



Write You a Haskell

Building a modern functional compiler from first principles.

Stephen Diehl

Write You a Haskell

Copyright © 2014-2026 Stephen Diehl.

Prose: Creative Commons Attribution-ShareAlike 4.0 International. Code: MIT License. Fonts: SIL
Open Font License.

Contents

1	Introduction	9
2	Parsing	19
3	Lambda Calculus	29
4	Simply Typed Lambda Calculus	41
5	Evaluation	49
6	Hindley–Milner Inference	55
7	Design of Fun	71
8	Layout and the Extended Parser	81
9	The Renamer	93
10	Typed Core and Core Lint	103
11	Datatypes and Pattern Matching	113
12	Records	123
13	Kinds and Type Constructors	133
14	Type Classes	139
15	Monads, Do Notation and IO	145
16	Pattern-Match Compilation	153
17	The Prelude and Deriving	163
18	A-Normal Form and Join Points	173
19	Closure Conversion	181
20	Data Representation and the Runtime	187
21	LLVM Code Generation	195
22	JIT and the REPL	205
23	Optimization	215
24	A Precise Garbage Collector	223
25	Future Work	233
A	Surface Syntax	239
B	Typing Rules	245
C	Intermediate Representations	251
D	Runtime ABI	259
E	Diagnostic Index	263
	Bibliography	267

Part I: Foundations

1

Introduction

Motivation

I started writing this text in 2014 because the introduction I needed did not exist. The material I found on functional compilers was often dense and hard to approach, or written for an academic audience rather than people building software in industry. I wanted someone to walk me through a working compiler and explain the ideas as we needed them.

I slowly filled in the gaps in my own knowledge, but there was no reason others should have to take the same slow route. In the spirit of “be the change you want to see,” I decided to write the introduction I had been looking for. This book is my way of lowering the ladder for the next person.

We’ll build Fun, a strict functional language with inferred types, algebraic data, nominal records, and type classes. And building Fun is going to be fun! We’ll write it in Rust and take it all the way to native code through LLVM. Along the way we’ll build interpreters that let us run programs before we have a code generator, and check our translations as we go. The Haskell Report gives us a starting point for syntax and typing (S. Marlow 2010).

A compiler becomes easier to understand when each transformation has a small input language and a small output language. The early chapters study parsing, evaluation, and inference separately. From Chapter 7 onward each crate is a complete snapshot of the same compiler. Read a chapter together with its diff. Run an earlier snapshot whenever you want to revisit a simpler implementation.

Tool Setup

Use Apple Silicon macOS, or Linux on x86-64 or AArch64. We need Rust and Cargo to build the chapters, just to run the repository recipes, a C compiler and Boehm GC for the runtime, and LLVM 22 for native compilation and the JIT. The book has a separate toolchain, installed by `just bootstrap`.

macOS

Install Apple’s command-line development tools and follow the Homebrew installation instructions if `brew` is not already available. Then install the system dependencies:

```

xcode-select --install
brew install git python just llvm@22 bdw-gc pkg-config
export LLVM_SYS_221_PREFIX="$(brew --prefix llvm@22)"
export GC_LIB_DIR="$(brew --prefix bdw-gc)/lib"

```

Wait for the command-line tools installer to finish before running the build. Keep the two exports in your shell configuration if you want them in later terminal sessions. `llvm@22` is intentional. An unversioned LLVM package can select a different ABI from the one used by our Rust bindings.

Ubuntu

These commands target Ubuntu 26.04 LTS, the current LTS when this edition was written. They also work with the supported Ubuntu 24.04 CI environment. Install the common tools, then use the official LLVM package installer to select version 22 explicitly:

```

sudo apt update
sudo apt install -y build-essential git curl wget ca-certificates \
  gnupg lsb-release software-properties-common python3 pkg-config \
  libgc-dev libssl-dev just
curl -fsSLo /tmp/wyah-llvm.sh https://apt.llvm.org/llvm.sh
sudo bash /tmp/wyah-llvm.sh 22
sudo apt install -y llvm-22-dev libpolly-22-dev
export LLVM_SYS_221_PREFIX=/usr/lib/llvm-22

```

The versioned LLVM prefix belongs in your shell configuration for later builds. The distribution's default Rust package need not match the book's pinned compiler, so install Rust through `rustup` next.

Rust and the Repository

Follow the canonical Rust installation guide. On either platform, its `rustup` installer is:

```

curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
. "$HOME/.cargo/env"
git clone https://github.com/sdiehl/write-you-a-haskell.git
cd write-you-a-haskell
rustup show
rustc --version
cargo --version

```

Inside this checkout, `rust-toolchain.toml` selects Rust 1.99.0 with Clippy and `rustfmt`. `Rustup` installs the selected toolchain when needed. The `rustup` book explains how directory toolchains work. `just` is a command runner, not another build system. Its installation guide lists alternatives to the system packages above.

Now build a small chapter and the complete workspace:

```
cargo run -p ch02_calc -- repl
just build
cargo fmt --all -- --check
cargo clippy --workspace --all-targets -- -D warnings
```

For HTML and PDF, install the pinned book tools locally:

```
just bootstrap
just book
just pdf
```

Bootstrap installs mdBook, its preprocessors, Pandoc, and Typst under `.tools`. The `just` recipes put `.tools/bin` on `PATH`. Fonts are already vendored in the repository. Open `book/build/html/index.html` or `build/write-you-a-haskell.pdf`. The published copies are the HTML at <https://sdiehl.github.io/write-you-a-haskell/> and the PDF beside it at <https://sdiehl.github.io/write-you-a-haskell/write-you-a-haskell.pdf>, which is also the asset on the repository's latest GitHub release. The full compiler tests run with `cargo test --workspace` and include long recursion and allocation regressions.

Nix Alternative

With Nix installed, the flake supplies the pinned Rust toolchain, LLVM 22, Boehm, a C compiler, and development tools for Apple Silicon macOS and both Linux architectures:

```
nix --extra-experimental-features 'nix-command flakes' develop
just build
just bootstrap
just pdf
```

The lock file fixes the package inputs. The shell sets the LLVM and GC paths, so it replaces the platform package-installation steps above. Bootstrap still installs the book's exact tool versions into `.tools`.

Why Rust

When I first wrote this book in the mid-2010s, Haskell was an appealing way to explain a compiler. Algebraic data types, pattern matching, and a compact syntax made the transformations almost read like their mathematical definitions. But getting from those definitions to a complete native compiler already meant relying on libraries whose support was thin. In my experience, that maintenance problem has grown worse with time.

I still love Haskell’s surface language. I no longer consider it a good practical choice for building the compiler in this book. A compiler needs maintained bindings, diagnostics, testing tools, packaging, and a predictable runtime as well as elegant tree transformations. Implicit laziness also puts extra distance between the program I write and its allocation and evaluation behavior. Those costs matter when a compiler must process large inputs reliably.

Rust is my choice for this edition because of its much broader ecosystem and sustained investment in production tooling. It offers enums and pattern matching for our intermediate languages, explicit ownership for their storage, and direct access to LLVM and the native runtime. Cargo makes the chapter snapshots easy to build and test together. I want readers to spend their time learning compilation, with a toolchain they can continue using after the last chapter.

For this project, Rust is the best balance of expressive implementation code and practical engineering. Perhaps there will be an OCaml edition one day. For now, Haskell remains a language worth studying, while Rust is the language I want to build in. The original Haskell edition is preserved in the repository under the `haskell-edition` tag.

Why not laziness?

Haskell’s strongest lessons survive without making every ordinary computation potentially lazy. Algebraic data, pattern matching, principal types, purity, and type classes are all worth keeping. Fun brings them together with call-by-value. Arguments are computed before a call, constructor fields before construction, and ordinary bindings before their bodies. This gives the reader a direct account of when work happens and where storage is needed.

I do not see implicit laziness as a productive default for a new language. A suspended expression may retain its environment long after its source binding looks finished. Forcing it can move both work and failure far from their cause. Sharing can avoid repeated work, but it introduces its own allocation and update protocol. Strictness analysis can recover efficient execution in many cases, yet that is substantial machinery to reconstruct a cost model we could have made explicit from the start. Chapter 5 studies the tradeoff in executable form.

Delayed computation is still useful. My preferred foundation for making it opt-in is **call-by-push-value** (CBPV). It separates values from computations. A computation can produce a value, and a value can hold a suspended computation. Explicit thunk and force operations mark the boundary. Both call-by-value and call-by-name can be expressed through these smaller operations (Levy 2001). The programmer or compiler can therefore specify where delay belongs instead of inheriting it everywhere.

CBPV does not by itself memoize a suspension. Call-by-need additionally requires sharing and an update that remembers the result. Forcing an ordinary thunk twice can run its computation twice. That distinction is precisely why I prefer this foundation for exploring laziness. Fun itself remains

a strict call-by-value language. It implements neither CBPV nor lazy values. An explicit account of delay is a direction for a later language, beyond this book’s compiler.

Chapter roadmap

The tables below describe the complete book. Linked titles open each chapter. Chapters 2–6 are standalone languages. Chapters 7–24 are successive snapshots of Fun. Each row names the piece you will build, rather than a list of prerequisites.

Part I: Foundations

No.	Topic	What you will build
1	Introduction	The workspace, test runner, chapter tools, and HTML/PDF book pipeline.
2	Parsing	An arithmetic language with logos, LALRPOP, a comparison Pratt parser, diagnostics, and a REPL.
3	Lambda Calculus	Substitution, de Bruijn indices, normal-order reduction, and strict closures.
4	Simply Typed Lambda Calculus	Annotated functions and a type checker that runs before evaluation.
5	Evaluation	Recursive functions, a constant-stack CEK machine, and a call-by-name comparison.
6	Hindley–Milner Inference	Polymorphic let inference, using both Algorithm W and level-based union-find.

Part II: Frontend

No.	Topic	What you will build
7	Design of Fun	The first Fun compiler, with source spans, diagnostics, pass sessions, Core, and a CLI.
8	Layout and the Extended Parser	Offside layout, comment-preserving formatting, signatures, fixities, and sections.
9	The Renamer	Unique names, lexical scopes, dependency components, and mutual recursion.
10	Typed Core and Core Lint	System F elaboration and independent Core lint to catch compiler mistakes.
11	Datatypes and Pattern Matching	Algebraic data, lists and tuples, nested patterns, equations, and guards.
12	Records	Nominal records with construction, updates, patterns, and resolved dot projection.

No.	Topic	What you will build
13	Kinds and Type Constructors	Higher-kinded constructors, inferred kinds, and checked type synonyms.
14	Type Classes	Constraints, instances, superclasses, defaults, dictionary passing, and Double.
15	Monads, Do Notation and IO	Higher-kinded control classes, do-notation, opaque IO, and effectful programs.
16	Pattern-Match Compilation	Decision trees and join points, with exhaustiveness and redundancy warnings.
17	The Prelude and Deriving	The full source Prelude and generated Eq, Ord, and Show instances.

Part III: Backend

No.	Topic	What you will build
18	A-Normal Form and Join Points	Type erasure, ANF, a small simplifier, and an iterative execution oracle.
19	Closure Conversion	Explicit environments, lifted functions, partial application, and a Flat IR oracle.
20	Data Representation and the Runtime	Tagged words, heap layouts, eval/apply, shared primitives, and a Boehm-backed runtime.
21	LLVM Code Generation	Native object emission, linking, tail calls, and tests against the interpreters.
22	JIT and the REPL	LLVM execution and a persistent REPL with type, kind, and compiler-pass inspection.
23	Optimization	Work preservation, reachable globals, and native numeric intermediates.
24	Precise GC (Optional)	Live-root tracking, a shadow stack, and a stress-tested copying collector.
25	Future Work	A guide to laziness, modules, richer types, optimization, and debugging extensions.

Appendices collect the grammar, typing rules, intermediate representations, runtime ABI, and diagnostic index. A bibliography closes the book.

How We Will Build It

We begin with a calculator. It is small enough that we can follow an expression from the text we type to the answer it produces. We then build a few small languages to explore functions, evaluation,

and types separately. Each has a complete Rust implementation you can run and change. You do not need to understand a whole compiler before you can experiment with one part of it.

In Chapter 7, those ideas come together in Fun. From there, each chapter starts with the previous chapter's compiler and adds one piece. Records extend the language, type classes introduce a way to share operations across types, and later chapters teach the compiler to produce native code. Every chapter has its own crate, so you can keep an earlier version open beside the one you are studying.

Read the changes as well as the finished code. Each chapter lists its new and changed source files, with the important ones first. Comparing two snapshots lets you see what a feature actually asks of the compiler. You might need a new syntax node, a typing rule, and a translation into simpler operations. The book includes code from those source files, so the implementation you read is the one you run. Tests carry forward too, helping us check that adding a feature has not broken what already worked.

