it had happened. The second succeeds, and its assignments stay. A condition is a *transaction*: it commits on success and aborts on failure.

For a programmer this is the feature that makes Verse conditions safe to write as searches ("find a key, take it, open the chest") without cleanup code for the paths that do not work out. For a logician it raises a question about connectives. If failure plays the role of falsity, what are `not`, `or` and `and`, and how do they interact with effects? This chapter proves the rollback law for `if`, then derives the others from it.

3.2 The theorem

Rollback (`ite_rollback`)

If the condition c fails from state s, then for every outcome r and state s'

$$P \vdash \langle s, c\rangle \Downarrow \langle \mathsf{fail}, s_1\rangle \implies \Big(P \vdash \langle s, \mathsf{ite}\ c\ t\ e\rangle \Downarrow \langle r, s'\rangle \iff P \vdash \langle s, e\rangle \Downarrow \langle r, s'\rangle\Big).$$

Once the condition has failed, the whole `if` is *indistinguishable* from its `else` branch run from the original state: the same outcomes, the same final states. Nothing that c did, and nothing in s_1, can be observed.

```
@ite_rollback : ∀ {P : Prog} {s s₁ : St} {c t e : Expr} {r : Res} {s' : St},
  Eval P s c Res.fail s₁ → (Eval P s (c.ite t e) r s' ↔ Eval P s e r s')
```

`c.ite t e` is Lean's dot notation for `Expr.ite c t e`. The hypothesis names the failed run of the condition, and the conclusion is a biconditional for *all* r and s': an equivalence of the two expressions, from s.

3.3 The derived forms

Verse Core has one failure context, `ite` (with the filter of `for`). The book's other failure contexts are *defined* from it, in `OpenVerse/Core/Derived.lean`:

Derived forms

$$\begin{aligned}
\mathsf{not}\ e &:= \mathsf{ite}\ e\ (\mathsf{prim}\ \mathit{fail}\ [\,])\ (\mathsf{lit}\ \mathit{void})\\
a\ \mathsf{or}_t\ b &:= \mathsf{ite}\ (\mathsf{assign}\ t\ a)\ (\mathsf{local}\ t)\ b\\
\mathsf{option}_t\{e\} &:= \mathsf{ite}\ (\mathsf{assign}\ t\ e)\ (\mathsf{prim}\ \mathit{mkSome}\ [\mathsf{local}\ t])\ (\mathsf{lit}\ \mathit{none})\\
\mathsf{logic}\{e\} &:= \mathsf{ite}\ e\ (\mathsf{lit}\ \mathit{true})\ (\mathsf{lit}\ \mathit{false})\\
a\ \mathsf{and}\ b &:= \mathsf{seq}\ a\ b
\end{aligned}$$

Here t is a fresh slot chosen by the elaborator, to carry the value of a (or e) from the condition into the `then` branch.

```
def Expr.notE (e : Expr) : Expr := .ite e (.prim .fail []) (.lit .void)
/-- `a or b`; slot `t` is a fresh temporary of the elaborator. -/
def Expr.orE (t : Nat) (a b : Expr) : Expr := .ite (.assign t a) (.local t) b
```

Read the definitions as logic. `not e` succeeds (with `void`) exactly when e fails: it is *negation as failure*, Prolog's `\+`. `a or b` tries a, and only if a fails tries b: a left-biased disjunction. `a and b` is sequencing: it fails if either fails, and its value is b's. `option{e}` reifies success or failure as a value of type `?t`, and `logic{e}` as a `logic`. The laws below say that each behaves as the Book of Verse describes, *including* what happens

to effects.

Laws of the derived forms

For all P, s, e, a, b, t:

1. (`notE_ok`) If $P \vdash \langle s, \text{not } e\rangle \Downarrow \langle \text{ok}\, v, s'\rangle$, then $v = \mathit{void}$, $s' = s$, and $\exists s_1.\ P \vdash \langle s, e\rangle \Downarrow \langle \text{fail}, s_1\rangle$.
2. (`notE_notE_ok`) If $P \vdash \langle s, \text{not (not } e)\rangle \Downarrow \langle \text{ok}\, v, s'\rangle$, then $s' = s$ and $\exists u\, s_1.\ P \vdash \langle s, e\rangle \Downarrow \langle \text{ok}\, u, s_1\rangle$.
3. (`orE_right`) If $P \vdash \langle s, a\rangle \Downarrow \langle \text{fail}, s_1\rangle$, then for all r, s': $P \vdash \langle s, a \text{ or}_t\ b\rangle \Downarrow \langle r, s'\rangle \iff P \vdash \langle s, b\rangle \Downarrow \langle r, s'\rangle$.
4. (`optionE_none`) If $P \vdash \langle s, \text{option}_t\{e\}\rangle \Downarrow \langle \text{ok}\, \mathit{none}, s'\rangle$, then $s' = s$.

```
@notE_ok : ∀ {P : Prog} {s : St} {e : Expr} {v : Value} {s' : St},
  Eval P s e.notE (Res.ok v) s' →
    v = Value.void ∧ s' = s ∧ ∃ s₁, Eval P s e Res.fail s₁
```

```
@notE_notE_ok : ∀ {P : Prog} {s : St} {e : Expr} {v : Value} {s' : St},
  Eval P s e.notE.notE (Res.ok v) s' →
    s' = s ∧ ∃ u s₁, Eval P s e (Res.ok u) s₁
```

```
@orE_right : ∀ {P : Prog} {s s₁ : St} {t : Nat} {a b : Expr} {r : Res}
    {s' : St},
  Eval P s a Res.fail s₁ → (Eval P s (Expr.orE t a b) r s' ↔ Eval P s b r s')
```

```
@optionE_none : ∀ {P : Prog} {s : St} {t : Nat} {e : Expr} {s' : St},
  Eval P s (Expr.optionE t e) (Res.ok Value.none) s' → s' = s
```

Law 2 is the surprising one, and it deserves a logician's reading. In classical logic $\neg\neg A \leftrightarrow A$. Here `not not e` succeeds exactly when `e` succeeds, but it is *not* `e`: it returns `void` instead of e's value, and it leaves the state exactly as it found it, discarding every effect of e (and, at the surface, every name e binds). It is a pure *test* of whether e would succeed. The analogy with intuitionistic logic is apt: there, $\neg\neg A$ asserts that A cannot be refuted without supplying a witness for A; here, `not not e` asserts that e succeeds without supplying its value or keeping its effects.

3.4 The proof

Rollback. The direction $\Leftarrow$ is the rule Ite-Fail itself, applied to the hypothesis $P \vdash \langle s, c\rangle \Downarrow \langle \text{fail}, s_1\rangle$ and the given derivation for e. For $\Rightarrow$, invert the derivation of the `ite`: its last rule is one of the three.

- Ite-Ok would need $P \vdash \langle s, c\rangle \Downarrow \langle \text{ok}\, v, s''\rangle$ for some v, s''. With the hypothesis, `Eval.det` gives $\text{ok}\, v = \text{fail}$, which is false.

- Ite-Stop would need $P \vdash \langle s, c\rangle \Downarrow \langle r'', s''\rangle$ with $r'' \neq \mathsf{fail}$; determinism gives $r'' = \mathsf{fail}$, a contradiction.
- Ite-Fail has the premise $P \vdash \langle s, e\rangle \Downarrow \langle r, s'\rangle$, which is what we want.

```
theorem ite_rollback {P s s₁ c t e r s'} (hc : Eval P s c .fail s₁) :
    Eval P s (.ite c t e) r s' ↔ Eval P s e r s' := by
  constructor
  · intro h
    cases h with
    | ite_ok h1 _ => exact absurd (Eval.det h1 hc).1 (by simp)
    | ite_fail _ h2 => exact h2
    | ite_stop h1 _ hf => exact absurd (Eval.det h1 hc).1 hf
  · intro h; exact .ite_fail hc h
```

Notice the dependency: rollback, a statement about *one* branch, needs determinism, a statement about *all* derivations. Without Chapter 2, the `if` could have a second derivation in which the same condition succeeds, and the equivalence would fail. The tactic `cases h with` is exactly the inversion above, one line per rule; `absurd` closes a case from a contradiction.

`not`. Unfold not e to ite e (prim *fail* []) (lit *void*) and invert a successful derivation. If the last rule is Ite-Ok, the `then` branch prim *fail* [] must have succeeded, but it always fails. If it is Ite-Stop, the outcome is not ok _, contradicting the hypothesis. So it is Ite-Fail: e failed from s, and the `else` branch, the literal `void`, succeeded *from* s without changing it (rule Lit). Hence $v = \mathit{void}$ and $s' = s$.

`not not`. Apply the law for `not` to the outer negation: $s' = s$, and the inner not e *failed* from s. Invert that failure: it cannot come from Ite-Fail, whose `else` branch succeeds; nor from Ite-Stop, whose outcome is not fail; so it is Ite-Ok, whose first premise is a success of e from s.

`or` is rollback again: a or$_t$ b is an `ite` whose condition assign t a fails when a does (by Assign-Stop), so `ite_rollback` applies. **`option{}`** is like `not`: an empty result can only come from the `else` branch, a literal, evaluated from s.

These are the laws a programmer relies on without thinking: an `or` fallback never sees a half-finished left side; `option{…}` of a failure has no effect; `not` never leaks.

3.5 In code

The double negation is a test that leaves no trace, and a single negation is too:

```
IsPositive(X:int)<decides>:int = X > 0

Main():void =
    var Count:int = 0
    if (not not (set Count = 99; IsPositive[1])):
```

```
        Print("the test passed; Count = {Count}")
    if (not (set Count = 7; IsPositive[-1])):
        Print("the negation passed; Count = {Count}")
```

Run in the playground

```
the test passed; Count = 0
the negation passed; Count = 0
```

In the first `if`, `e` succeeded, assigning 99; the outer `not` discarded that. In the second, `e` failed, so its assignment was rolled back by the inner `ite`. (Inside parentheses, `;` sequences expressions; a comma would build a tuple.) `openverse core` confirms the elaboration: the first condition is ite (ite *e fail void*) *fail void*, which is `Expr.notE (Expr.notE e)`.

`or`, `option{}` and the rollback of their left sides:

```
Halve(N:int)<decides>:int =
    Mod[N, 2] = 0
    Quotient[N, 2]

Main():void =
    var Tally:int = 0
    A := (set Tally += 1; Halve[7]) or 100
    Print("A = {A}, Tally = {Tally}")
    B:?int = option{set Tally += 10; Halve[8]}
    if (V := B?) { Print("B holds {V}, Tally = {Tally}") }
    C:?int = option{set Tally += 100; Halve[9]}
    if (not C?) { Print("C is empty, Tally = {Tally}") }
```

Run in the playground

```
A = 100, Tally = 0
B holds 4, Tally = 10
C is empty, Tally = 10
```

`Halve` is a `<decides>` function: its first line *fails* unless N is even. `A` takes the fallback 100 and the increment is undone (law 3). `B` succeeds and keeps its effect. `C` is empty and has none (law 4).