The theory arrives alongside the problems it helps us solve. We will try a small example, find the question it raises, and introduce enough notation to make the answer precise. You can work through the code and examples first, then return to a rule once you have seen it in action. No prior course in type theory is assumed. Familiarity with Rust enums, pattern matching, borrowing, and Cargo will help you follow the implementations.

Theory

A compiler rewrites a program several times before the computer can run it. The challenge is to change how the program is written while keeping what it does. If your program prints 42, the machine code we produce should also print 42. If it asks for your name before greeting you, compilation must keep those actions in that order.

We will build up those steps one at a time. First we work out how the source text fits together. In $1 + 2 * 3$, multiplication belongs inside the addition. Then we check that the operations make sense, so adding a number to a function is caught before the program runs. Later chapters translate the program into simpler forms that are easier to turn into machine code.

How do we know we got a translation right? One useful trick is to write a small interpreter for each form and run the same examples through them. If they give different answers, we have found a compiler bug. Matching answers do not establish that every possible program works, but they give us a practical way to check our work as we go. Along the way, we will also learn how types and rules about evaluation help us reason about programs beyond the examples we can test (Pierce 2002).

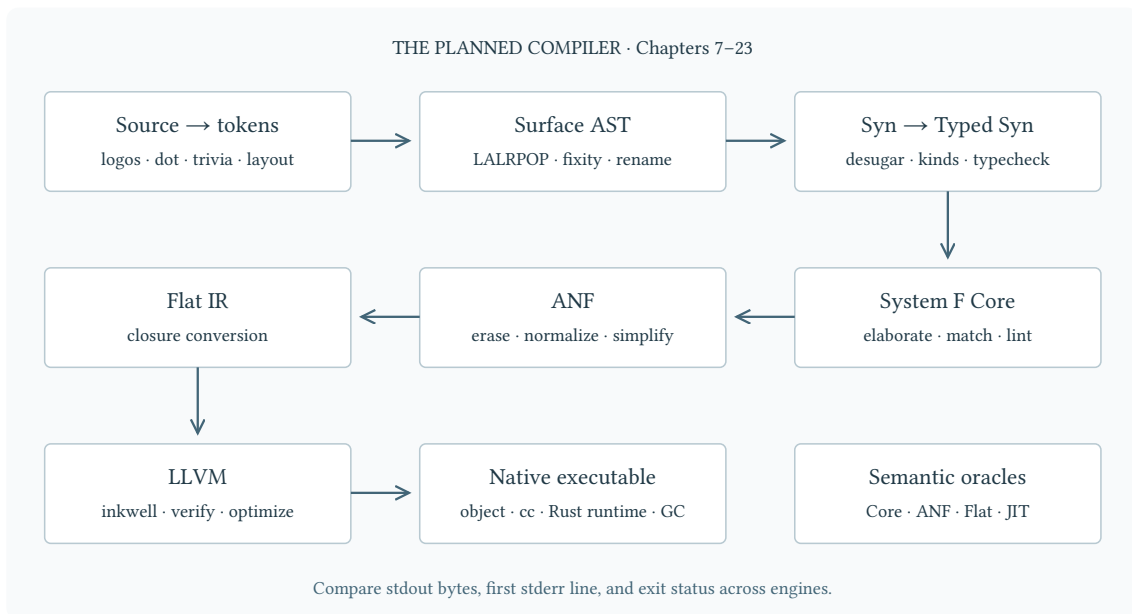


Figure 1: The planned compiler pipeline

Implementation

The repository is part of the book. Each chapter has a crate you can run, take apart, and change. If you're wondering what a transformation really did, ask the compiler to dump its output and compare it with the input. That habit will tell you more than staring at the implementation and hoping for enlightenment.

We also want to keep the examples working as the compiler grows. `testkit` runs small programs and checks their output, errors, and exit status. Later it runs the same program through several execution engines. If two engines disagree, we have something concrete to investigate.

```

/// The chapter supplies its binary and available engines; the runner owns
policy.
pub struct Config<'a> {
    pub binary: &'a Path,
    pub programs: &'a Path,
    pub engines: &'a [Engine],
    pub timeout: Duration,
}

```

`xtask` handles the repetitive work of copying a chapter and checking its inherited examples. The book itself includes Rust directly from the chapter sources. You can follow an included function into

the crate and experiment with the code you're reading. The HTML and PDF are built from the same chapters.

Running It

Install the pinned book tools with `just bootstrap`, then run:

```
$ cargo test --workspace
$ just book
$ just pdf
```

The HTML is in `book/build/html/`. The PDF is `build/write-you-a-haskell.pdf`. Compiler `--dump` commands start in Chapter 2. There is no compiler pass to dump in Chapter 1.

Further Reading

The Haskell Report is the language reference (S. Marlow 2010). Pierce introduces the operational and typing judgments we will use (Pierce 2002). Kiselyov connects type generalization with dependency tracking, preparing for Chapter 6 (Kiselyov 2013). When you want to compare our choices with the real thing, Takenobu Tani's GHC Reading Guide ([pdf](#) / [github](#)) is a map of the GHC source tree and its pipeline.

2

Parsing

New files

- `src/lexer.rs`
- `src/pratt.rs`
- `src/ast.rs`
- `src/eval.rs`
- `build.rs`
- `src/diagnostic.rs`
- `src/lib.rs`
- `src/main.rs`

Changed files

None.

Motivation

We read `1 + 2 * 3` and know that the multiplication comes first. The computer gets a string of characters. Before it can calculate anything, we have to teach it how those characters fit together. That's the parser's job.

We'll start with a calculator, small enough to follow all the way from source text to an answer. We'll build its parser twice. LALRPOP lets us write down a grammar and generates the parsing machinery for us. A Pratt parser puts the operator-handling logic in a short Rust function. Seeing both is a useful way to discover what a parser generator is doing on our behalf.

The Language, Now

A program is one expression. It has decimal integer literals, names, `+`, `-`, `*`, `/`, unary minus, parentheses, and a non-recursive `let`.

```
let x = 7 in (let y = x + 1 in y * y) - x
```

The binding is visible in its body, not its right-hand side. A nested binding shadows an outer name. Multiplication and division bind above addition and subtraction. All four operators associate left. Unary minus binds above them. The / operator truncates integer division toward zero in this mini-language. Full Fun will reserve / for fractional arithmetic and spell this operation quot.

Theory

There are two plausible trees for $1 + 2 * 3$. One adds first and returns 9. The other multiplies first and returns 7. We want the second, so our grammar needs to express that choice. We'll separate sums, products, and atoms into different rules. A sum can contain a whole product as an operand. Parentheses let us start again from the full expression grammar when we want another grouping.

Choosing a parsing strategy

There isn't one parsing technique that wins at everything. Some approaches make the whole grammar easy to inspect. Others make it easy to write a little parser directly in the host language. Before choosing, it helps to separate three things we often bundle together. The grammar says what syntax we accept, the algorithm recognizes it, and the API determines how we write the parser.

Approach	Main idea	Tradeoff for this book
LR(1)/ LALR(1)	A generated state machine shifts tokens and reduces grammar productions.	A declarative whole-language grammar, with conflicts reported during generation.
Pratt	An expression parser uses binding powers to decide which operator owns the next operand.	Small, readable operator parsing. Declarations and blocks still need their own grammar.
Pack-rat	Memoize parsing results by rule and input position.	Predictable recognition time for a fixed suitable grammar, at the cost of a memo table.
Parser com- bina- tors	Compose small parsers using ordinary functions.	Convenient abstraction. Backtracking, memoization, and error behavior depend on the library.

LR(1) scans left to right and constructs a rightmost derivation in reverse. Its state summarizes the prefix already read. One lookahead token helps decide whether to shift another token or reduce a completed production. The “1” does not limit expressions to one token. The parser’s stack retains arbitrarily deep syntactic context. Left recursion, such as a sum containing a smaller sum, fits this bottom-up process naturally.

LALR(1) merges LR(1) states with the same underlying LR(0) items, combining their lookahead sets. This reduces table size but can introduce conflicts absent from the unmerged grammar. Despite its name, our generator uses LR(1) by default. LALR(1) is an option. See the LALRPOP documentation. A conflict is useful feedback. The grammar needs attention before we can trust its generated parser.

Pratt parsing makes the operator decision local. After reading $1 + 2$, the parser compares the following $*$ with the binding power of $+$. Multiplication belongs inside the right operand, so $1 + 2 * 3$ becomes $1 + (2 * 3)$. For $8 - 3 - 1$, the right-operand threshold excludes an equal-strength subtraction, giving $(8 - 3) - 1$. Prefix and infix uses of $-$ can have different handlers. This is an excellent fit for expressions, but does not by itself specify the structure of a class declaration or a layout block.

Packrat parsing avoids repeated work during backtracking by remembering a rule's success or failure at each position. It is commonly associated with parsing expression grammars (PEGs), whose alternatives use ordered choice. The first successful alternative wins. Reordering alternatives can therefore change the language. For a fixed grammar with bounded work per memoized entry, this gives linear recognition time and generally linear memo storage. Ordinary left-recursive rules need rewriting or a specialized extension. Ford's packrat and PEG overview develops these choices.

Parser combinators are a way to write parsers, not one recognition algorithm. A sequencing combinator runs one parser after another. A choice combinator tries alternatives according to its library's rules. A combinator library may commit, backtrack, or memoize. Its pleasant surface does not automatically give linear time, good recovery, or support for left recursion. Those remain design choices.

Why LALRPOP and a comparison Pratt parser?

LALRPOP gives us one inspectable grammar for the language as a whole, typed Rust actions for constructing the AST, and explicit recovery productions. Pratt gives us a compact second account of the calculator's precedence rules. Implementing both here makes the grouping decisions concrete and gives us an independent comparison. The later compiler keeps LALRPOP. It does not run both parsers.

Haskell-style syntax also needs work around the grammar. In Chapter 8, *logos* recognizes tokens, *marginalia* retains comments and whitespace for formatting, and *offides* translates indentation into block delimiters. The latter two libraries supply the trivia and Haskell-style layout support this frontend needs. LALRPOP then consumes that prepared token stream. Layout remains visible as a separate, testable transformation instead of being scattered through parser actions. This calculator has neither layout nor comment preservation yet.

User-declared fixities introduce a separate issue. A generated parser cannot have a different precedence table for each source file. Chapter 8 parses flat operator chains and resolves their grouping

afterward. Neither a Pratt loop nor a parser generator removes the need to decide when those declarations become known.

Syntax and typing

A typing judgment already describes this small language. If two operands have type `Int`, their sum has type `Int`:

$$\frac{\Gamma \vdash e_1 : \text{Int} \quad \Gamma \vdash e_2 : \text{Int}}{\Gamma \vdash e_1 + e_2 : \text{Int}} \text{ (Add)}$$

We don't need an inferencer yet, since integers are our only values. It does check lexical scope before evaluating. Pierce develops the distinction between syntax, typing, and evaluation (Pierce 2002). The Haskell Report is useful for comparison, although Fun's later fixity rules are deliberately absent here (S. Marlow 2010).

Implementation

`lexer.rs` uses logos to recognize tokens and retain byte spans. Keywords win over identifiers of the same length. An invalid character or an out-of-range literal produces `E0001`. `ast.rs` stores a span on each expression, making both parsers comparable without losing source information.

```

/// Byte offsets into the single source file, including parentheses when
/// present.
#[derive(Clone, Debug, PartialEq, Eq)]
pub struct Expr {
    pub kind: Kind,
    pub span: Range<usize>,
}

/// An error node exists only during recovery; the driver never evaluates it.
#[derive(Clone, Debug, PartialEq, Eq)]
pub enum Kind {
    Int(i64),
    Var(String),
    Neg(Box<Expr>),
    Binary(Op, Box<Expr>, Box<Expr>),
    Let(String, Box<Expr>, Box<Expr>),
    Error,
}

#[derive(Clone, Copy, Debug, PartialEq, Eq)]
pub enum Op {
    Add,
    Sub,
    Mul,
    Div,
}

```

The generated parser implements precedence with nonterminals. The complete grammar also handles bindings, parentheses, unary minus, and a recovery production.

```

Sum: Expr = {
    <a:Sum> "+" <b:Product> ⇒ Expr::binary(Op::Add, a, b),
    <a:Sum> "-" <b:Product> ⇒ Expr::binary(Op::Sub, a, b),
    Product,
};
Product: Expr = {
    <a:Product> "*" <b:Unary> ⇒ Expr::binary(Op::Mul, a, b),
    <a:Product> "/" <b:Unary> ⇒ Expr::binary(Op::Div, a, b),
    Unary,
};

```

An ! production inserts an error node and stores the recovery diagnostic. The driver returns errors even when recovery produces a tree. The evaluator never receives recovery nodes. `diagnostic.rs` prints a stable error-code line and location, then lets `ariadne` render the source excerpt with byte indexing.

```
/// Parse and recover enough to report several syntax errors, then stop the
pass.
pub fn parse(source: &str) → Result<Expr, Vec<Diagnostic>> {
    let tokens = lexer::lex(source)?;
    let mut recovered = Vec::new();
    let parsed = parser::ExprParser::new()
        .parse(&mut recovered, tokens.into_iter().map(Ok));
    let mut errors: Vec<_> = recovered
        .into_iter()
        .map(|r| parse_error(r.error))
        .collect();
    match parsed {
        Ok(expr) if errors.is_empty() ⇒ Ok(expr),
        Ok(_) ⇒ Err(errors),
        Err(error) ⇒ {
            errors.push(parse_error(error));
            Err(errors)
        }
    }
}
```

The Pratt parser's loop consumes operators whose binding power is high enough. Parsing the right operand at `power + 1` makes the operator left-associative. Prefix parsing handles names, literals, parentheses, and negation. A separate entry handles `let`.

```

fn infix(&mut self, min_power: u8) → Result<Expr, Diagnostic> {
    let mut left = self.prefix()?;
    loop {
        let (op, power) = match self.peek() {
            Some(Token::Plus) ⇒ (Op::Add, 1),
            Some(Token::Minus) ⇒ (Op::Sub, 1),
            Some(Token::Star) ⇒ (Op::Mul, 3),
            Some(Token::Slash) ⇒ (Op::Div, 3),
            _ ⇒ break,
        };
        if power < min_power {
            break;
        }
        self.at += 1;
        let right = self.infix(power + 1)?;
        left = Expr::binary(op, left, right);
    }
    Ok(left)
}

```

`eval.rs` uses a stack of bindings, searched from the end, to model lexical scope. It evaluates a binary operator's left operand before its right operand and restores the environment even if a binding body fails. Arithmetic uses wrapping 63-bit values from the start, avoiding dependence on Rust's debug overflow checks. The normalization function is included as a Rust item:

```

pub fn normalize(n: i64) → i64 {
    (n << 1) >> 1
}

```

```

/// The environment is a stack: the most recent binding shadows earlier names.
pub fn evaluate(
    expr: &Expr,
    env: &mut Vec<(String, i64)>,
) → Result<i64, String> {
    match &expr.kind {
        Kind::Int(n) ⇒ Ok(normalize(*n)),
        Kind::Var(name) ⇒ env
            .iter()
            .rev()
            .find(|(key, _)| key == name)
            .map(|(_, value)| *value)
            .ok_or_else(|| format!("unbound variable `{name}`")),
        Kind::Neg(value) ⇒ {
            Ok(normalize(evaluate(value, env)?.wrapping_neg()))
        }
        Kind::Binary(op, left, right) ⇒ {
            let a = evaluate(left, env)?;
            let b = evaluate(right, env)?;
            Ok(normalize(match op {
                Op::Add ⇒ a.wrapping_add(b),
                Op::Sub ⇒ a.wrapping_sub(b),
                Op::Mul ⇒ a.wrapping_mul(b),
                Op::Div if b == 0 ⇒ return Err("divide by zero".into()),
                Op::Div ⇒ a / b,
            })))
        }
        Kind::Let(name, value, body) ⇒ {
            let value = evaluate(value, env)?;
            env.push((name.clone(), value));
            let result = evaluate(body, env);
            env.pop();
            result
        }
        Kind::Error ⇒ {
            Err("internal error: attempted to evaluate a recovery node"
                .into())
        }
    }
}

```

main.rs connects these pieces to commands and a REPL. Try both parsers on the same expression and inspect their trees. The tests do this too, including generated expressions with different group-

ings. Comparing the trees catches mistakes that a coincidentally correct numerical answer could hide.

Running It

```
$ cd crates/ch02_calc
$ cargo run -q -- run tests/programs/precedence.fun
7
$ cargo run -q -- dump --pass=parse tests/programs/precedence.fun
(+ 1 (* 2 3))
$ cargo run -q -- run tests/programs/precedence.fun --parser=pratt
7
$ cargo run -q -- repl
Calc: :quit to exit, :parse EXPR to inspect the AST.
calc> let x = 6 in x * 7
42
calc> :quit
```

check verifies syntax and names without executing arithmetic. Thus `1 / 0` passes check but run reports `fun: divide by zero` and exits with status 1. Use `dump --pass=tokens` to inspect lexer spans.

Exercises

1. **Easy.** Predict both the value and the parse dump of `20 - 3 * 4 - 1`, then run it.
2. **Easy.** Compare the spans of `1` and `(1)`. Explain why their values agree.
3. **Medium.** Change the Pratt parser's right binding power to `power`. Find the smallest subtraction expression that exposes the associativity change.
4. **Medium.** Write a malformed expression that reports two independent syntax errors. Explain where recovery can resume.
5. **Hard.** Extend the generator to insert redundant parentheses and verify that evaluation remains unchanged.

Further Reading

Pierce separates syntax and semantics carefully (Pierce 2002). The Haskell Report specifies a much larger concrete syntax and fixity-resolution procedure (S. Marlow 2010).

3

Lambda Calculus

New files

- `src/names.rs`
- `src/normal.rs`

Changed files

- `src/eval.rs`
- `src/ast.rs`
- `src/lexer.rs`
- `src/lib.rs`
- `src/main.rs`

Motivation

Our calculator can name a number. What if we could name a computation and use it with different inputs? That's a function, and adding functions takes us a surprisingly long way. With just lambdas and application we can express the pure untyped lambda calculus. Even data can be represented by functions, as we'll see with Church encodings (Pierce 2002). That's a remarkable amount of expressive power for two pieces of syntax!

There are a couple of details we need to get right. If two variables are both called `x`, how do we know which one a function means? And when we call a function, should we evaluate its argument first? We'll build substitution and closures ourselves so we can see the answers, rather than quietly borrowing Rust's answers.

The Language, Now

Programs remain single expressions. Lambdas have one parameter, application associates left, and booleans control an if expression. Comparisons are integer operations. There are no type annotations yet.

```
let twice = \f → \x → f (f x) in twice (\x → x + 1) 0
```

A closure remembers the bindings present where its lambda was evaluated. In `let x = 1 in let f = \y → x + y in let x = 100 in f 2`, the result is 3. Using the caller’s binding of `x` instead would give dynamic scope.

Theory

The Untyped Grammar

The pure untyped lambda calculus has just three forms. An expression is a variable, an application, or an abstraction. Read each vertical bar as “or”.

$e = x$	variable
$ e_1 e_2$	application
$ \lambda x. e$	abstraction

That’s the whole calculus! Our little language adds a few conveniences so we can experiment without encoding every number and boolean as a function.

$e = \dots$	the three forms above
$ n$	integer literal
$ \text{True} \text{False}$	boolean literal
$ \text{let } x = e_1 \text{ in } e_2$	local binding
$ \text{if } e_1 \text{ then } e_2 \text{ else } e_3$	conditional
$ -e$	integer negation
$ e_1 \text{ op } e_2$	binary operation

Here n is an integer and op is one of $+$, $-$, $*$, $/$, $=$, $<$, or $\leq$. Parentheses group expressions without adding a tree node. The parser also has an Error placeholder for recovery, which isn’t a form of an accepted program. “Untyped” means we have no static type checker yet, not that every combination of these forms makes sense at runtime.

Reduction and Evaluation

Calling a lambda means replacing its parameter with the supplied argument. This step is called *beta reduction*. For `\x → x + 1` applied to 41, we replace `x` and get `41 + 1`. So far, so good. The

subtlety begins when names repeat. We replace only *free* occurrences of the parameter in the body, and rename an inner binder if it would capture a free name from the argument. We'll work through exactly that mistake below (Pierce 2002).

Evaluation order decides which reduction we do next. Normal order takes the leftmost outermost reducible expression, and our full reducer also continues under lambdas. Call-by-value first evaluates the function and its argument, then enters the body. A lambda itself is already a value.

The difference becomes visible when a function ignores an argument that would run forever. Normal order can skip that argument. Call-by-value gets stuck trying to compute it. The Y combinator gives us a particularly striking example. It computes factorial under normal order but exhausts our strict evaluator's fuel. Running out of fuel means we've reached our experiment's limit, not that we've proved the computation will never finish.

Why De Bruijn Indices?

Names help us read programs, but their spelling is incidental to binding. $\lambda x \rightarrow x$ and $\lambda \text{answer} \rightarrow \text{answer}$ describe the same function. We call them *alpha-equivalent*. Renaming a bound variable consistently changes nothing. Comparing their syntax trees directly, however, finds different strings. A compiler that rewrites named terms must keep distinguishing harmless renaming from a change in which binder an occurrence refers to.

Consider this reduction of an open term, where y is free:

$$(\lambda x \rightarrow \lambda y \rightarrow x) y$$

Replacing x with y mechanically gives $\lambda y \rightarrow y$. That is wrong. The function should return the free y supplied as its argument, regardless of its own parameter. Rename the inner parameter first:

$$\begin{aligned} &(\lambda x \rightarrow \lambda z \rightarrow x) y \\ &\rightarrow \lambda z \rightarrow y \end{aligned}$$

This is the capture problem our substitution function must solve. De Bruijn indices give binding a representation that does not depend on fresh spellings. Replace each bound occurrence with its distance to its binder, counting from zero. Count enclosing binders, not application nodes or source lines.

We write $\lambda .$ body for an unnamed lambda in the examples below. Its numbers are variable indices, not integer literals. This is explanatory notation, not input syntax for the chapter's parser.

Named term	Indexed term	Reading
$\backslash x \rightarrow x$	$\lambda. 0$	Return the nearest parameter.
$\backslash x \rightarrow \backslash y \rightarrow x$	$\lambda. \lambda. 1$	Skip y to reach x .
$\backslash x \rightarrow \backslash y \rightarrow y$	$\lambda. \lambda. 0$	Return y .
$\backslash f \rightarrow \backslash x \rightarrow f\ x$	$\lambda. \lambda. 1\ 0$	Apply the outer parameter to the inner one.
$\backslash x \rightarrow \backslash x \rightarrow x$	$\lambda. \lambda. 0$	The inner x shadows the outer one.

For a longer example, walk into $\backslash f \rightarrow \backslash x \rightarrow f\ (f\ x)$. Entering the first lambda pushes f onto a scope stack. Entering the second pushes x :

```
scope, nearest first: [x, f]
index:                0  1

f (f x)                1 (1 0)
\ f → \ x → f (f x)    λ. λ. 1 (1 0)
```

Both occurrences of f have index 1. Application adds no binder, so entering $(f\ x)$ does not change either index. Renaming the parameters to g and $seed$ produces exactly the same indexed tree. Alpha-equivalence has become ordinary structural equality (Pierce 2002). This is useful in our printer tests. A parse–print–parse round trip should preserve binding even if names change.

An open term still needs a way to represent free variables. Our `Term` type keeps them as `Free(name)`, separate from `Bound(index)`. Thus $\backslash x \rightarrow x\ y$ becomes `Lam(App(Bound(0), Free("y")))`. A non-recursive `let` introduces a binder only in its body. In `let x = x in x`, the right-hand x is free and the final x is bound:

```
Let(Free("x"), Bound(0))
```

Indices make renaming simple, but moving terms across binders still requires care. Inside $\backslash a \rightarrow (\backslash x \rightarrow \backslash b \rightarrow x)\ a$, the argument a has index 0 at its original location. After reduction the result is $\backslash a \rightarrow \backslash b \rightarrow a$, where the same occurrence needs index 1. Copying the old zero would accidentally select b . An evaluator that substitutes directly into indexed terms needs *shifting*: adjust references to surrounding binders when a term moves into or out of a scope, while leaving its internally bound references alone.

Here we use indices for conversion and comparison. The normal-order reducer still substitutes in the named tree, and the strict evaluator uses closures. That keeps each example focused. Later

Fun passes use unique names so their dumps remain readable. Nameless terms are a useful tool for understanding binding, not a requirement for every compiler IR.

S, K, and I: Functions Without Binders

A *combinator* is a lambda term with no free variables. It needs no bindings from its surroundings. Three small combinators are enough to express the functions of the pure untyped lambda calculus:

```
I = \x → x
K = \x → \y → x
S = \f → \g → \x → f x (g x)
```

Reduction rule	Role
$I\ a \rightarrow a$	Pass the argument through.
$K\ a\ b \rightarrow a$	Keep the first argument.
$S\ f\ g\ a \rightarrow f\ a\ (g\ a)$	Supply an argument to both sides of an application.