Because a condition that cannot fail would make the `else` branch dead code, and is usually a mistake (a Boolean where a test was meant), the front end rejects it:

```
Main():void =
    if (1 + 1) { Print("two") }
```

Run in the playground

Rejected: `the condition of this if cannot fail`

Rollback catches *failure* only. A runtime error, such as overflow, is not a failure and is not caught by the `if`; it ends the program, with exit code 2. That is rule Ite-Stop, which propagates every outcome of the condition other than success and failure:

```
Main():void =
    Print("before")
    var Big:int = 9223372036854775807
    if (set Big += 1, Big < 0):
        Print("wrapped around")
    else:
        Print("rolled back")
```

Run in the playground

```
before
```

The `int` of openverse is 64-bit, and an overflow is an error, never a wrap-around (the BPVM reading; see `docs/LANGUAGE.md`).

3.6 Limits and readings

- **Effects inside the condition of a successful `if` are kept**, and so are the bindings, which the `then` branch can see. Rollback is about failure only.
- **Output is rolled back** because openverse makes `Print` transactional: the output is part of the state. The Book of Verse does not say what happens to output in a failed context; this is openverse's reading. The heap is rolled back too: object writes and allocations (Chapter 7).
- **`return` inside a failure context** leaves the function without rolling back what the condition did (another openverse reading): a $\mathsf{ret}\, v$ outcome is propagated by Ite-Stop, not caught.
- **The laws are about Verse Core.** The front end elaborates surface `not`, `or` and `option{}` into exactly these definitions, which `openverse core` lets you check on any program; the scoping rule that names defined in the operands of `and`, `or` and `not` do not escape is enforced by the front end.

Exercises

1. Prove `logicE_ne_fail`: $\mathsf{logic}\{e\}$ never fails. Which rules can end a derivation of $P \vdash \langle s, \mathsf{logic}\{e\}\rangle \Downarrow \langle \mathsf{fail}, s'\rangle$?
2. Is $\mathsf{not}\ (\mathsf{not}\ (\mathsf{not}\ e))$ equivalent to $\mathsf{not}\ e$, in the sense of `ite_rollback` (same outcomes and final states from every s)? Prove it or give a counterexample.
3. State and prove the dual of law 3: if $P \vdash \langle s, a\rangle \Downarrow \langle \mathsf{ok}\, v, s_1\rangle$ and writeSlot $s_1\ t\ v = \mathsf{some}\ s_2$, then $a\ \mathsf{or}_t\ b$ succeeds with v in state s_2. Why is the hypothesis about `writeSlot` needed? (Compare `optionE_some`, and what happens when t is outside the frame.)
4. Is `and` commutative? Give two expressions a, b such that a and b and b and a have different outcomes from the same state, or prove that none exist. What if both are effect-free?

5. Predict, then check with `openverse run`:

```
Main():void =
    var N:int = 0
    if (set N += 1; not (set N += 10; N > 100)):
        Print("N = {N}")
```

Run in the playground

6. (*) Formulate a theorem "a failed `for` filter leaves no trace", the analogue of `ite_rollback` for rule `EvalFor.skip`, and prove it on paper.

CHAPTER 4

Who May Fail

4.1 An opening problem

If any expression may fail, any function call may fail, and a programmer would have to treat every call as a possible failure. Verse avoids this with an *effect*: a function that may fail is marked `<decides>`, is called with square brackets, and may be called only where failure is caught. Everything else is guaranteed not to fail. Here is the discipline at work:

```
Key(N:int)<decides>:int = N > 0

Safe(N:int):int =
    if (K := Key[N]) then K else 0

Main():void =
    Print("{Safe(5)} {Safe(-5)}")
```

Run in the playground

```
5 0
```

`Key` fails for non-positive N; `Safe` calls it inside an `if` condition, so `Safe` itself cannot fail, needs no `<decides>`, and is called with parentheses. The front end enforces the discipline. The same call outside a failure context is rejected:

```
First(A:[]int):int = A[0]

Main():void =
    Print("{First(array{4})}")
```

Run in the playground

```
Rejected: this expression can fail, but it is not in a failure context
```

Indexing can fail (the index may be out of range), and `First` is not `<decides>`. The programmer's question: is the check *sound*, so that a function that passed it really never fails? The logician's: the check is a syntactic predicate, and the property is semantic; what invariant connects them, and why does it survive recursion? This chapter answers both.

4.2 The analysis

The front end decides, for each expression, whether it *may fail*: whether it contains a failable primitive, or a call to a `<decides>` function, outside every failure context it contains. In Verse Core this is a recursive function $\mathrm{mayFail}_P : \mathrm{Expr} \to \mathbb{B}$:

May fail

Write $M(e)$ for $\mathrm{mayFail}_P(e)$. Then $M(\mathsf{lit}\ v) = M(\mathsf{local}\ i) = M(\mathsf{brk}) = \bot$, and

$$\begin{aligned} M(\mathsf{seq}\ a\ b) &= M(a) \vee M(b) \\ M(\mathsf{prim}\ p\ \vec{e}) &= \mathrm{failable}(p) \vee \bigvee_i M(e_i) \\ M(\mathsf{call}\ f\ \vec{e}) &= \mathrm{decides}(P.f) \vee \bigvee_i M(e_i) \\ M(\mathsf{ite}\ c\ t\ e) &= M(t) \vee M(e) \\ M(\mathsf{for}\ i\ a\ f\ b) &= M(a) \vee M(b) \end{aligned}$$

where a, f and b are the iterated array, the filter and the body of the `for`, and the remaining constructs (`assign`, `ret`, `print`, `loop`, `heap`) pass their subexpressions' values up, a heap operation adding $\mathrm{failable}(o)$, true only for a cast.

The two lines that matter are the last two. The condition c of an `ite` **does not appear** on the right: it is a failure context, so its failure is caught. Likewise the filter of a `for`, whose failure skips an element. In Lean:

```
| .call f es =>
  (match P.fns[f]? with
    | some fn => fn.decides
    | none => false) || Expr.mayFailList P es
-- the condition is a failure context: its failure is caught
| .ite _ t e => t.mayFail P || e.mayFail P
-- the filter is a failure context: its failure skips the element
| .forE _ it _ body => it.mayFail P || body.mayFail P
```

The analysis of a call trusts the callee's *declaration* (`fn.decides`), not its body. That is what makes it modular and cheap, and it is also why soundness needs a global hypothesis: the declarations must be honest.

Effect soundness of a program

$$\mathrm{EffectSound}(P) \quad :\Longleftrightarrow \quad \forall fn \in P.fns.\ \neg\, fn.decides \ \Rightarrow\ \neg\, \mathrm{mayFail}_P(fn.body)$$

Every function not declared `<decides>` has a body that may not fail.

```
def Prog.EffectSound (P : Prog) : Prop :=
  ∀ fn ∈ P.fns, fn.decides = false → fn.body.mayFail P = false
```

4.3 The theorem

<decides> soundness (Eval.no_fail, Runs.no_fail)

If EffectSound(P), then for all s, e, r, s':

$$P \vdash \langle s,\, e\rangle \Downarrow \langle r,\, s'\rangle \;\wedge\; \neg\,\mathrm{mayFail}_P(e) \;\Longrightarrow\; r \neq \mathsf{fail}.$$

In particular, if P's `Main` is not `<decides>`, then no run of P fails: Runs $P\ r\ o \Rightarrow r \neq$ fail.

```
@Eval.no_fail : ∀ {P : Prog},
  P.EffectSound → ∀ {s : St} {e : Expr} {r : Res} {s' : St},
    Eval P s e r s' → Expr.mayFail P e = false → r ≠ Res.fail
```

```
@Runs.no_fail : ∀ {P : Prog},
  P.EffectSound →
    ∀ {fn : Fn} {r : Res} {out : List Nat},
      P.fns[P.main]? = some fn → fn.decides = false → Runs P r out →
        r ≠ Res.fail
```

The structure of the statement is typical of *effect systems*: a syntactic judgment (¬mayFail) implies a semantic property (no failure outcome), under an invariant of the whole program (`EffectSound`). Note what is *not* claimed: the expression may still raise a runtime error, `break`, `return`, or diverge. Only failure is excluded. `P.fns[P.main]? = some fn` says that `fn` is `Main`'s definition (`?` is safe indexing into the list of functions, returning an `Option`).

4.4 The proof

By induction on the derivation of $P \vdash \langle s,\, e\rangle \Downarrow \langle r,\, s'\rangle$, mutually with the corresponding statements for `EvalList` and `EvalFor`. Assume EffectSound(P) throughout. Most cases are immediate: a stop rule propagates the outcome of a subexpression that, by the definition of mayFail, may not fail either, so the induction hypothesis applies. Four cases carry the idea.

Primitives. A successful argument list leads to p.apply $\vec{v}$, and a lemma, `Prim.apply_ne_fail`, says that a primitive that is not failable never returns fail. The failable ones are the comparisons, indexing and indexed update, the division family, `?`, and `false?`.

`ite`. If the derivation ends in Ite-Ok or Ite-Fail, the outcome is the outcome of t or of

e, and both may not fail by hypothesis, so the induction hypothesis on that branch concludes. The condition's failure, the one thing that mayFail ignored, is exactly what Ite-Fail consumes. If it ends in Ite-Stop, the outcome is the condition's, and the side condition of the rule says it is not fail. Notice that this case uses no induction hypothesis about the condition at all: soundness here is a property of the *rules*, which do not let a failed condition escape.

`for`. The filter's failure is consumed by `EvalFor.skip`; the body's failure would be propagated, but the body may not fail.

Calls. This is the case that seems to need more than induction. The derivation ends in

$$\frac{\vec{e} \Downarrow \vec{v} \quad P.\text{callable}\, f\ |\vec{v}| = \text{some}\, fn \quad P \vdash \langle\langle \text{frameOf}\, fn\ \vec{v}, s_1.g\rangle, fn.body\rangle \Downarrow \langle r,\, s_2\rangle}{P \vdash \langle s,\, \text{call}\, f\ \vec{e}\rangle \Downarrow \langle \text{retOf}(r),\, \langle s_1.fr, s_2.g\rangle\rangle}\ \textsc{Call-Ok}$$

Since $\neg\text{mayFail}(\text{call}\, f\ \vec{e})$, the callee is not `<decides>`. By EffectSound(P) its body may not fail. The derivation of the body is a *premise* of this rule, hence a subderivation, and the induction hypothesis applies to it: $r \neq \text{fail}$. Finally retOf, which turns $\text{ret}\, v$ into $\text{ok}\, v$, maps non-failures to non-failures (`retOf_ne_fail`).

The call case is where the global invariant earns its keep. A recursive function's body contains calls to the function itself, so an induction on the *syntax* of expressions would go round in circles. Induction on the *derivation* does not: each call's body has its own, strictly smaller, derivation. The price is the one we met in Chapter 1: the theorem speaks only of evaluations that finish.