These capital letters are mathematical names. The executable tests use lowercase *s*, *k*, and *i* as ordinary local bindings. Applications associate left, so $S\ f\ g\ a$ means $((S\ f)\ g)\ a$. Each reduction rule needs all its displayed arguments before it can fire.

I and *K* are easy to picture. *S* explains how an application can depend on a shared input. Given *f*, *g*, and *a*, compute $f\ a$ and $g\ a$, then apply the first result to the second. The parentheses in $f\ a\ (g\ a)$ matter.

Try choosing *K* for both functions:

```
S K K a
→ K a (K a)    apply the S rule
→ a            apply the K rule
```

So $S\ K\ K$ behaves as *I*. That's a beautiful little result. We can build identity out of two operations that, at first glance, don't look much like identity at all. *S* and *K* suffice, though keeping *I* around makes expressions easier to read. Try the chapter's `s k k (i 42)` example and follow the reductions.

Why should these particular operations eliminate lambdas? For a body *t*, write $[x]t$ for a combinator expression that accepts *x* and produces *t*. Once inner lambdas have been translated, there are three cases:

$[x]x$	$= I$	
$[x]t$	$= K\ t$	when x is not free in t
$[x](p\ q)$	$= S\ ([x]p)\ ([x]q)$	otherwise

If the body is the parameter itself, return it. If the body ignores the parameter, use K to discard it. Otherwise the body is an application, and S passes the parameter to both translated children. This procedure is called *bracket abstraction*. Work from the innermost lambda outward. Translating a closed pure lambda term eventually leaves only combinators and applications.

For example, remove the binder from $\lambda x \rightarrow f\ x$, treating f as a free name:

```
[x](f x)
= S ([x]f) ([x]x)
= S (K f) I
```

Check the result by giving it an argument:

```
S (K f) I a
→ (K f a) (I a)
→ f (I a)
→ f a
```

The occurrence of x has disappeared. The free f remains, as it should. De Bruijn indices retain binders and encode their references as distances. Bracket abstraction removes the binders and expresses their work as application. This gives us a small example of compilation between two representations of the same computation.

There is a catch, and it is the same evaluation-order question we started with. Normal order reduces $K\ \omega$ to ω even if ω runs forever. Our strict evaluator tries to compute ω before entering K 's body, so it never gets that far. These combinator equations describe rewriting in the pure calculus. They don't let a strict compiler discard arbitrary computations. We'll use SKI to understand binding and reduction, while Fun's back end will use closures. Closures also spare us the large application trees that a simple combinator translation can produce.

Implementation

`ast.rs` adds lambdas, application, booleans, and conditionals. Its printer uses precedence contexts and adds parentheses only where required to retain the tree. The parser gives application higher precedence than arithmetic:

```

Application: Expr = {
  <a:Application> <b:Atom> => {
    let (start, end) = (a.span.start, b.span.end);
    Expr::new(Kind::App(Box::new(a), Box::new(b)), start, end)
  },
  Atom,
};

```

names.rs computes free variables and performs capture-avoiding substitution. The helper below handles a binder. The recursive substitution handles each term form. Fresh names avoid both free and bound names in the original body and the replacement, preventing a new name from colliding with an inner lambda.

```

fn binder(
  name: &str,
  body: &Expr,
  target: &str,
  value: &Expr,
) → (String, Expr) {
  if name == target {
    return (name.to_owned(), body.clone());
  }
  let mut name = name.to_owned();
  let mut body = body.clone();
  if free(value).contains(&name) && free(&body).contains(target) {
    let mut used = BTreeSet::from([target.to_owned()]);
    all_names(&body, &mut used);
    all_names(value, &mut used);
    let mut fresh = format!("{name}");
    while used.contains(&fresh) {
      fresh.push('\''');
    }
    let variable = Expr::new(Kind::Var(fresh.clone()), 0, 0);
    body = substitute(&body, &name, &variable);
    name = fresh;
  }
  (name, substitute(&body, target, value))
}

```

The same module converts a named term to indices. A stack holds names from the outermost to the innermost scope. A let right-hand side is converted before its binder is pushed. That binder belongs only to the body.

```

/// Bound names are distances to binders; free names remain explicit.
#[derive(Clone, Debug, PartialEq, Eq)]
pub enum Term {
    Bound(usize),
    Free(String),
    Int(i64),
    Bool(bool),
    Lam(Box<Term>),
    App(Box<Term>, Box<Term>),
    Let(Box<Term>, Box<Term>),
    If(Box<Term>, Box<Term>, Box<Term>),
    Neg(Box<Term>),
    Binary(Op, Box<Term>, Box<Term>),
    Error,
}

/// Convert without substitution; alpha-equivalent expressions have equal
terms.
pub fn debruijn(expr: &Expr, bound: &mut Vec<String>) → Term {
    let mut go = |e: &Expr| Box::new(debruijn(e, bound));
    match &expr.kind {
        Kind::Var(n) ⇒ bound
            .iter()
            .rev()
            .position(|x| x == n)
            .map_or_else(|| Term::Free(n.clone()), Term::Bound),
        Kind::Lam(n, a) ⇒ {
            bound.push(n.clone());
            let body = debruijn(a, bound);
            bound.pop();
            Term::Lam(Box::new(body))
        }
        Kind::Let(n, a, b) ⇒ {
            let value = go(a);
            bound.push(n.clone());
            let body = debruijn(b, bound);
            bound.pop();
            Term::Let(value, Box::new(body))
        }
        Kind::App(a, b) ⇒ Term::App(go(a), go(b)),
        Kind::Binary(op, a, b) ⇒ Term::Binary(*op, go(a), go(b)),
        Kind::If(c, a, b) ⇒ Term::If(go(c), go(a), go(b)),
        Kind::Neg(a) ⇒ Term::Neg(go(a)),
        Kind::Int(n) ⇒ Term::Int(*n),
        Kind::Bool(b) ⇒ Term::Bool(*b),
        Kind::Error ⇒ Term::Error,
    }
}

```

`normal.rs` implements one small step. In the application case it substitutes without first evaluating the argument. If the function is not a lambda, it steps that function first, then the argument when no earlier redex exists.

```
Kind::App(function, argument) => {
    if let Kind::Lam(name, body) = &function.kind {
        return Ok(Some(substitute(body, name, argument)));
    }
    if let Some(next) = step(function)? {
        Kind::App(Box::new(next), argument.clone())
    } else if let Some(next) = step(argument)? {
        Kind::App(function.clone(), Box::new(next))
    } else {
        return Ok(None);
    }
}
```

`reduce` repeatedly calls `step` with a limit and a callback that displays each term. This reducer can inspect open terms, including irreducible applications. It is a term-rewriting demonstration rather than Fun's eventual execution engine.

`eval.rs` instead stores a lambda's environment in a closure:

```
/// Lexical bindings retained by a closure.
pub type Env = Vec<(String, Value)>;

#[derive(Clone, Debug)]
/// Runtime integers, booleans, and code paired with its lexical environment.
pub enum Value {
    Int(i64),
    Bool(bool),
    Closure(String, Rc<Expr>, Rc<Env>),
}
```

The call-by-value application case makes its order explicit:

```
Kind::App(function, argument) => {
  let function = evaluate(function, env, fuel)?;
  let argument = evaluate(argument, env, fuel)?;
  if let Value::Closure(name, body, captured) = function {
    let mut local = (*captured).clone();
    local.push((name, argument));
    evaluate(&body, &local, fuel)
  } else {
    Err("expected a function".into())
  }
}
```

The body runs in the captured environment extended with the argument. Each recursive visit consumes fuel. The CLI uses a larger host stack for these tree walkers. Chapter 5 replaces host recursion with an explicit machine. Neither fuel nor a larger stack is a substitute for that transformation.

Running It

```
$ cd crates/ch03_lambda
$ cargo run -q -- run tests/programs/church.fun
2
$ cargo run -q -- dump --pass=debruijn tests/programs/identity.fun
App(Lam(Bound(0)), Int(42))
$ cargo run -q -- repl
Lambda: :quit, :parse EXPR, :trace EXPR.
lambda> :trace (\x → x) 42
(\x → x) 42
42
lambda> :quit
```

Select normal order with `run --engine=normal`. Use `--fuel=N` when experimenting with terms that might not finish. The two engines count different operations, so the same fuel limit needn't stop them at the same point. Try a function that ignores a failing argument to see why their answers can differ.

Exercises

1. **Easy.** Convert $\lambda x \rightarrow \lambda y \rightarrow y\ x$ to de Bruijn indices by hand.
2. **Medium.** Substitute the free name y for x in $\lambda y \rightarrow \lambda y' \rightarrow x\ y$. Check the result's free variables.
3. **Medium.** Translate $\lambda x \rightarrow x\ x$ by bracket abstraction, then apply the result to I . Compare its reduction with the lambda term.
4. **Medium.** Trace $K\ I\ a\ b$ and explain which argument is discarded.

5. **Hard.** Construct a fixed-point combinator suitable for call-by-value and compare its trace with Y .

Further Reading

Pierce treats substitution, reduction, and nameless terms (Pierce 2002). Felleisen and Friedman introduce the abstract-machine setting that we return to in Chapter 5 (Felleisen and Friedman 1986).

4

Simply Typed Lambda Calculus

New files

- `src/types.rs`

Changed files

- `src/ast.rs`
- `src/eval.rs`
- `src/lexer.rs`
- `src/lib.rs`
- `src/main.rs`
- `src/names.rs`
- `src/normal.rs`

Motivation

Our evaluator will happily try to run `1 True`. It eventually discovers that an integer isn't a function, but we didn't need to run the program to spot that mistake. Could the compiler catch it for us?

A type checker does exactly this sort of reasoning. A function type tells us what a function accepts and what it returns. We check a call by comparing those types with the argument we're about to pass. The rules are small enough to write down, and the Rust implementation will follow them closely.

For now we'll ask the programmer to annotate lambda parameters. This gives us a clear starting point. In Chapter 6 we'll teach the compiler to work out types for itself, including types that work for many different kinds of values.

The Language, Now

Types are `Int`, `Bool`, and arrows. Arrows associate right, so `Int → Bool → Int` means `Int → (Bool → Int)`. An arrow on the left needs parentheses.

```
let choose = \(b : Bool) → \(x : Int) → \(y : Int) →
  if b then x else y
in choose True 10 20
```

We annotate every lambda parameter, but calls, arithmetic, and scope work as before. Try calling an integer as a function again. This time `E0201` comes from the checker, before the evaluator gets involved. We’ve moved the discovery of the mistake earlier in the process.

Theory

The Typed Grammar

The new ingredient is a type on each lambda parameter. First, here are the types we can write. A function’s input and output can themselves be functions.

$\tau = \text{Int}$	integers
Bool	booleans
$\tau_1 \rightarrow \tau_2$	function type

The expressions are familiar from the previous chapter. The annotation in $\lambda x : \tau. e$ tells the checker what it may assume about x .

$e = x$	variable
$e_1 e_2$	application
$\lambda x : \tau. e$	typed abstraction
n	integer literal
<code>True</code> <code>False</code>	boolean literal
<code>let</code> $x = e_1$ <code>in</code> e_2	local binding
<code>if</code> e_1 <code>then</code> e_2 <code>else</code> e_3	conditional
$-e$	integer negation
$e_1 \text{ op } e_2$	binary operation

As before, n is an integer and op is one of `+`, `-`, `*`, `/`, `=`, `<`, or `≤`. We omit source spans and the parser’s recovery placeholder. This grammar describes what we can parse. The typing rules below decide which of those expressions we can safely run. For example, `1 True` still has the shape of an application, but it won’t have a typing derivation.

Typing and Safety

A typing environment maps names to types. Reading a variable looks up its nearest binding. A `lambda` extends that environment while checking its body. Application checks its argument against the function’s domain (Pierce 2002).

$$\frac{\Gamma, x : \tau \vdash e : \sigma}{\Gamma \vdash (\text{lambda } x : \tau. e) : \tau \rightarrow \sigma} \text{ (Abs)}$$

$$\frac{\Gamma \vdash f : \tau \rightarrow \sigma \quad \Gamma \vdash a : \tau}{\Gamma \vdash f a : \sigma} \text{ (App)}$$

The condition of an `if` has type `Bool`. Both branches have the same type. Arithmetic takes integers, while comparisons return booleans. A `let` checks its right-hand side before extending the environment, so it remains non-recursive. There is no implicit quantification. An identity annotated `Int → Int` cannot also accept `Bool`.

Two useful ideas explain what we’ve gained. *Preservation* says that taking an evaluation step doesn’t change an expression’s type. If we start with an `Int` expression, reducing it shouldn’t suddenly produce a function. The main proof step for a `lambda` call shows that substituting a correctly typed argument keeps the body’s type intact. This is the substitution lemma.

Progress says that a closed, well-typed expression is either a value or can take a step. We won’t reach a nonsensical instruction such as calling 42. Our division primitive needs a small qualification, since division by zero can still report a runtime error. Types catch this language’s shape mistakes. They don’t promise that every computation succeeds. Pierce develops the proofs behind these claims (Pierce 2002).

Implementation

`types.rs` represents arrows recursively and prints them with the required parentheses. There are no metavariables or substitution maps yet.

```

/// Simple types have no variables, schemes, or implicit quantification.
#[derive(Clone, Debug, PartialEq, Eq)]
pub enum Type {
    Int,
    Bool,
    Arrow(Box<Type>, Box<Type>),
}

impl Type {
    pub fn arrow(domain: Self, codomain: Self) → Self {
        Self::Arrow(Box::new(domain), Box::new(codomain))
    }
}

impl fmt::Display for Type {
    fn fmt(&self, f: &mut fmt::Formatter<'_,>) → fmt::Result {
        match self {
            Self::Int ⇒ write!(f, "Int"),
            Self::Bool ⇒ write!(f, "Bool"),
            Self::Arrow(a, b) ⇒ {
                if matches!(**a, Self::Arrow(..)) {
                    write!(f, "({a})")?;
                } else {
                    write!(f, "{a}")?;
                }
                write!(f, " → {b}")
            }
        }
    }
}

```

The grammar uses right recursion for the arrow's codomain. The AST's lambda variant now stores a `Type` beside the parameter name, so a successfully parsed lambda always has its annotation.

```

Type: Type = {
    <a:TypeAtom> "→" <b:Type> ⇒ Type::arrow(a, b),
    TypeAtom,
};
TypeAtom: Type = {
    "Int" ⇒ Type::Int, "Bool" ⇒ Type::Bool, "(" <Type> ")",
};

```

The checker mirrors the typing rules. Its environment is a stack, searched from the innermost binding outward. A scope is popped even if checking its body returns an error. Application first finds the function type, then checks the argument, giving a useful error at the argument rather than at the whole call.

```
fn infer(
  expr: &Expr,
  env: &mut Vec<(String, Type)>,
) → Result<Type, Diagnostic> {
  match &expr.kind {
    Kind::Int(_) ⇒ Ok(Type::Int),
    Kind::Bool(_) ⇒ Ok(Type::Bool),
    Kind::Var(name) ⇒ env
      .iter()
      .rev()
      .find(|(n, _)| n == name)
      .map(|(_, ty)| ty.clone())
      .ok_or_else(|| {
        Diagnostic::new(
          "E0101",
          format!("unbound variable {name}"),
          expr.span.clone(),
        )
      }),
    Kind::Lam(name, parameter, body) ⇒ {
      env.push((name.clone(), parameter.clone()));
      let result = infer(body, env);
      env.pop();
      Ok(Type::arrow(parameter.clone(), result?))
    }
    Kind::App(function, argument) ⇒ {
      let function_ty = infer(function, env)?;
      let Type::Arrow(domain, codomain) = function_ty else {
        return Err(mismatch(
          function,
          "a function type",
          &function_ty,
        ));
      };
      expect(argument, infer(argument, env)?, &domain)?;
      Ok(*codomain)
    }
    Kind::Let(name, value, body) ⇒ {
```

```

    let ty = infer(value, env)?;
    env.push((name.clone(), ty));
    let result = infer(body, env);
    env.pop();
    result
}
Kind::If(condition, yes, no) => {
    expect(condition, infer(condition, env)?, &Type::Bool)?;
    let ty = infer(yes, env)?;
    expect(no, infer(no, env)?, &ty)?;
    Ok(ty)
}
Kind::Neg(value) => {
    expect(value, infer(value, env)?, &Type::Int)?;
    Ok(Type::Int)
}
Kind::Binary(op, left, right) => {
    expect(left, infer(left, env)?, &Type::Int)?;
    expect(right, infer(right, env)?, &Type::Int)?;
    Ok(match op {
        Op::Eq | Op::Lt | Op::Le => Type::Bool,
        _ => Type::Int,
    })
}
Kind::Error => Err(Diagnostic::new(
    "E0002",
    "unrecovered syntax error",
    expr.span.clone(),
)),
}
}

```

The small `expect` helper compares two simple types structurally. On a mismatch, it reports both the expected and found types at the expression's byte span. Ariadne renders the source excerpt. This standalone checker stops at its first type error. The earlier lexical and scope passes still collect their errors.

The evaluator ignores annotations and retains the closure representation from Chapter 3. Substitution, normal-order reduction, and de Bruijn conversion carry types through their lambda cases. This lets the preservation tests check every intermediate term, and keeps annotated terms distinct in nameless dumps.

Running It

```
$ cd crates/ch04_stlc
$ cargo run -q -- run tests/programs/typed-return.fun
42
$ cargo run -q -- dump --pass=types tests/programs/church.fun
Int
$ cargo run -q -- repl
STLC: :quit, :type EXPR, :parse EXPR, :trace EXPR.
stlc> :type \ (f : Int → Bool) → \ (x : Int) → f x
(Int → Bool) → Int → Bool
stlc> :quit
```

check checks without evaluating. Both execution engines check types first. `:trace` also rejects ill-typed input before displaying reductions. A successful check of `1 / 0` is expected. Its type is `Int`, even though execution fails.

Exercises

1. **Easy.** Derive the type of the choose example using the abstraction rule.
2. **Easy.** Explain why `(Int → Bool) → Int` needs its parentheses.
3. **Medium.** Write an application with a well-typed function and an argument of the wrong type. Check the diagnostic's span.
4. **Hard.** State the substitution lemma for this calculus, including the condition on the replacement term, and work through the lambda case.

Further Reading

Pierce gives the formal progress and preservation proofs (Pierce 2002). Damas and Milner motivate the next step beyond simple types: principal type schemes and polymorphic `let` (Damas and Milner 1982). Takenobu Tani's *Type Introduction Illustrated for Casual Haskellers* (pdf / github) draws the same ideas as pictures, from simple types up to the extensions GHC adds.

5

Evaluation

New files

- `src/cek.rs`
- `src/cbn.rs`

Changed files

- `src/eval.rs`
- `src/types.rs`
- `src/ast.rs`
- `src/lexer.rs`
- `src/lib.rs`
- `src/main.rs`
- `src/names.rs`
- `src/normal.rs`

Motivation

A recursive evaluator is a lovely way to explain evaluation. Each Rust call follows one more piece of the program. Unfortunately, each call can also use another piece of Rust's stack. Our little language has inherited a limit we never intended to give it. A bigger stack buys us time, but the problem is still there.

We'll fix this by writing down what the evaluator needs to do next. Instead of leaving that information in Rust call frames, we'll store it in a data structure and drive the whole evaluator with a loop. This is the idea behind an abstract machine.

The machine we'll use is called CEK. The C tells us what we're evaluating, the E records the current variable bindings, and the K records what comes next (Felleisen and Friedman 1986). The

name is more intimidating than the idea. We're turning the evaluator's implicit bookkeeping into something we can inspect.

The Language, Now

A recursive binding gives its type before its right-hand side. That right-hand side must be syntactically a lambda. A value such as `let rec x : Int = x in x` is rejected with E0120.

```
let rec sum : Int → Int → Int =
  \ (acc : Int) → \ (n : Int) →
    if n = 0 then acc else sum (acc + n) (n - 1)
in sum 0 100
```

The self name is available inside the function and its enclosing `let` body. Its annotation is checked against the inferred right-hand-side type. `fix` is also an explicit term form, binding tighter than application:

```
fix (\ (self : Int → Int) → \ (n : Int) →
  if n = 0 then 1 else n * self (n - 1)) 6
```

Both examples are well typed, but only the `sum` is tail recursive. Multiplication must wait for each recursive factorial call to return.

Theory

A fixed point of a function satisfies an unfolding equation. `fix f` behaves as `f (fix f)`. Its type rule requires the argument to return the same type that it accepts (Pierce 2002).

$$\frac{\Gamma \vdash f : \tau \rightarrow \tau}{\Gamma \vdash \text{fix } f : \tau} \text{ (Fix)}$$

With `fix`, a well-typed program can run forever. Try `fix (\ (x : Int) → x)` and watch it use up any fuel budget you give it. The type checker hasn't failed us. It never promised this recursive program would terminate. Recursive `let` uses the same fixed-point idea, but requires a function on the right-hand side so we know how to initialize it. Fun will keep that restriction.

Call-by-value evaluates a function and its argument before entering its body. Call-by-name substitutes an unevaluated argument each time it is used and stops at a lambda. Call-by-need additionally shares the argument's result, requiring a suspended computation and a memoized result. Our small comparison interpreter implements call-by-name without memoization. Its substitution code makes the absence of sharing explicit. Chapter 3's full normal-order reducer also enters lambda bodies, unlike this weak call-by-name evaluator.

Fun chooses call-by-value to make order and allocation easier to follow through its intermediate languages. Constructors will also be strict. A lambda remains a value, so the language can represent an action without performing it yet. Chapter 15 uses this property for IO. Laziness would require another runtime representation and another account of effects and space usage, beyond this book's chosen compiler.

Implementation

The parser checks the recursive-binding restriction and lowers an accepted binding through this constructor:

```
/// Lower function-only recursive bindings to a fixed point with an annotated
self parameter.
pub fn recursive(
    name: String,
    ty: Type,
    rhs: Expr,
    body: Expr,
    start: usize,
    end: usize,
) → Result<Expr, Diagnostic> {
    if !matches!(rhs.kind, Kind::Lam(..)) {
        return Err(Diagnostic::new(
            "E0120",
            "recursive binding must be a function",
            rhs.span,
        ));
    }
    let self_function =
        Expr::new(Kind::Lam(name.clone(), ty, Box::new(rhs)), start, end);
    let fixed = Expr::new(Kind::Fix(Box::new(self_function)), start, end);
    Ok(Expr::new(
        Kind::Let(name, Box::new(fixed), Box::new(body)),
        start,
        end,
    ))
}
```

`Value::Recursive` stores the function used to construct a fixed point. Reading that recursive variable applies the stored function to its recursive reference. For a recursive function this rebuilds the closure in its original environment. It does not form a Rust ownership cycle. Entering `fix` evaluates its defining function immediately, so errors in that initialization are still strict. This is a teaching implementation of explicit recursion, not a general lazy binding mechanism.

cek.rs separates control from continuation frames. The frame variants are the places where the tree evaluator would otherwise wait for a recursive call.

```
/// The remaining work after the current expression returns a value.
enum Frame {
    Argument(Rc<Expr>, Rc<Env>),
    Apply(Value),
    Let(String, Rc<Expr>, Rc<Env>),
    If(Rc<Expr>, Rc<Expr>, Rc<Env>),
    Neg,
    Left(Op, Rc<Expr>, Rc<Env>),
    Right(Op, i64),
    Fix,
}

/// Control alternates between evaluating an expression and returning a value.
enum Control {
    Expr(Rc<Expr>, Rc<Env>),
    Value(Value),
    Apply(Value, Value),
}

/// Statistics make tail-space behavior testable without relying on timing.
pub struct ResultValue {
    pub value: Value,
    pub steps: usize,
    pub max_continuations: usize,
}
```

An application first pushes an Argument frame while evaluating its function. That frame then becomes an Apply frame while evaluating the argument. Once both values are available, the machine enters the body in the closure’s captured environment extended with the parameter. The application frames have already been popped:

```
Control::Apply(
    Value::Closure(name, body, captured),
    argument,
) => {
    let mut env = (*captured).clone();
    env.push((name, argument));
    Control::Expr(body, Rc::new(env))
}
Control::Apply(_, _) => {
    return Err("expected a function".into());
}
```

Look at the sum again. Its recursive call already produces the answer, so there's nothing to remember for the caller. Factorial still has a multiplication to finish. That pending work needs a frame.

Both versions now avoid growing Rust's call stack. Only the tail-recursive version avoids growing our explicit continuation stack as well. We check this by comparing the maximum continuation depth of a short run with a million-call run. Finishing on one generously sized machine wouldn't tell us nearly as much.

The complete transition loop follows the expression forms and pops one frame when a value returns. `max_continuations` is measured before every transition, and a fuel counter makes divergent examples safe to inspect. Fuel counts machine transitions, so its numerical value differs from the tree evaluator's count of expression visits.