`Runs.no_fail` is the instance $e = \text{call}\ main\ []$, with no arguments to fail and mayFail reduced to *main*'s `decides` flag.

Where does EffectSound(P) come from for a real program? The front end does not prove it; Lean checks it. The Boolean function `Prog.checkEffectSound` tests every function, and a two-line lemma shows that it returning `true` implies the proposition:

```
def Prog.checkEffectSound (P : Prog) : Bool :=
  P.fns.all fun fn => fn.decides || !(fn.body.mayFail P)
```

Every program the front end emits passes this check (Chapter 8), so the theorem applies to every accepted program.

4.5 In code

A `<decides>` function propagates failure to its caller; a context catches it. Here `Pick` fails out of range, and the `for` clause skips those indices:

```
Pick(A:[]int, I:int)<decides>:int = A[I]

Main():void =
    Scores := array{10, 20, 30}
    for (I := 0..5, S := Pick[Scores, I]):
        Print("score {I}: {S}")
```

Run in the playground

```
score 0: 10
score 1: 20
score 2: 30
```

The two syntaxes of calls are part of the discipline. A `<decides>` function called with parentheses is rejected, because the reader of the call could not see that it may fail:

```
Key(N:int)<decides>:int = N > 0

Main():void =
    Print("{Key(3)}")
```

Run in the playground

Rejected: `Key is <decides>: call it with Key[…]`

A `<decides>` `Main` may fail. Its failure is not caught by anything, so it ends the program with exit code 1. The output printed before the failure stays: it was not inside a failure context.

```
Key(N:int)<decides>:int = N > 0

Main()<decides>:void =
    Print("looking for the key")
    K := Key[-1]
    Print("found {K}")
```

Run in the playground

```
looking for the key
```

The analysis is *conservative*. It is syntactic, so it rejects some programs that could never fail. Index `0` of a three-element array literal always exists, but indexing is failable, and the program is rejected anyway:

```
Main():void =
    X := array{1, 2, 3}[0]
    Print("{X}")
```

Run in the playground

Rejected: `this expression can fail, but it is not in a failure context`

This is the usual trade of a static analysis: soundness and predictability in exchange for completeness. A theorem cannot rescue completeness here; deciding exactly which expressions can fail is as hard as deciding termination.

4.6 Limits and readings

- **Only failure is excluded.** A function without `<decides>` may still overflow, call `Err()`, or run forever. Runtime errors are discussed in Chapter 8.
- **Soundness is relative to the declarations.** `Eval.no_fail` assumes `EffectSound`, which is checked, not assumed, for programs from the front end. A Verse Core program built by hand could lie about a function, and then the theorem would not apply to it.
- **Failable primitives are fixed by the semantics** (`Prim.failable`): the comparisons, indexing and indexed update, the division family (`Floor`, `Ceil`, `Quotient`, `Mod`, which fail when the divisor is zero), `?`, and the always-failing `false?`; among heap operations, only casts. Array methods such as `Find` and `Slice` are elaborated into code built from these. That division by zero fails, rather than raising an error, follows the Book of Verse.

Exercises

1. Compute mayFail for the Verse Core of `Safe` in the opening program. (Use `openverse core` to see it.) Which subterms are ignored?
2. The analysis ignores the condition of `ite`. Show that it would be *unsound* to ignore the `then` branch instead: give an expression e that may fail by the real definition, for which the modified analysis says it may not, and a derivation of fail for it.
3. Why does the call case of the proof not need `<decides>` callees to be handled at all? Write the case for a call to a `<decides>` function.
4. Prove that `Prog.checkEffectSound P = true` implies `P.EffectSound`, using the Lean lemma `List.all_eq_true`.
5. (*) Design an analysis $\mathrm{mustFail}_P(e)$ that is sound for the converse property: if $\mathrm{mustFail}(e)$ then every finished evaluation of e fails. What is $\mathrm{mustFail}(\mathsf{ite}\ c\ t\ e)$?
6. (*) The theorem is proved by induction on derivations and so says nothing about diverging runs. Can a diverging run "fail"? Explain why the question does not arise in this semantics.

CHAPTER 5

Well-Typed Programs Do Not Go Wrong

5.1 An opening problem

A value of a subclass can stand wherever its superclass is expected, and so can an array of them:

```
shape := class<abstract>:
    Area():int

square := class(shape):
    Side:int
    Area<override>():int = Side * Side

rect := class(shape):
    W:int
    H:int
    Area<override>():int = W * H

Total(Shapes:[]shape):int =
    var Sum:int = 0
    for (S : Shapes) { set Sum += S.Area() }
    Sum

Main():void =
    Squares:[]square = array{square{Side := 2}, square{Side := 3}}
    Print("{Total(Squares)}")
    Mixed:[]shape = array{square{Side := 1}, rect{W := 2, H := 5}}
    Print("{Total(Mixed)}")
    for (S : Mixed, Sq := square[S]) { Print("a square of side {Sq.Side}") }
```

Run in the playground

```
13
11
a square of side 1
```

`Total` expects a `[]shape` and receives a `[]square`, with no conversion at run time. The last line *casts* each shape to `square`; the cast fails for the rectangle, and the `for` clause skips it. Is this safe? In Java, array covariance is famously unsound: storing a `rect`

into a `square[]` seen as a `shape[]` fails only at run time, with an `ArrayStoreException`. Here, does any typed program ever apply an operation to a value of the wrong shape?

Milner's slogan, "well-typed programs do not go wrong", is a theorem about a particular language. This chapter states and proves it for Verse Core, with classes, subtyping, a heap and generic functions, and explains why Verse's covariant arrays are sound where Java's are not.

5.2 Types, values, heaps

Verse Core types are

$$\tau \;::=\; \mathit{int} \mid \mathit{logic} \mid \mathit{char} \mid \mathit{float} \mid [\,]\tau \mid \mathit{tuple}(\tau_1, \ldots, \tau_n) \mid ?\tau \mid \alpha_j \mid \mathit{ref}\; C\; \vec{\tau}$$

where `void` is the empty tuple, `string` is $[\,]\mathit{char}$, α_j is the j-th type parameter of a generic function (Chapter 6), and $\mathit{ref}\; C\; \vec{\tau}$ is a reference to an object whose class is one of the set C of class tags, with type arguments $\vec{\tau}$. A surface class type such as `shape` elaborates to the set of the *concrete* classes at or below it, so `shape` is $\mathit{ref}\ \{\mathit{square}, \mathit{rect}\}\ [\,]$ (a tag is a class's index in definition order).

Subtyping $\tau \le \tau'$ is a decidable relation (`VTy.sub`): references are ordered by inclusion of class sets (and prefixes of type arguments), arrays, options and tuples are covariant, and otherwise a type is below only itself.

$$\frac{C \subseteq C' \quad \vec{\tau}' \text{ prefix of } \vec{\tau}}{\mathit{ref}\; C\; \vec{\tau} \le \mathit{ref}\; C'\; \vec{\tau}'} \qquad \frac{\tau \le \tau'}{[\,]\tau \le [\,]\tau'} \qquad \frac{\tau \le \tau'}{?\tau \le ?\tau'}$$

Values are typed relative to a heap typing Ξ, a list assigning each heap address a class tag and type arguments. We write $\Xi \models v : \tau$ for `hasTy Ξ v τ = true`. Integers have type int, an array has type $[\,]\tau$ when all its elements have type τ, and a reference ref a has type $\mathit{ref}\; C\; \vec{\tau}$ when $\Xi(a)$ is a class in C whose type arguments extend $\vec{\tau}$:

```
def hasTy (Ξ : HeapTy) : Value → VTy → Bool
  | .int _, .int => true
  | .logic _, .logic => true
  | .char c, .char => decide (c < 256)
  | .float _, .float => true
  | .arr vs, .arr t => allTy Ξ vs.toList t
...
  | .ref a, .ref ts args => match Ξ[a]? with
    | some (t, args') => ts.contains t && args.isPrefixOf args'
    | none => false
  | _, _ => false
```

A heap h is **well typed** under Ξ, written HeapOK $S\ \Xi\ h$, when Ξ has one entry per

object, and each object's fields have the types that its class declares in the signature S, instantiated at its type arguments (which agree with any its class fixes, as Chapter 6 explains). A frame is well typed, FrameOK $\Xi\ \Gamma\ fr$, when slot i holds a value of type Γ_i.

5.3 The typing judgment

Expressions are typed by $S;\Gamma;\rho \vdash e : \tau$ (`Typed S Γ ρ e τ`): in a program with signatures S, in a function whose slots have types Γ and whose result type is ρ, expression e has type τ. A selection of the rules:

$$\frac{\Gamma_i = \tau}{S;\Gamma;\rho \vdash \mathsf{local}\ i : \tau}\ \textsc{T-Local} \qquad \frac{S;\Gamma;\rho \vdash e : \tau \quad \tau \le \tau'}{S;\Gamma;\rho \vdash e : \tau'}\ \textsc{T-Sub}$$

$$\frac{S;\Gamma;\rho \vdash c : \sigma \quad S;\Gamma;\rho \vdash t : \tau \quad S;\Gamma;\rho \vdash e : \tau}{S;\Gamma;\rho \vdash \mathsf{ite}\ c\ t\ e : \tau}\ \textsc{T-Ite}$$

$$\frac{S;\Gamma;\rho \vdash e : \rho}{S;\Gamma;\rho \vdash \mathsf{ret}\ e : \tau}\ \textsc{T-Ret} \qquad \frac{}{S;\Gamma;\rho \vdash \mathsf{brk} : \tau}\ \textsc{T-Brk}$$

$$\frac{S;\Gamma;\rho \vdash e : \mathit{ref}\ C\ \vec{\tau} \quad \text{field } k \text{ has type } \tau \text{ in every class of } C}{S;\Gamma;\rho \vdash \mathsf{heap}\ (\mathit{get}\ k)\ [e] : \tau}\ \textsc{T-Get}$$

Two features stand out. In T-Ite the condition may have *any* type σ: in Verse a condition is not a Boolean, it is anything that may fail. And `return` and `break` have every type, since they never produce a value where they stand; ρ records what `return` must deliver.

5.4 The theorem

The outcome of a typed expression is described by a predicate ResOK $\Xi\ \tau\ \rho\ r$:

```
def ResOK (Ξ : HeapTy) (τ ρ : VTy) : Res → Prop
  | .ok v => hasTy Ξ v τ = true
  | .ret v => hasTy Ξ v ρ = true
  | .err k => k ≠ .type
  | .fail => True
  | .brk => True
```

A success has the expression's type; a `return` carries the function's result type; an error is *not a type error*; failure and `break` are unconstrained. The third line is where "does not go wrong" lives.