The tiny call-by-name evaluator uses substitution instead of an environment for its application and `let` cases:

```
Kind::App(f, a) => {
    let function = evaluate(f, fuel)?;
    if let Kind::Lam(name, _, body) = function.kind {
        evaluate(&substitute(&body, &name, a), fuel)
    } else {
        Err("expected a function".into())
    }
}
Kind::Let(n, a, b) => evaluate(&substitute(b, n, a), fuel),
```

A call-by-need variant would replace this substitution with a shared cell whose first demand computes and saves the argument. We leave that implementation for future work. The production path here remains the strict machine, and the comparison interpreter is selected explicitly.

Running It

```
$ cd crates/ch05_eval
$ cargo run -q -- run tests/programs/sum-tail.fun
5050
$ cargo run -q -- run --engine=core tests/programs/factorial.fun
720
$ cargo run -q -- dump --pass=types tests/programs/fix-factorial.fun
Int
$ cargo run -q -- run --engine=cbn tests/programs/strict-argument.fun
1
$ cargo test -q a_million_tail_calls_use_bounded_continuations
```

The default engine is `cek`. `core` selects the recursive closure evaluator, `cbn` selects the comparison interpreter, and `normal` retains the earlier full reducer. `--fuel=N` bounds execution. Use a modest fuel budget for the tree interpreters. CEK is the one that removes their dependence on the host stack. The two strict engines should agree on both answers and failures. The call-by-name comparison deliberately differs when an unused argument would fail. Try it before reading the implementation and predict which error, if any, you'll see.

Exercises

1. **Easy.** Draw the continuation frames for $1 + 2 * 3$ as each operand returns.
2. **Medium.** Change the tail sum into a non-tail sum and measure both maximum continuation depths.
3. **Medium.** Explain why a closure must capture its definition environment rather than its caller's environment.
4. **Hard.** Give an expression for which call-by-name repeats substantial work that call-by-value computes once. Explain what sharing would change.

Further Reading

Felleisen and Friedman develop the abstract-machine account (Felleisen and Friedman 1986). Pierce relates evaluation strategies, typing, and fixed points (Pierce 2002). Appel develops the related technique of making continuations explicit during compilation (Appel 1992). Our ANF compiler returns to that connection later.

6

Hindley–Milner Inference

New files

- `src/algorithm_w.rs`
- `src/union_find.rs`

Changed files

- `src/types.rs`
- `src/ast.rs`
- `src/cbn.rs`
- `src/cek.rs`
- `src/eval.rs`
- `src/lexer.rs`
- `src/lib.rs`
- `src/main.rs`
- `src/names.rs`
- `src/normal.rs`

Motivation

Look at $\lambda x \rightarrow x$. We haven't said what type x has, but we already know something useful. Whatever goes in is exactly what comes back out. Its type must have the shape $a \rightarrow a$.

That's pretty cool! We learned something about every possible call without running the function once. An integer, a boolean, or another function would all work. A *type scheme* lets each use choose its own a . Hindley–Milner inference finds the most general scheme that the program permits (Damas and Milner 1982).

We’ll implement it in two ways. Algorithm W keeps an explicit record of the type substitutions it discovers (Damas and Milner 1982). Our second implementation updates mutable cells and uses levels to track which variables can be generalized. They solve the same equations, but make different implementation choices. Working through both will help us understand the algorithm before we choose the version we’ll carry into Fun.

The Language, Now

Poly has unannotated lambdas, application, polymorphic `let`, function-only `let rec`, integers, booleans, conditionals, and arithmetic. The explicit `fix` form remains available from the evaluation chapter for comparison exercises.

```
let id = \x → x in
let first = id 1 in
id True
```

The two occurrences of `id` instantiate its scheme independently. Lambda parameters behave differently. `\f → let first = f 1 in f True` is rejected, because `f` has one type throughout that lambda body.

We can drop the parameter annotations from the previous chapter. In particular, identity no longer has to choose between accepting integers and booleans. We’ll keep the evaluation machinery we already understand and concentrate on how the compiler discovers that freedom.

Theory

How to read a typing rule

A page of typing rules can look forbidding the first time you see one. The notation is mostly there to save us from repeating the same sentences. Each rule explains one piece of syntax. Start below the bar to see what we’re trying to establish, then look above it to see what we need to know about the parts. Pierce uses this style throughout his treatment of type systems (Pierce 2002).

$$\Gamma \vdash e : \tau$$

Read this as “under the assumptions in gamma, expression `e` has type tau.” The symbols keep repeated sentences short:

Notation	Meaning
e	A program expression, such as x , 42 , or $f\ x$.
τ, α, β	A type, or a type variable standing for a type.
σ	A type scheme, possibly quantifying type variables.
Γ	The typing environment: names and the types or schemes we know for them.
$\Gamma, x : \tau$	Extend the environment with a local assumption about x .
$\vdash$	“Under these assumptions, we can establish ...”
$\alpha \rightarrow \beta$	A function accepting an α and returning a β .
$\forall \alpha. \alpha \rightarrow \alpha$	A scheme whose user may choose the type replacing α .

The colon asserts a type. It does not evaluate the expression. Gamma contains static assumptions, not runtime values. In this chapter it maps let-bound names to schemes and lambda parameters to monotypes, represented as schemes with no quantifiers. Greek variables in the rules are mathematical placeholders. The inferencer’s mutable metavariables are an implementation technique for finding which types satisfy the rules.

The horizontal bar means “if everything above holds, then the statement below holds.” It is not division. A rule with an empty top has no premises. Integer literals have type `Int` without consulting a subexpression.

$$\frac{}{\Gamma \vdash 42 : \text{Int}} (\text{Int})$$

For application, two facts must agree on the same input type:

$$\frac{\Gamma \vdash e_1 : \alpha \rightarrow \beta \quad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash e_1\ e_2 : \beta} (\text{App})$$

If e_1 accepts an integer and returns a boolean, e_2 must be an integer and the call returns a boolean. The repeated α expresses that agreement. During inference we discover it by unification. The rule describes a valid relationship. It does not prescribe an order in which to solve the premises.

Build a small derivation

Consider $\backslash x \rightarrow x + 1$. Give x a provisional type. The addition rule requires both operands to be integers, which fixes that provisional type to `Int`. Writing $\Delta = \Gamma, x : \text{Int}$ keeps the resulting tree compact:

$$\frac{\frac{x:\text{Int} \in \Delta}{\Delta \vdash x:\text{Int}} (\text{Var}) \quad \frac{}{\Delta \vdash 1:\text{Int}} (\text{Int})}{\Delta \vdash x + 1:\text{Int}} (\text{Add})$$

The two leaves justify Add by lookup and the literal rule. The lambda rule then moves the assumption about the parameter into a function type:

$$\frac{\Gamma, x:\alpha \vdash e:\beta}{\Gamma \vdash \lambda x. e:\alpha \rightarrow \beta} (\text{Lam})$$

Here $\lambda x. e$ means $\backslash x \rightarrow e$. Lam gives our function type $\text{Int} \rightarrow \text{Int}$ and removes the body's local assumption about x . Replacing 1 with `True` breaks an Add premise, so we can't complete the derivation.

From monotypes to schemes

The interesting part of `let id = \x → x in ...` is that each use of `id` gets to choose its own argument type. We record that permission in a *scheme*. A monotype describes one type, perhaps with variables we haven't solved yet. A scheme marks which variables a caller may choose afresh. Making those fresh copies is *instantiation*. Deciding which variables can safely be offered to callers is *generalization* (Damas and Milner 1982).

We can't generalize a variable that the surrounding scope still depends on. Otherwise a local function could pretend to accept any type while quietly using an outer value whose type was already fixed. A lambda parameter therefore starts with one monotype shared by all its uses.

$$\frac{\Gamma \vdash e_1:\tau \quad \sigma = \text{gen}(\Gamma, \tau) \quad \Gamma, x:\sigma \vdash e_2:\beta}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2:\beta} (\text{Let})$$

Read Let in three steps. First infer the binding's right-hand side, then generalize its type. Finally check the body with that scheme in scope. The equation for σ is a side condition defining that scheme, rather than another program to evaluate. A variable is *free* in a type or environment when it is not bound by a scheme's quantifier. Only variables absent from the environment's free variables may be generalized.

For `let id = \x → x in ...`, the right-hand side has type $\alpha \rightarrow \alpha$. If α is independent of the surrounding environment, we bind `id` to $\forall \alpha. \alpha \rightarrow \alpha$. Each lookup instantiates this scheme afresh. `id 1` chooses `Int`. `id True` chooses `Bool`. By contrast, the parameter `f` in `\f → ...` receives one monotype from Lam. All its uses must satisfy that same type. Quantification belongs to the let rule, not to every name that happens to denote a function.

Unification solves equality constraints between types. Two arrows unify by unifying their domains, applying what that learned, and unifying their codomains. A variable can be solved by a type only if it does not occur inside that type. Without this occurs check, self-application would require an infinite type. It is reported as E0202. Incompatible base types produce E0201.

A recursive function has one provisional monotype while its body is checked. Only after that check may the surrounding `let` generalize it. Our lowering to `fix` expresses this rule through the ordinary monomorphic self parameter. There is no polymorphic recursion here. The environment-based generalization rule and its efficient level-based form are developed in the cited accounts (Kiselyov 2013; Rémy 1992).

Poly has no mutable state, so a `let` right-hand side need not be a syntactic value to generalize. For example, applying an identity to another identity still produces a polymorphic binding. This unrestricted rule will also apply to `Fun`.

A Small Payoff

Composition is a lovely example of how much a short definition can say.

```
let compose = \f → \g → \x → f (g x) in
compose (\x → x + 1) (\x → x * 2) 20
```

The result is 41. More interestingly, `compose` itself knows nothing about integers. Its principal type is $(b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$. The output of `g` is the input of `f`, so those two positions share `b`. Nothing requires `a`, `b`, and `c` to be the same type. Two applications in the definition give us that whole story. This is the kind of thing that makes inference worth building.

Implementation

The algorithms share immutable output types and schemes, but no unifier or inference traversal. Variable identities are canonicalized by first occurrence only after inference. That makes their principal types comparable up to renaming.

```

/// Variables are fresh identities; algorithms choose their own representation
of solutions.
#[derive(Clone, Debug, PartialEq, Eq)]
pub enum Type {
    Int,
    Bool,
    Var(u32),
    Arrow(Box<Type>, Box<Type>),
}

/// Quantified variables are freshened independently at every occurrence.
#[derive(Clone, Debug)]
pub struct Scheme {
    pub variables: Vec<u32>,
    pub ty: Type,
}

impl Scheme {
    pub fn mono(ty: Type) → Self {
        Self {
            variables: vec![],
            ty,
        }
    }
}

impl Type {
    pub fn arrow(a: Self, b: Self) → Self {
        Self::Arrow(Box::new(a), Box::new(b))
    }

    /// Rename by first occurrence, making principal types comparable up to
    alpha-renaming.
    pub fn canonical(&self) → Self {
        fn go(ty: &Type, names: &mut BTreeMap<u32, u32>) → Type {
            match ty {
                Type::Var(n) ⇒ {
                    let next = names.len() as u32;
                    Type::Var(*names.entry(*n).or_insert(next))
                }
                Type::Arrow(a, b) ⇒ {
                    Type::arrow(go(a, names), go(b, names))
                }
                other ⇒ other.clone(),
            }
        }
        go(self, &mut BTreeMap::new())
    }
}

```

Algorithm W

`algorithm_w.rs` separates substitution, instantiation, generalization, and inference. The core rules extend to Poly's arithmetic, conditional, and fixed-point forms, with source spans retained for diagnostics.