Type safety (`Eval.typed`, `Runs.typed`)

Let P be typed with signatures S (ProgTyped P S). If

$$P \vdash \langle s, e\rangle \Downarrow \langle r, s'\rangle, \qquad S;\Gamma;\rho \vdash e : \tau,$$

$$\text{FrameOK } \Xi\ \Gamma\ s.\mathit{fr}, \qquad \text{HeapOK } S\ \Xi\ s.\mathit{heap},$$

then there is a heap typing $\Xi' \sqsupseteq \Xi$ (an extension: Ξ is a prefix of Ξ') with

$$\text{FrameOK } \Xi'\ \Gamma\ s'.\mathit{fr}, \qquad \text{ResOK } \Xi'\ \tau\ \rho\ r, \qquad \text{HeapOK } S\ \Xi'\ s'.\mathit{heap}.$$

For whole programs: if `Main` has type $() \to \tau$, then every run has $r \neq \text{err}\ \mathit{type}$, ends in a well-typed heap, and a successful run returns a value of type τ.

```
@Eval.typed : ∀ {P : Prog} {S : Sigs},
  ProgTyped P S →
    ∀ {s : St} {e : Expr} {r : Res} {s' : St} {Γ : List VTy} {ρ τ : VTy}
        {Ξ : HeapTy},
      Eval P s e r s' →
        Typed S Γ ρ e τ →
          FrameOK Ξ Γ s.fr →
            HeapOK S Ξ s.g.heap →
              ∃ Ξ', FrameOK Ξ' Γ s'.fr ∧ ResOK Ξ' τ ρ r ∧
                HeapOK S Ξ' s'.g.heap ∧ Ξ <+: Ξ'
```

`Ξ <+: Ξ'` is Lean's notation for "is a prefix of". The theorem is *preservation* in big-step form, and here it also does the work of *progress*. In a small-step semantics, progress says a typed term is a value or can step. In big-step form that is not implied by preservation in general, because a stuck configuration simply has no derivation. It is in this semantics because, as Chapter 1 noted, there are no stuck configurations: every ill-shaped step derives an explicit error, so "goes wrong" means "derives err *type*", which ResOK excludes.

```
@Runs.typed : ∀ {P : Prog} {S : Sigs},
  ProgTyped P S →
    ∀ {τ : VTy} {r : Res} {out : List Nat},
      S.fns[P.main]? = some ([], τ) →
        Runs P r out →
          r ≠ Res.err ErrKind.type ∧
            ∃ fr heap,
              Eval P Prog.initSt (Expr.call P.main []) r
                  { fr := fr, g := { out := out, heap := heap } } ∧
                ∃ Ξ, HeapOK S Ξ heap ∧
                  ∀ (v : Value), r = Res.ok v → hasTy Ξ v τ = true
```

Supporting it is the soundness of subsumption:

```
@hasTy_sub : ∀ {Ξ : HeapTy} (t u : VTy) (v : Value),
  t.sub u = true → hasTy Ξ v t = true → hasTy Ξ v u = true
```

5.5 The proof

Subsumption is sound (`hasTy_sub`), by induction on t. For references: if $\Xi(a) \in C$ and $C \subseteq C'$ then $\Xi(a) \in C'$, and a prefix of a prefix is a prefix. For arrays: if every element has type τ and $\tau \leq \tau'$, every element has type τ' by the induction hypothesis. This is the whole reason covariant arrays are sound in Verse: a Verse array is an immutable *value*. `set A[i] = x` does not write into a shared array; it builds a new array and assigns it to the variable `A`, whose declared type Γ_A the assignment rule checks x against. Java's unsoundness needs an array *object* that two variables of different static types share; Verse has no such thing. (Objects are shared, and their fields are typed invariantly: `set` on a field checks the field's declared type.)

Preservation. openverse does not prove `Eval.typed` by induction on the derivation but, as in Chapter 2, through the interpreter: by strong induction on the *fuel*, for the statement "if `eval P n s e = some (r, s')` and the hypotheses hold, then the conclusion holds", and then transports the result to `Eval` with `eval_iff`. The reason is the call case. A callee's body is an initialization prefix followed by the user's code b; the frame satisfies Γ only *after* the prefix has run, so the induction hypothesis is needed for b evaluated from that later state, at whatever smaller fuel remains (`eval_initPrefixE`). That evaluation is not a subderivation of anything in hand, so structural induction on derivations does not reach it, while strong induction on fuel does. The ingredients:

1. **Inversion up to subsumption.** A typing derivation may end with any number of T-Sub steps. The proof peels them off: if $e : \tau$ then $e : \tau_0$ by a non-subsumption rule, for some $\tau_0 \leq \tau$; it proves the conclusion at τ_0, then lifts it with `hasTy_sub` (`ResOK_sub`).
2. **Heap typings only grow.** Allocation appends an object, and the proof appends its class and type arguments to Ξ. A field write replaces a field with a value of the field's type, leaving Ξ unchanged. Values typed under Ξ remain typed under any extension (`hasTy_mono`), which is why the conclusion is stated with $\Xi \sqsubseteq \Xi'$.
3. **Primitives are type safe** (`prim_typed`): a primitive applied to values of its argument types returns ResOK of its result type. This is a finite case analysis over the primitive table.
4. **Calls** instantiate the callee's signature with a substitution θ for its type parameters (Chapter 6), and the callee's body is typed with its own Γ, which the new frame satisfies because the arguments have the parameter types and the other slots are initialized first (the elaborator emits an initialization prefix; `initPre-`

`fixE`).

5. **Rollback is harmless.** In ITE-FAIL the `else` branch runs from the original state s, typed under the original Ξ; whatever Ξ the failed condition reached is discarded with its state.

`Runs.typed` instantiates preservation at the call of `Main`, from the initial state, whose frame and heap are empty. When the program has classes, there is one more step. Class-typed variables start as *prototype* objects (one per non-parametric concrete class), so the Verse Core `Main` is a prelude that allocates the prototypes and then calls the user's `Main`. `Runs.typed` evaluates the prelude (`Eval.of_protoPrelude`) and applies preservation to that inner call, from the heap of prototypes, which ProgTyped requires to be well typed (`PreludeOK`).

The type checker is sound. Where does ProgTyped P S come from? openverse runs a bidirectional type checker, `tc`, on every program the front end emits, together with a certificate of each function's slot types. `tc` either returns a type or fails, and `tc_sound` proves that a returned type is derivable:

```
@tc_sound : ∀ {S : Sigs} {Γ : List VTy} {ρ : VTy} {e : Expr} {m : Option VTy}
    {τ : VTy},
  tc S Γ ρ e m = some τ → Typed S Γ ρ e τ
```

`m` is the optional *expected* type that bidirectional checking pushes down. A corollary, `Prog.checkTypes_sound`, lifts the result to whole programs. Only soundness is proved: a correct program that `tc` cannot type is rejected, which is safe.

5.6 In code

The front end reports type errors before anything runs:

```
Main():void =
    X:int = "five"
```

Run in the playground

Rejected: `expected int, found string`

Options are not their contents; a `?string` must be taken apart in a failure context:

```
Main():void =
    Name:?string = option{"Ada"}
    N:int = Name
```

Run in the playground

Rejected: `expected int, found ?string`

And updating an array element rebuilds the variable's value rather than writing into a shared array, which is why the array types can be covariant:

```
Main():void =
    var A:[]int = array{1, 2, 3}
    if (set A[1] = 20) { Print("{A.Length}") }
    if (V := A[1]) { Print("{V}") }
```

Run in the playground

```
3
20
```

(`set A[1] = …` fails when the index is out of range, so it, too, must stand in a failure context.)

5.7 Limits and readings

- **Type errors are excluded; other errors are not.** Overflow and `Err()` remain possible, and so do slot, call and control errors unless the program also passes `checkWellScoped` and `checkControl`, which Chapter 8 adds.
- **The trusted part is the statement.** The typing rules and hasTy are definitions. That they match Verse's surface type system is the elaborator's business, and it is not proved: what is proved is that whatever the elaborator emits, if it passes `checkTypes`, does not go wrong.
- **Prototypes.** Class-typed slots start as shared prototype objects rather than as an uninitialized value. Verse code always assigns such a variable before reading it, so the prototypes are never observed; they exist to make every slot hold a value of its type from the start.
- **Comparability.** `=` on values of different surface types that coincide in Verse Core (an enum and an `int`, say) is rejected by the front end, a reading documented in `docs/LANGUAGE.md`; the Core rule types `=` at the type of its left operand.
- **Types the front end lowers.** `rational`, `any` and refinement types (`type{_X:int where 0 <= _X, _X <= 120}`) have no Verse Core type of their own: a rational is the tuple of its numerator and denominator in lowest terms, an `any` is `void`, and a refined `int` or `float` is just its base, with the cast `age[X]` a comparison that may fail. The theorem covers these Core images. That a refined value stays within its bounds, or that a rational stays in lowest terms, is the front end's business: tested, not proved. (`float` is different: it is a Core type, with its own case in hasTy.)

Exercises

1. Prove `hasTy_sub` for the option case on paper.
2. Show that the rule T-Ite with a *Boolean* condition would make no sense in Verse Core: which Verse Core conditions in this book have type `void`? Which have type `int`?

3. Write the case of preservation for assign i e. Where is `FrameOK` used? Why does the slot write itself not extend the heap typing (take Ξ' from the induction hypothesis for e, which may allocate)?
4. Java's array store check exists because arrays are mutable and shared. Suppose Verse Core gained a primitive *arrWrite* that mutates an array *in place*, visible through every alias. Give a typed program that goes wrong, and say which step of the proof breaks.
5. (*) State the fuel-indexed version of preservation: for every n, if `eval P n s e = some (r, s')` and the hypotheses hold, then ... Show that, by `eval_iff`, it is equivalent to `Eval.typed`, so it says nothing more about diverging programs. Why is it nonetheless the version openverse proves first? What would a statement about diverging runs need (for example, an interpreter that returns its state when the fuel runs out)?

CHAPTER 6

Checked Once, Correct Everywhere

6.1 An opening problem

A generic function is written once and used at many types:

```
Find(Items:[]t, Target:t where t:subtype(comparable))<decides>:int =
    var Result:int = -1
    for (I -> X : Items, X = Target, Result < 0):
        set Result = I
    Result >= 0
    Result

Main():void =
    if (I := Find[array{"map", "key", "gem"}, "gem"]) { Print("gem at {I}") }
    if (J := Find[array{3, 1, 4}, 4]) { Print("4 at {J}") }
    if (not Find[array{3, 1, 4}, 9]) { Print("no 9") }
```

Run in the playground

```
gem at 2
4 at 2
no 9
```

`Find` is used at `string` and at `int`. There are two classical ways to make such code safe. A *template* system (C++) type-checks each instantiation separately, after substituting the type: errors appear at use sites, and code is duplicated. A *parametric* system (ML, Haskell, Java generics) checks the definition once, with `t` opaque, and guarantees that every instantiation is then fine. Verse is parametric: the body of `Find` is checked once, and operations not valid for every `t` are rejected even if `Find` is never called.