A substitution maps variables to types. Composition means applying the earlier substitution first, then rewriting its result with the newer one. Applying a substitution to a scheme must protect the scheme's quantified variables.

```

impl Subst {
  fn apply(&self, ty: &Type) → Type {
    match ty {
      Type::Var(n) ⇒ {
        self.0.get(n).map_or_else(|| ty.clone(), |t| self.apply(t))
      }
      Type::Arrow(a, b) ⇒ Type::arrow(self.apply(a), self.apply(b)),
      other ⇒ other.clone(),
    }
  }

  /// self after earlier: every earlier range is rewritten by the newer
  substitution.
  fn compose(&self, earlier: &Self) → Self {
    let mut result: BTreeMap<_, _> =
      earlier.0.iter().map(|(n, t)| (*n, self.apply(t))).collect();
    for (n, t) in &self.0 {
      result.entry(*n).or_insert_with(|| t.clone());
    }
    Self(result)
  }

  fn environment(&self, env: &Env) → Env {
    env.iter()
      .map(|(name, scheme)| {
        let mut filtered = self.clone();
        for n in &scheme.variables {
          filtered.0.remove(n);
        }
        (
          name.clone(),
          Scheme {
            variables: scheme.variables.clone(),
            ty: filtered.apply(&scheme.ty),
          },
        )
      })
      .collect()
  }
}

```

Application threads information learned from the function into the environment used to infer its argument. It then unifies the function’s type with an arrow from the argument’s type to a fresh result.

```
Kind::App(f, a) => {
  let (s1, tf) = self.infer(env, f)?;
  let (s2, ta) = self.infer(&s1.environment(env), a)?;
  let result = self.fresh();
  let s3 = unify(
    &s2.apply(&tf),
    &Type::arrow(ta, result.clone()),
    expr,
  );
  Ok((s3.compose(&s2.compose(&s1)), s3.apply(&result)))
}
```

The `let` case applies the right-hand side's substitution to the environment before generalizing. Generalizing against the old environment would allow an already constrained variable to escape.

```
Kind::Let(n, value, body) => {
  let (s1, t) = self.infer(env, value)?;
  let mut local = s1.environment(env);
  local.insert(n.clone(), generalize(&local, s1.apply(&t)));
  let (s2, result) = self.infer(&local, body)?;
  Ok((s2.compose(&s1), result))
}
```

Mutable solutions and levels

`union_find.rs` replaces the explicit substitution map with metavariable cells. A cell records its creation level and an optional solution. Following a solved variable compresses its path, so later visits can reach the resolved type sooner.

```

#[derive(Clone)]
struct Meta {
    level: u32,
    solution: Option<Type>,
}

#[derive(Default)]
struct Infer {
    level: u32,
    metas: Vec<Meta>,
}

impl Infer {
    fn fresh(&mut self) → Type {
        fn fresh(&mut self) → Type {
            let n = self.metas.len() as u32;
            self.metas.push(Meta {
                level: self.level,
                solution: None,
            });
            Type::Var(n)
        }
    }
    fn zonk(&mut self, ty: &Type) → Type {
        match ty {
            Type::Var(n) ⇒ {
                match self.metas[*n as usize].solution.clone() {
                    None ⇒ ty.clone(),
                    Some(t) ⇒ {
                        let t = self.zonk(&t);
                        self.metas[*n as usize].solution = Some(t.clone());
                        t
                    }
                }
            }
            Type::Arrow(a, b) ⇒ Type::arrow(self.zonk(a), self.zonk(b)),
            other ⇒ other.clone(),
        }
    }
}

```

Inferring a `let` right-hand side enters a deeper level. Before installing a solution for an older metavariable, the occurs-check traversal lowers every metavariable reachable from that solution to the older level. These variables now belong to the outer scope, even if they were allocated inside the `let`.

```

/// One traversal rejects cycles and prevents younger variables escaping their
scope.
fn occurs_and_lower(
    &mut self,
    n: u32,
    level: u32,
    ty: &Type,
    expr: &Expr,
) → Result<(), Diagnostic> {
    match self.zonk(ty) {
        Type::Var(m) if m == n ⇒ Err(types::occurs(expr)),
        Type::Var(m) ⇒ {
            self.metas[m as usize].level =
                self.metas[m as usize].level.min(level);
            Ok(())
        }
        Type::Arrow(a, b) ⇒ {
            self.occurs_and_lower(n, level, &a, expr)?;
            self.occurs_and_lower(n, level, &b, expr)
        }
        _ ⇒ Ok(()),
    }
}

fn unify(
    &mut self,
    a: &Type,
    b: &Type,
    expr: &Expr,
) → Result<(), Diagnostic> {
    let (a, b) = (self.zonk(a), self.zonk(b));
    match (&a, &b) {
        _ if a == b ⇒ Ok(()),
        (Type::Var(n), t) | (t, Type::Var(n)) ⇒ {
            self.occurs_and_lower(
                *n,
                self.metas[*n as usize].level,
                t,
                expr,
            )?;
            self.metas[*n as usize].solution = Some(t.clone());
            Ok(())
        }
        (Type::Arrow(a, b), Type::Arrow(c, d)) ⇒ {

```

```

        self.unify(a, c, expr)?;
        self.unify(b, d, expr)
    }
    _ => Err(types::mismatch(expr, &a, &b)),
}
}

```

After returning to the enclosing level, generalization selects only unsolved variables whose levels are still deeper. It does not scan the environment. Every occurrence of a scheme replaces its quantified variables with fresh cells. Unquantified variables retain their identities and therefore their constraints.

```

Kind::Let(n, value, body) => {
    self.level += 1;
    let result = self.infer(env, value);
    self.level -= 1;
    let scheme = self.generalize(result?);
    env.push((n.clone(), scheme));
    let result = self.infer(env, body);
    env.pop();
    result
}

```

Try $\lambda x \rightarrow \text{let } y = \lambda z \rightarrow x \ z \text{ in } \dots$. The type of y depends on x , so we mustn't let two calls choose unrelated types for it. Calling y with both an integer and a boolean should fail. This is a useful example to keep beside the level-lowering code. It shows exactly what goes wrong if we generalize too much. We compare both inferencers on examples like this, including programs that should be rejected.

Running It

```
$ cd crates/ch06_poly
$ cargo run -q -- run tests/programs/poly-id.fun
True
$ cargo run -q -- dump --pass=types tests/programs/missing-annotation.fun
a → a
$ cargo run -q -- dump --pass=types-w tests/programs/missing-annotation.fun
a → a
$ cargo run -q -- repl
Poly: :quit, :type EXPR, :parse EXPR, :trace EXPR.
poly> :type \f → \x → f x
(a → b) → a → b
poly> :quit
```

types and the normal checking path use the level-based inferencer. `types-w` selects Algorithm W's independent result. Free variables in the displayed principal type are implicitly quantified. Evaluation still uses the strict CEK machine by default. Changing inference does not change its runtime values.

Exercises

1. **Easy.** Infer the type of $\lambda x \rightarrow \lambda y \rightarrow x$ by writing down fresh variables first.
2. **Medium.** Trace the substitutions for `let id =  $\lambda x \rightarrow x$  in id id`.
3. **Medium.** Explain why `$\lambda f \rightarrow \text{let } a = f\ 1 \text{ in } f\ \text{True}$` fails while the opening `id` example succeeds.
4. **Hard.** Draw the metavariable cells and levels for an inner function that captures an outer lambda parameter. Identify the unification that lowers a level.

Further Reading

Damas and Milner establish principal type schemes (Damas and Milner 1982). Kiselyov explains levels through the OCaml implementation (Kiselyov 2013), building on Rémy's work (Rémy 1992). Pierce provides a broader introduction to typing judgments and derivations (Pierce 2002).

See my alternative writeup about just type systems for another route through these ideas.

Part II: Frontend

7

Design of Fun

New files

- `src/session.rs`
- `src/core.rs`
- `src/typecheck.rs`
- `src/elaborate.rs`
- `src/diagnostics/codes.rs`
- `src/diagnostics/mod.rs`
- `src/parse.rs`
- `src/source.rs`

Changed files

- `src/eval.rs`
- `src/ast.rs`
- `src/lexer.rs`
- `src/lib.rs`
- `src/main.rs`
- `src/names.rs`
- `src/types.rs`

Motivation

We've built a parser, explored how functions run, and taught a compiler to infer types. Now we can bring those pieces together. From this chapter onward we'll be building Fun, the language we'll eventually compile to native code. Each chapter keeps a runnable snapshot, so you can always return to a simpler version when you want to see where an idea began.

Some of the work here is the ordinary engineering that makes the interesting parts pleasant to build. When the type checker finds a mistake, it needs to point back to the source. When one pass fails, the next shouldn't try to make sense of its unfinished output. We'll give names, source locations, and errors a shared home, then connect our passes into a compiler we can keep extending.

The Language, Now

A program is an explicit brace block containing declarations. Semicolons separate declarations. A trailing semicolon is optional. Each declaration may refer to itself and to earlier declarations. Forward references and mutual recursion wait until Chapter 9.

```
{
  twice f x = f (f x);
  sum n = if n == 0 then 0 else n + sum (n - 1);
  main = twice (\x → x + 1) (sum 10);
}
```

Local `let` uses braces too, with one binding in this snapshot. Recursive bindings need no `rec` keyword. A self-reference identifies recursion, and the right-hand side must be a lambda. `let { x = x; } in x` is rejected with E0120.

`Char` contains a Unicode scalar value, and `String` is a primitive UTF-8 string. Strings concatenate with the monomorphic `++` operator. Integers accept hexadecimal notation and underscores. Both line comments and nested block comments are supported.

```
{ {- outer {- nested -} -} main = "Hello, " ++ "λ\n"; }
```

`run` still prints the value of `main`, including quotes and escapes for strings. It does not yet execute IO. Integer `div` and `mod` are curried built-in functions with floor semantics: `div 7 (-3)` is `-3`, and `mod 7 (-3)` is `-2`. The old calculator's truncating division fixture changes accordingly.

Theory

A source program and an executable intermediate language serve different purposes. The source tree keeps spelling, syntax, and source spans. Core keeps only the operations needed for evaluation. Elaborating one into the other makes language conveniences explicit. Later we add types to Core so that a separate checker can validate each transformation (Pierce 2002).

Strictness remains a semantic requirement. The function is evaluated before its argument, operator operands are evaluated left to right, and a conditional selects only one branch. Top-level values are initialized once, in declaration order in this snapshot, before the final value of `main` is selected. Even an unused failing initializer stops the program. Top-level function bodies wait until application.

The current type system remains Hindley–Milner, using the level-based checker from Chapter 6 (Damas and Milner 1982; Kiselyov 2013). The recursive-binding restriction avoids constructing a value before its own initialization has completed. Recursive functions instead refer to their own closure when entered. Ordinary name scopes still use spelling-based maps here. Unique binding identities become operational in Chapter 9.

Implementation

Source and session state

`source.rs` gives a file its own identity and stores a line-start table. Spans are half-open byte ranges, matching the lexer, while displayed columns count Unicode scalar values. The file identity prevents equal offsets in different files from being confused.

```
/// Index of a source file in a session's source map.
#[derive(Clone, Copy, Debug, PartialEq, Eq, Hash)]
pub struct FileId(pub u32);

/// Half-open byte offsets within one source file.
#[derive(Clone, Copy, Debug, PartialEq, Eq)]
pub struct Span {
    pub file: FileId,
    pub start: u32,
    pub end: u32,
}

impl Span {
    pub fn new(file: FileId, start: usize, end: usize) → Self {
        Self {
            file,
            start: start.try_into().expect("source exceeds 4 GiB"),
            end: end.try_into().expect("source exceeds 4 GiB"),
        }
    }
    pub fn range(self) → Range<usize> {
        self.start as usize..self.end as usize
    }
    pub fn through(self, other: Self) → Self {
        assert_eq!(self.file, other.file);
        Self {
            end: other.end,
            ..self
        }
    }
}
```

```

    }
}

/// Original text and line starts; columns count Unicode scalar values.
pub struct SourceFile {
    pub name: String,
    pub text: String,
    lines: Vec<usize>,
}

#[derive(Default)]
/// Owns every source used by this compilation, including future Prelude
input.
pub struct SourceMap {
    pub files: Vec<SourceFile>,
}

impl SourceMap {
    pub fn add(&mut self, name: impl Into<String>, text: String) → FileId {
        assert!(u32::try_from(text.len()).is_ok(), "source exceeds 4 GiB");
        let mut lines = vec![0];
        lines.extend(text.match_indices('\n').map(|(i, _)| i + 1));
        let id = FileId(
            self.files.len().try_into().expect("too many source files"),
        );
        self.files.push(SourceFile {
            name: name.into(),
            text,
            lines,
        });
        id
    }

    pub fn get(&self, file: FileId) → &SourceFile {
        &self.files[file.0 as usize]
    }

    pub fn line_col(&self, span: Span) → (usize, usize) {
        let file = self.get(span.file);
        let offset = span.start as usize;
        let line = file.lines.partition_point(|start| *start ≤ offset) - 1;
        (
            line + 1,
            file.text[file.lines[line]..offset].chars().count() + 1,
        )
    }
}

```

```
    }
}
```

`names.rs` supplies a handwritten interner and fresh-name supply. Repeated spellings share a `Symbol`. Repeated calls to the supply produce distinct `Name` identities. These definitions establish the interface for the later renamer without changing lexical scoping prematurely.

A session owns these facilities along with options and a diagnostic sink:

```
#[derive(Default)]
/// No pass needs a global interner, source map, or diagnostic sink.
pub struct Session {
    pub sources: SourceMap,
    pub interner: Interner,
    pub supply: Supply,
    pub options: Options,
    pub diagnostics: Vec<Diagnostic>,
}

impl Session {
    pub fn emit(&mut self, diagnostic: Diagnostic) {
        self.diagnostics.push(diagnostic);
    }
    pub fn finish<T>(&self, value: T) → Result<T, Stop> {
        if self
            .diagnostics
            .iter()
            .any(|d| d.severity == Severity::Error)
        {
            Err(Stop)
        } else {
            Ok(value)
        }
    }
}
```

Each pass takes `&mut Session` and returns `Result<Output, Stop>`. A `Stop` carries no second copy of the diagnostics. The driver renders the accumulated structured errors and exits before calling the next pass. Lexing and duplicate-definition checks collect errors. Inference stops where a type error prevents a meaningful result.