For a programmer, this means a library function that compiles cannot break at a new type. For a logician, it is a statement about a typing judgment under substitution, the *substitution lemma*, one of the workhorses of type theory. openverse proves it for Verse Core, and its proof of type safety uses it at every call.

6.2 Type variables and substitutions

A generic function's types may mention **type variables** $\alpha_0, \alpha_1, \ldots$ (`VTy.var j`): its j-th type parameter. A **substitution** $\theta : \mathbb{N} \to \mathrm{VTy}$ replaces each variable by a type, and acts on types homomorphically:

$$\theta(\alpha_j) = \theta\, j, \qquad \theta([\,]\tau) = [\,]\theta(\tau), \qquad \theta(\mathit{tuple}(\vec{\tau})) = \mathit{tuple}(\theta\vec{\tau}), \qquad \theta(\mathit{int}) = \mathit{int}, \ \ldots$$

We write $\theta\Gamma$ for the pointwise image of a list of slot types. The typing rule for calls is where substitutions enter: a call may instantiate the callee's signature $(\vec{p}, \tau)$ with *any* θ, provided the arguments have the instantiated parameter types.

$$\frac{S.\mathit{fns}_f = (\vec{p}, \tau) \quad S;\Gamma;\rho \vdash \vec{e} : \theta\vec{p}}{S;\Gamma;\rho \vdash \mathsf{call}\, f\ \vec{e} : \theta\tau}\ \textsc{T-Call}$$

```
/-- A call instantiates the callee's type variables with any `θ`. -/
| call {f es ps τ} (θ : Nat → VTy) : S.fns[f]? = some (ps, τ) →
    TypedL S Γ ρ es (ps.map (VTy.subst θ)) →
    Typed S Γ ρ (.call f es) (τ.subst θ)
```

No *value* has a type variable as its type: $\Xi \models v : \alpha_j$ is always false. A value can have a type *mentioning* a variable only where the variable is not inhabited: the empty array has type $[\,]\alpha_j$, and the empty option type $?\alpha_j$.

6.3 The theorem

The substitution lemma (`Typed.subst`)

If the signatures S are closed (Closed S: the prototype objects' type arguments are closed, and so are the types that classes fix their type arguments to), then for every substitution θ:

$$S;\Gamma;\rho \vdash e : \tau \implies S;\theta\Gamma;\theta\rho \vdash e : \theta\tau.$$

```
@Typed.subst : ∀ {S : Sigs} {Γ : List VTy} {ρ : VTy},
  S.Closed →
    ∀ (θ : Nat → VTy) {e : Expr} {τ : VTy},
      Typed S Γ ρ e τ →
        Typed S (List.map (VTy.subst θ) Γ) (VTy.subst θ ρ) e (VTy.subst θ τ)
```

The *same expression* e is typed on both sides: nothing is recompiled or re-elaborated. A typing derivation with variables is a proof schema, and every instance of the schema

is a proof. That is the logical content of "checked once".

The hypothesis Closed looks technical, and it is. It has two halves. ClosedProtos: literals in Verse Core may be references to the prototype objects that exist before `Main` (Chapter 5), and their types must not mention variables, or a literal typed at α_j-dependent arguments could not be retyped. ClosedPins: a class may fix its parent's type arguments, as `int_box := class(box(int))` fixes `box`'s to `int`; S records these *pins*, and they must be closed types. The front end always satisfies both.

6.4 The proof

By induction on the typing derivation, mutually with the lemma for lists of expressions (`TypedL.subst`). Most cases rebuild the same rule from the induction hypotheses, because the rules are *uniform* in the types: they never inspect whether a type is a variable. Three cases need a lemma, and the classes that fix type arguments need an argument of their own.

Literals. T-Lit has the premise hasTy $S.\mathit{protos}\ v\ \tau$. We need hasTy $S.\mathit{protos}\ v\ (\theta\tau)$: typing of values is stable under substitution (`hasTy_subst`). By induction on v and τ: an integer has type $\mathit{int} = \theta(\mathit{int})$; the empty array has every array type; a reference's type arguments are a prefix of a prototype's, which are closed, so θ does not change them. A variable itself is never the type of a value, so that case is vacuous.

Subsumption. T-Sub has the premise $\tau \le \tau'$; we need $\theta\tau \le \theta\tau'$. Subtyping is stable under substitution (`VTy.sub_subst`), by structural recursion on τ (subtyping is a Boolean function, `VTy.sub`, not an inductive relation); a variable is below only itself, and $\theta\alpha_j \le \theta\alpha_j$ by reflexivity.

Calls. The derivation ends in T-Call with *its own* substitution θ': the arguments have types $\theta'\vec{p}$ and the call has type $\theta'\tau$. The induction hypothesis turns the argument typings into typings at $\theta(\theta'\vec{p})$. Composition of substitutions, $\theta \circ \theta' := \lambda j.\ \theta(\theta'\ j)$, satisfies

$$\theta(\theta'\sigma) = (\theta \circ \theta')\,\sigma \qquad \text{for every type } \sigma$$

(`VTy.subst_comp`), so the arguments have the types $(\theta \circ \theta')\vec{p}$, and T-Call with the substitution $\theta \circ \theta'$ gives the call the type $(\theta \circ \theta')\tau = \theta(\theta'\tau)$, as required:

```
| _, _, .call θ' hs hes => by
  have h := TypedL.subst hS θ hes
  rw [List.map_map] at h
  have h' := Typed.call (fun j => (θ' j).subst θ) hs
    (by rw [← VTy.substL_comp]; simpa [Function.comp_def] using h)
  rwa [← VTy.subst_comp] at h'
```

This case is why the rule T-Call quantifies over *every* θ, rather than fixing one per call

site: the set of typings must be closed under composition. The class of substitutions being all functions $\mathbb{N} \to \mathrm{VTy}$ makes it so.

Pins. Creating an object requires its type arguments to agree with its class's pins. The pins are closed, so θ leaves them alone and the agreement survives (`VTy.pinsOK_subst`); that is where ClosedPins is used. The call of an override in a class with pins goes further, and looks at the types themselves: it is typed under a *refinement* μ, the most general substitution that makes the receiver's type arguments agree with the pins (`int` for `box(t)`'s `t`). After θ, either some ρ' makes the instance agree, so that $\rho' \circ \theta$ makes the original arguments agree and therefore factors through the most general one, $(\rho' \circ \theta) \circ \mu = \rho' \circ \theta$ (`PinRefines.factor`), which carries the typing over to the instance's own refinement; or none does, and the call is typed as unreachable (`Typed.dispatch_none`), as it then is at every instance.

6.5 How type safety uses it

The proof of `Eval.typed` (Chapter 5) reaches a call of a generic function f at instance θ. The callee's body was typed once, generically, with slot types Γ_f and result type ρ_f mentioning variables. The substitution lemma retypes it with $\theta\Gamma_f$ and $\theta\rho_f$. The new frame holds the actual arguments and `void` in the other slots; after the body's initialization prefix has run (`eval_initPrefixE`), it satisfies $\theta\Gamma_f$, and the induction hypothesis applies to the rest of the body at the instance. No step of the proof ever looks inside a type variable.

There is one more obstacle, and it is a design problem rather than a proof problem. Every slot of a Verse Core frame must hold a value of its type from the start (`FrameOK`), and the elaborator initializes local slots before the body runs. But a slot of type α_j has no default value: there is no value of an unknown type. openverse's solution is to pass **one hidden argument per type parameter**, a value of that type, chosen by the caller, and initialize such slots from it. For

```
Keep(X:t where t:type):t =
    var Y:t = X
    Y

Main():void =
    A := Keep(5)
    Print("{A}")
```

Run in the playground

```
5
```

`openverse core` shows that `Keep` has arity 2, and that the call is call 1 [lit 5, lit 0]: the second argument, 0, is the witness for $t = int$. The body begins with assign 2 (local 1), initializing `Y`'s slot from the witness, before the user's assign 2 (local 0). The witness

is never observed: Verse code assigns every variable before reading it. The typing rules allow exactly such initializers (`DefaultE`: a literal, a parameter, a tuple of these, or a new object), and the semantics needs no change at all: hidden arguments are ordinary arguments.

6.6 In code

Parametric structs work the same way. `pair(t, u)` is a family of struct types, and `Swap` is checked once:

```
pair(t:type, u:type) := struct:
    First:t
    Second:u

Swap(P:pair(t, u) where t:type, u:type):pair(u, t) =
    pair(u, t){First := P.Second, Second := P.First}

Main():void =
    P := pair(int, string){First := 7, Second := "seven"}
    Q := Swap(P)
    Print("{Q.First} is {Q.Second}")
```

Run in the playground

```
seven is 7
```

Checking once means rejecting operations that are not valid at *every* type, even when every actual use would be fine:

```
Twice(X:t where t:type):t = X + X

Main():void = Print("{Twice(2)}")
```

Run in the playground

Rejected: `no operator '+' for t and t`

Equality is available only under the constraint `t:subtype(comparable)`, which restricts the instances to types with a decidable equality:

```
Same(A:t, B:t where t:type)<decides>:void = A = B

Main():void = Print("unused")
```

Run in the playground

Rejected: `an operand of '=' must be comparable`

Because a generic function is a single Verse Core function, it can call itself at a *larger* type, which template expansion cannot do (it would generate infinitely many instances). Here `Depth` calls itself at `tuple(t, t)`:

```
Depth(X:t, N:int where t:type):int =
    if (N <= 0) then 0 else 1 + Depth((X, X), N - 1)

Main():void = Print("{Depth(1, 5)}")
```

Run in the playground

```
5
```

The fifth recursive call has `t` = a tuple nested five deep. Each call instantiates T-Call with a different θ; the substitution lemma does not care.

Classes can be parametric too, and a class may fix its parent's type argument and override the parent's methods at that type. This is where the pins of the theorem come from:

```
box(t:type) := class:
    V:t
    Get():t = V

int_box := class(box(int)):
    Get<override>():int = V + 1

GetAny(B:box(t) where t:type):t = B.Get()

Main():void =
    Print("{GetAny(box(int){V := 41})}")
    Print("{GetAny(int_box{V := 41})}")
    Print(GetAny(box(string){V := "s"}))
```

Run in the playground

```
41
42
s
```

`GetAny` is checked once, with `t` opaque, yet on an `int_box` it runs `int_box`'s `Get`, whose body `V + 1` makes sense only at `int`. Verse Core has no method tables: the call tests the receiver's class tag, one *isTag* test for each implementation of `Get` (here two), and calls the one that matches (`openverse core` shows the chain). The branch that calls `int_box`'s `Get`, which returns an `int` where `GetAny` promises a `t`, is typed under the refinement $\mu = [t \mapsto int]$, the most general substitution that agrees with the pin. At the instance $t = string$ no substitution agrees with it, and the branch is typed as unreachable (`Typed.dispatch_none`). It is: a `box(string)` is never an `int_box`, because an object's type arguments must agree with its class's pins.

The refinement binds variables to closed types and nothing else. A class that repeats an argument, `dup(t) := class(pair(t, t))`, would need a refinement that makes two variables equal, and its overrides are refused:

```
pair(t:type, u:type) := class:
    A:t
    B:u
    Get():t = A

dup(t:type) := class(pair(t, t)):
    Get<override>():t = B

Main():void = Print("unused")
```

Run in the playground

Rejected: `so it cannot override Get`

6.7 Limits and readings

- **The lemma is about typing, not about behavior.** It does not say that a generic function *behaves* uniformly at all types (Reynolds' parametricity, the "theorems for free"). openverse's hidden witnesses are never observed, so uniform behavior is plausible, but it is not a theorem here.
- **Inference is unverified.** The front end infers each call's type arguments by first-order matching against the expected result type and then the arguments (`docs/LANGUAGE.md`). It need not be complete or even correct: the checker rechecks each call against the instantiated signature, and only that check is part of the trusted argument.
- **Restrictions of the subset**: no named parameters, generic `Main`, or explicit type arguments for generic functions; several constraints on one type parameter are not supported (nor, at the time of writing, in the Book). A class may override only where it fixes type arguments to closed types: not where it repeats them, as `dup` does, or adds type parameters of its own.

Exercises

1. Prove `VTy.subst_comp`, $\theta(\theta'\sigma) = (\theta \circ \theta')\sigma$, by induction on σ. Which case uses the definition of $\circ$?
2. Show by an example why the lemma needs ClosedProtos: give a prototype with an open type argument and a literal whose typing does not survive a substitution.
3. Where in the proof of `Eval.typed` would *monomorphization* (copying each generic function once per instance) replace the substitution lemma? What does it cost, and why can it not handle `Depth`?
4. The identity substitution is $\lambda j.\, \alpha_j$ (`VTy.var`). Prove that it acts as the identity on types (`VTy.subst_id`), and use it to derive the non-generic case of T-Call

from the generic one.

5. Predict whether the front end accepts this function, then check:

```
Second(Items:[]t, Fallback:t where t:type):t =
    if (X := Items[1]) then X else Fallback

Main():void = Print(Second(array{"a", "b"}, "none"))
```

Run in the playground

6. Give `GetAny` the receiver type `box(string)` in place of `box(t)` and read `open-verse core`: which class has no *isTag* test any more? Why may the front end leave it out, and which rule would type its branch if it were there?
7. (*) State Reynolds' parametricity for a function of type $\forall \alpha.\ \alpha \to \alpha$ in Verse Core, and explain what the hidden witness argument would have to satisfy for it to hold.

CHAPTER 7

Look but Do Not Touch

7.1 An opening problem

Values in Verse are immutable, but objects are not: a class may have `var` fields, an object is shared by every reference to it, and a write through one reference is seen through all the others. Rollback extends to the heap:

```
chest := class:
    var Gold:int = 0

Deposit(C:chest, N:int)<transacts>:void =
    set C.Gold += N

Peek(C:chest)<reads>:int = C.Gold

Main():void =
    Mine := chest{}
    Alias := Mine
    Deposit(Mine, 5)
    Print("Alias sees {Peek(Alias)}")
    if (Deposit(Mine, 100), Peek(Mine) > 1000):
        Print("rich")
    Print("after the failed attempt: {Peek(Mine)}")
```

Run in the playground

```
Alias sees 5
after the failed attempt: 5
```

Mutation makes reasoning harder: to know what a call does, one must know which objects it may change. Verse answers with **heap effects**, specifiers on a function's signature: `<computes>` (no access to mutable state), `<reads>` (may read `var` fields), `<allocates>` (may construct objects with mutable state), `<writes>` (may set `var` fields), and `<transacts>` (all of these). `Peek` above is `<reads>`. The programmer wants to know that calling `Peek` cannot change the chest. The logician recognizes a *frame property*: an analysis on the syntax implies that a whole region of the state is untouched. This chapter proves it.

7.2 The analysis

The front end checks each body against its function's declared effects, and records in Verse Core, for each function, a single bit: whether it may `writes`. A function declared `<computes>`, `<reads>` or `<allocates>` (or a combination without `<writes>`) has $\mathit{writes} = \bot$. In Verse Core the only operation that changes an existing object is the field write heap $(\mathit{set}\ k)\ [r, v]$, and the analysis is:

Writes the heap

Write $W(e)$ for $\mathrm{writesHeap}_P(e)$. Then

$$\begin{aligned} W(\mathsf{heap}\ o\ \vec{e}) &= [o = \mathit{set}\ k] \vee \bigvee_i W(e_i) \\ W(\mathsf{call}\ f\ \vec{e}) &= \mathit{writes}(P.f) \vee \bigvee_i W(e_i) \\ W(\mathsf{ite}\ c\ t\ e) &= W(c) \vee W(t) \vee W(e) \\ W(\mathsf{for}\ i\ a\ f\ b) &= W(a) \vee W(f) \vee W(b) \end{aligned}$$

where a, f, b are the array, filter and body of the `for`, and the other constructs pass their subexpressions' values up.

Compare the `ite` line with Chapter 4's mayFail, which *ignored* the condition. Here the condition counts. The two analyses look at the same construct from opposite sides: a failure inside the condition never escapes it, so it cannot make the `if` fail; but a write inside the condition escapes whenever the condition *succeeds*, since success commits. Rollback undoes writes only on failure.

```
| .heap op es => op.writes || Expr.writesHeapList P es
| .call f es =>
  (match P.fns[f]? with
    | some fn => fn.writes
    | none => false) || Expr.writesHeapList P es
| .ite c t e => c.writesHeap P || t.writesHeap P || e.writesHeap P
```

As with `<decides>`, the analysis of a call trusts the callee's declaration, and soundness needs the declarations to be honest:

$$\mathrm{WriteSound}(P) \ :\Longleftrightarrow\ \forall \mathit{fn} \in P.\mathit{fns}.\ \neg\, \mathit{fn}.\mathit{writes} \Rightarrow \neg\, \mathrm{writesHeap}_P(\mathit{fn}.\mathit{body}).$$

7.3 What "untouched" means

A function that does not write may still *allocate*, so its heap may grow. The right relation between the global state before and after is that every object that existed is still there, at the same address, with the same fields:

Keeps

$$g \preceq g' \;:\Longleftrightarrow\; \forall a\, o.\;\; g.heap[a] = \text{some } o \;\Rightarrow\; g'.heap[a] = \text{some } o$$

(`Glob.keeps g g'`). It is reflexive and transitive: a preorder.

```
def Glob.keeps (g g' : Glob) : Prop :=
  ∀ (a : Nat) (o : Obj), g.heap.toList[a]? = some o →
    g'.heap.toList[a]? = some o
```

Note what $\preceq$ does *not* constrain: the output (a non-writing function may print), new objects, and the frame.

7.4 The theorem

Heap frame property (`Eval.no_writes`, `Prog.accepted_no_writes`)

If WriteSound(P), then for all s, e, r, s':

$$P \vdash \langle s,\, e\rangle \Downarrow \langle r,\, s'\rangle \;\wedge\; \neg\,\text{writesHeap}_P(e) \;\Longrightarrow\; s.g \preceq s'.g.$$

For a program accepted by the front end, a call of a function f declared without `writes`, whose arguments do not write, keeps every existing object.

```
@Eval.no_writes : ∀ {P : Prog},
  P.WriteSound →
    ∀ {s : St} {e : Expr} {r : Res} {s' : St},
      Eval P s e r s' → Expr.writesHeap P e = false → s.g.keeps s'.g
```

```
@Prog.accepted_no_writes : ∀ {P : Prog},
  P.check = true →
    ∀ {s : St} {f : Nat} {es : List Expr} {r : Res} {s' : St} {fn : Fn},
      P.fns[f]? = some fn →
        fn.writes = false → Expr.writesHeapList P es = false →
          Eval P s (Expr.call f es) r s' → s.g.keeps s'.g
```

The second theorem has no `WriteSound` hypothesis: it takes `P.check = true`, the Boolean check every accepted program passes (Chapter 8), and derives `WriteSound` from it (`Prog.check_writeSound`). The theorem holds whatever the outcome r: even if the call fails, raises an error or returns early, no existing object has changed.

7.5 The proof

By induction on the derivation, mutually for lists and `for` iterations, assuming WriteSound(P). The global state threads through evaluation, and each case composes the steps with the transitivity of $\preceq$.

- **Leaves** (Lit, Local, Brk) do not change the state: reflexivity.
- **Sequencing and arguments**: $s.g \preceq s_1.g \preceq s_2.g$ by the hypotheses, then transitivity. The same for every rule with several premises evaluated in order.
- **Heap operations.** Since $\neg$ writesHeap, the operation is not a field write. Allocation *pushes* an object at the end of the heap, which keeps every existing address; reads, class tests and casts leave the heap as it is.
- **`print`** changes the output, which $\preceq$ ignores.
- **Calls.** The arguments keep the heap by the hypothesis. The callee does not write, so by WriteSound its body does not write, and the hypothesis for the body's derivation, a premise of Call-Ok, gives that the body keeps the heap. The call's final state takes the caller's frame and the body's global part, so the global part is kept.
- **`ite`.** With Ite-Ok, the condition and then the branch keep the heap (both are analyzed): transitivity. With Ite-Fail, the else branch runs from s, so the hypothesis for it gives $s.g \preceq s'.g$ directly. The failed condition's state is discarded, and the proof never needs to know what it did.

The last case shows rollback and the frame property cooperating: the proof would go through even if writesHeap ignored conditions *in the Ite-Fail case*. What forces the analysis to count them are Ite-Ok, where success commits the condition's writes, and Ite-Stop, where a `return`, `break` or error inside the condition leaves the `if` without rollback, in the condition's final state.

7.6 In code

Rollback of field writes is the opening program. The effect checks are the compile-time half of the story; each forbidden step is named:

```
chest := class:
    var Gold:int = 0

Steal(C:chest)<reads>:void = set C.Gold = 0

Main():void = Steal(chest{})
```

Run in the playground

```
Rejected: Steal sets a var field, which its effects (<reads>) do not allow
```

Calling a writing method from a `<reads>` function is just as forbidden: the analysis of a call uses the callee's declared effects.

```
counter := class:
    var Count:int = 0
    Bump()<transacts>:void = set Count += 1

Look(C:counter)<reads>:void = C.Bump()

Main():void = Look(counter{})
```

Run in the playground

Rejected: `which its effects (<reads>) do not allow`

`<computes>` is stricter than `<reads>`: it may not even *read* a `var` field, since the result would then depend on mutable state.

```
chest := class:
    var Gold:int = 0

Peek(C:chest)<computes>:int = C.Gold

Main():void = Print("{Peek(chest{})}")
```

Run in the playground

Rejected: `Peek reads a var field, which its effects (<computes>) do not allow`

Reading an *immutable* field is not an effect: immutable objects are values as far as effects are concerned, so this `<computes>` function is accepted.

```
point := class:
    X:int
    Y:int

Norm(P:point)<computes>:int = Abs(P.X) + Abs(P.Y)

Main():void = Print("{Norm(point{X := 3, Y := -4})}")
```

Run in the playground

```
7
```

An `<allocates>` function may construct objects with `var` fields, which is compatible with $\preceq$ because new objects go to new addresses:

```
chest := class:
    var Gold:int = 0

Fresh(N:int)<allocates>:chest = chest{Gold := N}

Main():void =
```

```
    C := Fresh(3)
    Print("{C.Gold}")
```

Run in the playground

```
3
```

7.7 Limits and readings

- **Only `writes` has a theorem.** openverse checks `<computes>` and `<reads>` as the front end's discipline, but the proved property is about the `writes` bit: a function without it keeps every existing object. A theorem that a `<computes>` function's result does not depend on the heap would need a relational (two-run) statement; it is an exercise.
- **The frame and the output are not covered.** `var` local variables live in the frame and are not "the heap": a `<computes>` function may use them freely. Printing is not a heap effect either (openverse accepts Verse's other effect specifiers, such as `<converges>`, without checking them).
- **Field writes, constructions and rollback.** A failed failure context undoes the field writes and the constructions it made, as the opening program shows; this is the heap half of the transactional state of Chapter 3. The reference interpreter implements it by copying the heap on the first write or allocation inside a context. The C runtime rolls back slots and output with the journal of `store.c`, which has a bounded machine-checked proof (Chapter 8), and object writes and allocations with a field log in its object store (`objects.c`), which is tested but not proved.

Exercises

1. Prove that $\preceq$ is a preorder. Is it antisymmetric? (Hint: which part of a `Glob` does $\preceq$ not look at? Is it antisymmetric on heaps alone? See `Glob.keeps.prefix`.)
2. Write the Heap-Ok case of the proof for allocation in full, using `Array.push`. Which property of arrays do you need?
3. Explain why writesHeap must count the *filter* of a `for`. Give a program whose filter writes a field and succeeds.
4. Suppose a function is declared `<reads>` and calls a `<decides>` function that writes a field and then fails. Write the program and check it: the front end rejects it. Semantically, the write would be rolled back by the failure, so the call could not change any object. Why is it rejected anyway, and what would an analysis that accepted it have to know?
5. () *State a* non-interference* theorem for `<computes>`: two runs of a `<computes>` function from states that differ only in `var` fields of existing objects

return the same outcome. What analysis would its proof need, beyond writesHeap?

CHAPTER 8

Everything Accepted Is Guaranteed

8.1 An opening problem

The theorems of Chapters 4 to 7 had a hypothesis about the program: `EffectSound`, `ProgTyped`, `WriteSound`. The Verse programs in this book were not written in Verse Core. They were written in surface Verse and translated by openverse's front end, more than five thousand lines of lexer, parser and elaborator that nobody has proved correct. Why should the theorems apply to them?

Here is a program that uses most of what the earlier chapters proved something about: failure, rollback (of a field write, inside `logic{…}`, which commits on success and undoes on failure), a class with a `var` field, a generic function and a `<reads>` function.

```
chest := class:
    var Gold:int = 0

Peek(C:chest)<reads>:int = C.Gold

Count(Items:[]t, Target:t where t:subtype(comparable)):int =
    var N:int = 0
    for (X : Items, X = Target) { set N += 1 }
    N

Main():void =
    C := chest{}
    for (N : array{5, 12, 7}):
        Added := logic{set C.Gold += N; Peek(C) < 20}
        if (not Added?) { Print("too heavy at {N}") }
    Print("gold: {Peek(C)}")
    Print("rubies: {Count(array{"ruby", "opal", "ruby"}, "ruby")}")
```

Run in the playground

The front end accepts it, and the guarantees of this chapter hold for it: whatever happens, it has one outcome and one output, it raises no slot, call, type or control error, its result has its declared type, it cannot fail because `Main` is not `<decides>`, and `Peek` changes no object. The only runtime errors it could raise are integer overflow

and `Err()`.

```
too heavy at 7
gold: 17
rubies: 2
```

The design that makes this possible is called **translation validation**. The front end is not trusted to be correct; each program it emits is checked, by Lean functions, and each check is proved to establish the hypotheses of the theorems. The capstone theorem `Prog.accepted` puts them together.

8.2 The checks

An accepted Verse Core program passes three Boolean checks:

- check(P), the conjunction

$$\begin{aligned}&\text{checkEffectSound}(P) \wedge \text{checkWellScoped}(P)\\ \wedge\ &\text{checkMain}(P) \wedge \text{checkWriteSound}(P),\end{aligned}$$

  where checkMain says that `Main` exists and takes no arguments;
- checkTypes$(P, S, \vec{\Gamma})$, the type checker of Chapter 5, run with signatures S and a certificate $\vec{\Gamma}$ of every function's slot types;
- checkControl(P), which holds when brkSafe$_{\perp}(\mathit{fn.body})$ for every function fn of P.

Each is a computable function, and each has a soundness lemma: Chapter 4's `checkEffectSound` gives `EffectSound`, Chapter 7's `checkWriteSound` gives `WriteSound`, `Prog.checkTypes_sound` gives `ProgTyped`. Two we have not met:

Scoping. checkWellScoped verifies that every slot index in a function's body is below its frame size and every call names an existing function with the right number of arguments. Under it, evaluation never raises a slot or call error and never changes the size of a frame:

```
@Eval.scoped_ok : ∀ {P : Prog},
  P.WellScoped →
    ∀ {s : St} {e : Expr} {r : Res} {s' : St},
      Eval P s e r s' → Expr.scoped P s.fr.length e = true →
        r.scopeErr = false ∧ s'.fr.length = s.fr.length
```

`WellScoped` is also exactly what openverse's C loader validates before it runs a binary, so the C interpreter can use fixed-size frames with no bounds checks of its own. That the C loader really establishes it, for every input buffer up to 4 KiB, is a separate machine-checked proof in Rocq (QuickProof's law `loader_wellscoped`).

Control. brkSafe$_b(e)$, with b recording whether we are inside a `loop`, is true when every `break` is inside a `loop` of the same function body; a `for` body does not count.

Then no `break` escapes:

```
@Runs.no_control : ∀ {P : Prog},
  P.checkControl = true → ∀ {r : Res} {out : List Nat},
    Runs P r out → r ≠ Res.err ErrKind.control
```

8.3 The theorem

Accepted programs (`Prog.accepted`)

Let P pass check, checkTypes with signatures S, and checkControl, and let `Main` have type $() \to \tau$. If Runs $P\ r\ o$, then

1. r is not a slot or call error;
2. r is not a type error;
3. r is not a control error;
4. the final heap is well typed under some Ξ, and if $r = \mathsf{ok}\, v$ then $\Xi \models v : \tau$;
5. if `Main` is not `<decides>`, then $r \neq \mathsf{fail}$;
6. every run of P has outcome r and output o.

So the only runtime errors left are $\mathsf{err}\, \mathit{overflow}$ and $\mathsf{err}\, \mathit{user}$ (`Err()`).

```
@Prog.accepted : ∀ {P : Prog} {S : Sigs} {Γs : List (List VTy)} {τ : VTy},
  P.check = true →
    P.checkTypes S Γs = true →
      P.checkControl = true →
        S.fns[P.main]? = some ([], τ) →
          ∀ {r : Res} {out : List Nat},
            Runs P r out →
              r.scopeErr = false ∧
                r ≠ Res.err ErrKind.type ∧
                  r ≠ Res.err ErrKind.control ∧
                    (∃ fr heap,
                        Eval P Prog.initSt (Expr.call P.main []) r
                            { fr := fr, g := { out := out, heap := heap } } ∧
                          ∃ Ξ, HeapOK S Ξ heap ∧
                            ∀ (v : Value), r = Res.ok v →
                              hasTy Ξ v τ = true) ∧
                      (∀ (fn : Fn), P.fns[P.main]? = some fn →
                          fn.decides = false → r ≠ Res.fail) ∧
                        ∀ (r' : Res) (out' : List Nat), Runs P r' out' →
                          r' = r ∧ out' = out
```

The statement is a long conjunction, and its proof is short, because each conjunct is a theorem of an earlier chapter applied to the hypothesis its check establishes:

```
  obtain (hs, hf) := Prog.check_ok hc hr
  obtain (hty, hv) := Runs.typed (Prog.checkTypes_sound ht) hm hr
  exact (hs, hty, Runs.no_control hctl hr, hv, hf,
    fun r' out' h' => Runs.det h' hr)
```

`Prog.check_ok` packages conjuncts 1 and 5 (scoping and `<decides>` soundness, from `Eval.scoped_ok` and `Runs.no_fail`), `Runs.typed` gives 2 and 4, `Runs.no_control` gives 3, and `Runs.det` gives 6. The heap-effect theorem of Chapter 7 is stated separately as `Prog.accepted_no_writes`, because it is about individual calls rather than whole runs.

8.4 Why the front end need not be trusted

The front end runs `Prog.check`, `Prog.checkTypes` and `Prog.checkControl` on its own output, and refuses to emit or run a program that fails them. So for every program it accepts, the hypotheses of `Prog.accepted` hold *by computation*, whatever bugs the elaborator has. A bug can show up in only two ways:

- **The front end rejects a correct program.** An elaborator that emits ill-typed Core is caught by its own checker. This is an inconvenience, not a broken promise. (openverse's error messages name the function that failed `checkTypes` precisely so that such bugs can be found.)
- **The front end emits a program that means something else.** If the elaborator translated `X + 1` as `X + 2`, the result would still be checked, typed, deterministic and safe: just wrong. `Prog.accepted` does not say that the Core program is *the* meaning of the surface program; it says that whatever the meaning is, it is well behaved. Faithfulness of the elaboration is tested, against the Book of Verse and the test suite, not proved.

This is the general bargain of translation validation, and it is a good one: the checkers are small and proved, the translator is large and free to change.

8.5 The remaining errors, and the seam to C

Overflow and `Err()` remain, by design. `Err()` is a program asking to stop. Overflow is the reading openverse takes for `int`: 64-bit, with overflow an error rather than a wrap-around or a failure. Neither is caught by a failure context. The integer division family shows the fine print:

```
Main():void =
    Low:int = -9223372036854775807 - 1
    if (M := Mod[Low, -1]) { Print("Mod[Low, -1] = {M}") }
    if (Q := Quotient[Low, -1]) { Print("Quotient = {Q}") }
```

Run in the playground

```
Mod[Low, -1] = 0
```

The remainder of -2^{63} by -1 is 0, representable; the quotient is 2^{63}, which is not, so `Quotient` raises an overflow error. The `if` does not catch it, the program stops with exit code 2, and only the first line was printed. Lean states this corner exactly:

```
apply_min_neg_one :
  Prim.floorDiv.apply [Value.int Int64.minV, Value.int (-1)] =
    Res.err ErrKind.overflow ∧
  Prim.ceilDiv.apply [Value.int Int64.minV, Value.int (-1)] =
      Res.err ErrKind.overflow ∧
    Prim.quotient.apply [Value.int Int64.minV, Value.int (-1)] =
        Res.err ErrKind.overflow ∧
      Prim.modulo.apply [Value.int Int64.minV, Value.int (-1)] =
        Res.ok (Value.int 0)
```

This theorem sits on a **seam**. openverse's C runtime implements the integer operations in C, and a separate proof, in Rocq with the QuickProof tool under CompCert's semantics of C, shows that each C function computes a reference formula written with C99's truncating `/` and `%`. `Seams.lean` restates each formula in Lean, symbol for symbol, and proves that it *is* the Lean semantics of the primitive:

```
@apply_floorDiv : ∀ {a b : Int},
  b ≠ 0 →
    Prim.floorDiv.apply [Value.int a, Value.int b] = Res.ofInt (cFloorDiv a b)
```

where `cFloorDiv a b` is "truncate, then subtract one if the remainder is non-zero and its sign differs from the divisor's". The two proofs meet at the formula; the one step no tool checks is that the C text and the Lean text are the same formula, and openverse prints them side by side (`docs/SEAMS.md`) for a human to compare. Here is the semantics they agree on, for negative operands:

```
Main():void =
    if (A := Floor(-7 / 2), B := Ceil(-7 / 2),
        Q := Quotient[-7, 2], M := Mod[-7, 2]):
        Print("Floor {A}, Ceil {B}, Quotient {Q}, Mod {M}")
    if (Q := Quotient[7, -2], M := Mod[7, -2]):
        Print("Quotient {Q}, Mod {M}")
    if (not Mod[1, 0]) { Print("Mod by zero fails") }
```

Run in the playground

```
Floor -4, Ceil -3, Quotient -4, Mod 1
Quotient -3, Mod 1
Mod by zero fails
```

`Quotient` and `Mod` are Euclidean: the remainder is never negative, and $a = b \cdot \text{Quotient}[a,b] + \text{Mod}[a,b]$.

8.6 The whole argument

Put together, the chain from a Verse source file to a guarantee about the bytes the C runtime prints has these links:

Link	Evidence
surface Verse → Verse Core	front end: unverified, tested; output checked
checks ⇒ hypotheses	Lean: `checkEffectSound_ok`, `checkTypes_sound`, …
hypotheses ⇒ guarantees	Lean: Chapters 2–7, combined in `Prog.accepted`
relation = reference interpreter	Lean: `eval_iff` (Chapter 1)
Verse Core → binary (OVBC, `Prog.encode`)	unverified; covered by differential testing
OVBC → C loader	loader proved (QuickProof, Rocq) to accept only `WellScoped` programs
reference interpreter ≈ C interpreter	differential testing and fuzzing, under sanitizers; not proved
C arithmetic = `Prim.apply`	QuickProof law `arith_exact` + `Seams.lean`, meeting at a formula read by a human
C rollback store	QuickProof law `store_rollback_model`, against a naive snapshot model, bounded

What is trusted is Lean's kernel, the statement of the semantics (`Semantics.lean`, `Prims.lean`), QuickProof's own trusted base for the C proofs (Rocq, VST, CompCert's C semantics), the reading of each seam, and the gap between CompCert's C and the compiler that builds the runtime. That is not nothing, but every item is named, and none of them is five thousand lines of elaborator.

8.7 Limits and readings

- **Divergence** is outside every theorem (Chapter 1): `Runs` describes finished runs. openverse reports running out of fuel, recursion depth or memory as a resource limit (exit 3).
- **The C interpreter is not proved.** It is a transcription of the Lean interpreter, tested against it on every test program and on random ones. Its slot-and-output rollback store (`store.c`), its loader and its integer arithmetic are proved, with

bounds stated in their laws; its object store and its value heap are not.

- **Meaning is a definition.** All of this is relative to the semantics as defined, and that the definition is Verse, as the Book of Verse describes it, is a matter of reading, documented in openverse's `docs/LANGUAGE.md`.

Exercises

1. List, for each conjunct of `Prog.accepted`, the chapter and the theorem it comes from, and the check that establishes that theorem's hypothesis.
2. `Runs.no_control` needs `checkControl`, but the front end also rejects `break` in a `for` body with a message. Why is the Lean check still needed? What would a bug in the front end's own check lead to without it?

```
Main():void =
    for (X := 1..3) { break }
```

Run in the playground

```
Rejected: break is only allowed inside loop
```

3. Give a Verse Core program, not produced by the front end, that passes `Prog.check` but raises a type error. Which check rejects it?
4. Prove $a = b \cdot \mathrm{Quotient}[a,b] + \mathrm{Mod}[a,b]$ and $0 \le \mathrm{Mod}[a,b] < |b|$ for $b \neq 0$, from the definitions Quotient $= \lfloor\cdot\rfloor$-or-$\lceil\cdot\rceil$ by the sign of b. Then check `Mod[-7, -2]` with openverse.
5. (*) Translation validation checks outputs, not the translator. Describe a check that would make "faithfulness" of the elaboration of `X + 1` machine-checkable for each program. What would you have to define first?

Chapter A

Notation and the Grammar of Verse Core

This appendix collects the notation of the book. The authoritative definitions are the Lean files named in each section, in the openverse repository under `OpenVerse/Core/`.

A.1 Values, outcomes, states (`Value.lean`, `Semantics.lean`)

Values v, outcomes r, error kinds k and states s (a frame, an output and a heap):

$$
\begin{array}{lcl}
v & ::= & \mathsf{int}\ n \mid \mathsf{logic}\ b \mid \mathsf{char}\ c \mid \mathsf{float}\ x \mid \mathsf{arr}\ [\vec{v}] \mid \mathsf{tup}\ [\vec{v}] \mid \mathsf{none} \mid \mathsf{some}\ v \mid \mathsf{ref}\ a \\
r & ::= & \mathsf{ok}\ v \mid \mathsf{fail} \mid \mathsf{brk} \mid \mathsf{ret}\ v \mid \mathsf{err}\ k \\
k & ::= & \mathit{overflow} \mid \mathit{type} \mid \mathit{slot} \mid \mathit{call} \mid \mathit{control} \mid \mathit{user} \\
s & ::= & \langle \mathit{fr}, \langle o, h \rangle \rangle
\end{array}
$$

Integers satisfy $-2^{63} \le n < 2^{63}$ (an invariant that `Res.ofInt` maintains, not one the Lean type enforces); `void` is tup `[]`; a `string` is an array of `chars`, each a UTF-8 code unit (below 256 in a typed program: `hasTy` requires it); a float holds the bits of an IEEE 64-bit double, with one canonical NaN; `false` as an empty option is none. A heap is an array of objects, each a class tag and a list of field values.

A.2 Expressions (`Syntax.lean`)

$$
\begin{array}{lcl}
e & ::= & \mathsf{lit}\ v \mid \mathsf{local}\ i \mid \mathsf{assign}\ i\ e \mid \mathsf{seq}\ e_1\ e_2 \mid \mathsf{prim}\ p\ \vec{e} \mid \mathsf{call}\ f\ \vec{e} \\
 & \mid & \mathsf{ite}\ c\ t\ e \mid \mathsf{for}\ i\ e_{it}\ e_{filt}\ e_{body} \mid \mathsf{loop}\ e \mid \mathsf{brk} \mid \mathsf{ret}\ e \mid \mathsf{print}\ e \mid \mathsf{heap}\ o\ \vec{e} \\
o & ::= & \mathit{new}\ t \mid \mathit{get}\ k \mid \mathit{set}\ k \mid \mathit{tag} \mid \mathit{isTag}\ \vec{t}
\end{array}
$$

Primitives p: arithmetic (`add`, `sub`, `mul`, `neg`, `abs`), the division family (`floorDiv`, `ceilDiv`, `quotient`, `modulo`), comparisons (`lt`, `le`, `gt`, `ge`, `eq`, `ne`), `query` (X?), construction (`mkArr`, `mkTup`, `mkSome`), arrays (`concat`, `index`, `len`, `arrSet`, `flatten`, `range`), `tupGet`, `toStr`, `fail`, `userErr`. A function has a name, an arity, a number of slots, a `decides` flag, a `writes` flag and a body; a program is a list of functions and the index of `Main`.

Derived forms (`Derived.lean`, Chapter 3): not e, a or$_t$ b, a and b, option$_t\{e\}$, logic$\{e\}$, all defined with ite and seq.

A.3 Judgments

Notation	Lean	Meaning
$P \vdash \langle s, e\rangle \Downarrow \langle r, s'\rangle$	`Eval P s e r s'`	evaluation (Chapter 1)
eval $P\ n\ s\ e$	`eval P n s e`	interpreter with fuel n
Runs $P\ r\ o$	`Runs P r o`	a whole run: outcome, output
$S;\Gamma;\rho \vdash e : \tau$	`Typed S Γ ρ e τ`	typing (Chapter 5)
$\Xi \models v : \tau$	`hasTy Ξ v τ = true`	value typing
$\tau \leq \tau'$	`τ.sub τ' = true`	subtyping
$\theta\tau$	`τ.subst θ`	substitution (Chapter 6)
mayFail$_P(e)$	`e.mayFail P`	may fail (Chapter 4)
writesHeap$_P(e)$	`e.writesHeap P`	may write a field (Chapter 7)
$g \preceq g'$	`g.keeps g'`	keeps every object (Chapter 7)
$\Xi \sqsubseteq \Xi'$	`Ξ <+: Ξ'`	prefix of a heap typing

Rule names in small caps correspond to Lean constructors: Ite-Fail is `Eval.ite_fail`, T-Call is `Typed.call`.

A.4 Index of theorems

Theorem	File	Chapter
`eval_iff`, `eval_fuel_irrelevant`	`Adequacy.lean`	1
`Eval.det`, `Runs.det`	`Adequacy.lean`	2
`ite_rollback`, `notE_ok`, `notE_notE_ok`, `orE_right`, `optionE_none`	`Derived.lean`	3
`Eval.no_fail`, `Runs.no_fail`	`Effects.lean`	4
`Eval.typed`, `Runs.typed`, `hasTy_sub`, `tc_sound`	`TypeSafety.lean`, `TypeCheck.lean`	5

Theorem	File	Chapter
`Typed.subst`	`TypeSafety.lean`	6
`Eval.no_writes`, `Prog.accepted_no_writes`	`Writes.lean`, `Guarantees.lean`	7
`Prog.accepted`, `Eval.scoped_ok`, `Runs.no_control`, `apply_*`	`Guarantees.lean`, `Scoping.lean`, `Control.lean`, `Seams.lean`	8

The full statements, as Lean prints them, with the axioms each proof uses, are in openverse's `docs/CLAIMS.md`, which is generated from the sources and checked by openverse's continuous integration. This book's build compares each statement it quotes against that file.