`diagnostics/codes.rs` is the stable registry for errors and warnings, including codes reserved for upcoming features. A diagnostic has a severity, message, source labels, and notes. Ariadne renders

it. The first two stderr lines remain the stable code and primary source location used by the corpus runner.

Surface checking and Core

`lexer.rs` uses logos, including callbacks for nested comments and literal escapes. Its reserved-word and operator tables also generate the HTML and PDF syntax definitions through `cargo xtask syntax`. The book build checks these outputs for drift.

`parser.lalrpop` builds spanned expressions and top-level bindings. Equation parameters become nested lambdas, and duplicate parameters are rejected. The checker processes declarations in source order. For a self-dependent binding, it introduces a provisional monotype while checking the lambda, unifies the result, then generalizes as in the previous chapter.

`elaborate.rs` lowers the checked tree into a separate Core representation:

```

#[derive(Clone, Debug, PartialEq, Eq)]
/// Immediate source values; strings are primitive UTF-8 values, not character
lists.
pub enum Lit {
    Int(i64),
    Bool(bool),
    Char(char),
    String(String),
}

#[derive(Clone, Debug)]
/// The initial lambda Core; types, data constructors, and compiled cases
arrive later.
pub enum Expr {
    Var(String),
    Lit(Lit),
    Lam(String, Box<Expr>),
    App(Box<Expr>, Box<Expr>),
    Let(Bind, Box<Expr>),
    If(Box<Expr>, Box<Expr>, Box<Expr>),
    Prim(Op, Vec<Expr>, Span),
}

#[derive(Clone, Debug)]
/// A recursive binding is guaranteed to have a lambda right-hand side.
pub struct Bind {
    pub name: String,
    pub rhs: Box<Expr>,
    pub recursive: bool,
}

#[derive(Clone, Debug)]
/// Globals retain source order in this chapter; SCC ordering arrives in
Chapter 9.
pub struct Program {
    pub bindings: Vec<Bind>,
    pub main: String,
}

```

Operators become saturated primitive calls. Negation becomes subtraction from zero. The initial built-in environment provides curried wrappers for `div` and `mod`. `--no-prelude` removes those wrappers for isolated tests. The source Prelude arrives with datatypes. Source names beginning with `prim_` are rejected with `E0111` outside that future Prelude path.

--lint currently checks Core scope, primitive saturation, and the recursive lambda invariant. It is structural lint, not a type checker. Chapter 10 will replace it with the full System F check. The golden harness enables it on every run.

Reference evaluation

eval.rs interprets Core directly. A recursive closure stores its self name without creating a cyclic Rust environment. Application inserts that closure into its captured environment before the parameter:

```
Expr::App(function, argument) => {
    let function = evaluate(function, env, fuel)?;
    let argument = evaluate(argument, env, fuel)?;
    if let Value::Closure {
        parameter,
        body,
        env,
        recursive,
    } = &function
    {
        let mut local = (**env).clone();
        if let Some(name) = recursive {
            local.push((name.clone(), function.clone()));
        }
        local.push((parameter.clone(), argument));
        evaluate(body, &local, fuel)
    } else {
        Err("internal error: non-function application".into())
    }
}
```

The program runner initializes every top-level binding in order and retains those values in its environment. Integers normalize to 63 bits after arithmetic, and division checks zero before computing its floor quotient and matching remainder. Character and string printing share the source literal escape rules.

This reference evaluator is a recursive tree walker on a 512 MiB thread stack. It has a fuel limit for experiments, but no constant-stack guarantee. The explicit machine from Chapter 5 returns as an ANF oracle in Chapter 18. Keeping Core's reference interpreter direct makes the forthcoming elaboration passes easier to inspect.

Running It

```
$ cd crates/ch07_fun
$ cargo run -q -- run tests/programs/fun-top.fun
42
$ cargo run -q -- dump --pass=types tests/programs/fun-string.fun
main : String
$ cargo run -q -- dump --pass=core --no-prelude tests/programs/fun-char.fun
val main = 'λ'
$ cargo run -q -- repl
Fun: expressions, :type EXPR, :core EXPR, :quit.
fun> :type \x → x
main : a → a
fun> :quit
```

Try the tokens, parse, types, and core dumps on the same small program. Each shows a different stage of the journey. You can use `dump --pass=...` or the common `--dump=...` option. The REPL evaluates individual expressions for now. Persistent definitions will arrive with the JIT.

Exercises

1. **Easy.** Use `--dump=tokens` to compare a nested comment with the operator `→`.
2. **Medium.** Add an unused top-level initializer that divides by zero. Explain why `main` never produces a value.
3. **Medium.** Inspect Core for a local recursive function and explain where the self reference becomes bound.
4. **Hard.** Construct an invalid Core program that passes source parsing but could only be produced by a faulty compiler transformation. Extend a lint test to reject it.

Further Reading

The Haskell Report provides the lexical and scoping reference that Fun gradually adopts (S. Marlow 2010). Pierce develops the typed intermediate-language perspective (Pierce 2002).

8

Layout and the Extended Parser

New files

- `src/layout.rs`
- `src/fixity.rs`
- `src/format.rs`

Changed files

- `src/parse.rs`
- `src/typecheck.rs`
- `src/ast.rs`
- `src/elaborate.rs`
- `src/lexer.rs`
- `src/lib.rs`
- `src/main.rs`
- `src/types.rs`

Motivation

The braces were useful while we were getting started. By now they're mostly making us type punctuation for structure we can already see in the indentation. Let's make Fun a little nicer to write.

We'll add layout, local helpers with `where`, type signatures, and user-defined operators. Much of this can happen before evaluation. Indentation becomes punctuation, sections become lambdas, and operator chains acquire their grouping once we know the declared fixities. It's a satisfying compiler trick. We can make the source language more expressive while keeping its evaluator small.

The Language, Now

Braces and semicolons become optional when indentation supplies their boundaries. A `where` block belongs to its equation. The equation's parameters remain visible inside it. Several local declarations may share a `let` block.

```
addTwo x = x + y
  where
    y = 2

main = let result = addTwo 40 in result
```

Signatures can appear before or after their binding within the same block. Type variables are implicitly quantified. A signature may specialize an inferred type, but it cannot make the implementation more polymorphic than it is.

```
id :: a → a
id x = x

main = let { n :: Int; n = id 42; } in n
```

Operators may be defined in prefix or infix form. Their fixities apply throughout the file, even before the fixity declaration. An undeclared operator associates to the left at precedence 9. Named functions become operators inside backticks.

```
infixr 5 %%

a %% b = a * 10 + b

main = 1 %% 2 %% 3
```

This program prints 33 because it means `1 %% (2 %% 3)`. Sections supply one argument: `(40 +)` and `(+ 2)` are functions. `(- 2)` instead means negation. Unary minus has precedence 6, so `-2 * 3 + 4` means `(-(2 * 3)) + 4`. Write parentheses around negation used after an operator of precedence 6 or greater.

Theory

Layout is easier to understand once we stop treating whitespace as mysterious. We're going to translate indentation into the braces and semicolons our parser already knows. The Haskell Report describes this translation (S. Marlow 2010). Let's do it by hand before looking at the adapter.

See the Invisible Delimiters

Think of indentation as a shorthand for punctuation. The first declaration in a block sets its left margin. Another declaration at that margin begins a new item. Moving back to an enclosing margin finishes the block.

```
scale x = result
  where
    doubled = x + x
    result = doubled * 3

main = let n = 7 in scale n
```

Before parsing, layout makes the omitted punctuation explicit. Here every brace and semicolon was inserted by the layout pass:

```
{ scale x = result
  where { doubled = x + x
        ; result = doubled * 3
        }
; main = let { n = 7 } in scale n
}
```

The file gets an outer pair of braces. `doubled` establishes the `where` block's indentation, and `result` at the same column gets a separator before it. `main` is farther left, so layout closes the `where` block and separates the next top-level declaration. The same-line `in` closes the `let` block even though there was no dedent. The final brace closes the file at end of input. Writing the braces yourself produces the same grouping.

The later pattern and IO chapters use the same mechanism after `of` and `do`:

```
main = do
  case flag of
    True  → putStrLn "yes"
    False → putStrLn "no"
  putStrLn "done"
```

Its explicit form is:

```

{ main = do
  { case flag of
    { True → putStrLn "yes"
    ; False → putStrLn "no"
    }
    ; putStrLn "done"
  }
}

```

The dedent before the last action closes only the `of` block. That action is still inside `do`. These constructs arrive later, but the rule for translating their indentation is already the same. Use `--dump=layout` to see the actual tokens, including the spans that distinguish inserted punctuation from source characters.

Fixity resolution compares adjacent operators before constructing applications. A greater precedence binds first. Equal precedences require compatible associativities. Two left-associative operators reduce from the left, and two right-associative operators from the right. Non-associative operators and mixed associativities at the same precedence require parentheses (S. Marlow 2010).

A signature makes a promise. If we write `a → a`, we mean the function works for any `a` its caller chooses. The checker mustn't make that promise true by quietly choosing `a = Int`.

We enforce this by replacing the signature's quantified variables with rigid identities, called *skolems*. Inference variables can be solved, but a skolem stands for an arbitrary type and must stay arbitrary. We instantiate the inferred scheme with fresh flexible variables and try to unify it with the rigid signature. We also check that no skolem escapes into the surrounding scope. Together these checks tell us whether the inferred type supports the promise (Damas and Milner 1982; Pierce 2002).

Implementation

Layout and trivia

The logos lexer emits comment tokens with their original text. `TriviaLexer` records them in a side table and exposes only semantic tokens. `LayoutLexer` then performs indentation insertion. The adapter groups explicit braces and parentheses, recursively handling layout inside them, so a parenthesized `let` still gets its own block.

```

pub fn layout(
    sess: &mut Session,
    file: FileId,
) → Result<Vec<Spanned>, Stop> {
    let tokens = lexer::lex(sess, file)?;
    let source = &sess.sources.get(file).text;
    let eager = !matches!(tokens.first(), Some( (_, Token::Open, _)));
    // offside measures byte columns with fixed-width tabs. Give it a shadow
    // source whose byte positions are visual columns, then restore original
    spans.
    let (shadow, forward, backward) = visual_source(source);
    let tokens = tokens
        .into_iter()
        .map(|(lo, t, hi)| (forward[lo], t, forward[hi]))
        .collect::<Vec<_>>();
    match region(&shadow, &tokens, eager)
        .map(|tokens| {
            tokens
                .into_iter()
                .map(|(lo, t, hi)| (backward[lo], t, backward[hi]))
                .collect()
        })
        .map_err(|(lo, hi, message)| (backward[lo], backward[hi], message))
    {
        Ok(tokens) ⇒ Ok(tokens),
        Err((lo, hi, message)) ⇒ {
            sess.emit(Diagnostic::error(
                ErrorCode::Parse,
                message,
                Span::new(file, lo, hi),
            ));
            Err(Stop)
        }
    }
}

```

The library does not expose a `let/in` closure operation. The adapter tracks which virtual frames have already closed at `in`, suppressing their later separators and closing braces. It also handles empty blocks and redundant virtual separators beside explicit semicolons. Tokens keep their original byte offsets. Inserted punctuation has zero-width spans.

`of` and `do` participate in lexical layout tests now. Their language constructs still wait for Chapters 11 and 15, respectively. The compiler does not accept those expressions prematurely.

Operator chains

The grammar retains a first operand followed by operator/operand pairs. Parentheses remain explicit until the resolver has established their boundary. A stack resolves these chains. A distinguished unary entry gives negation its precedence.

```

fn chain(
    &mut self,
    first: Expr,
    pairs: Vec<(String, Expr)>,
) → Result<Expr, Diagnostic> {
    let mut values = Vec::new();
    let mut operators: Vec<Option<String>> = Vec::new();
    self.operand(first, &mut values, &mut operators)?;
    for (op, value) in pairs {
        let next = self.fix(&op);
        while let Some(previous) = operators.last() {
            let prior =
                previous.as_ref().map(|p| self.fix(p)).unwrap_or(Fix {
                    precedence: 6,
                    assoc: Assoc::Left,
                });
            if prior.precedence < next.precedence {
                break;
            }
            if prior.precedence == next.precedence && previous.is_some()
            {
                if prior.assoc != next.assoc
                || prior.assoc == Assoc::None
                {
                    return Err(self.error(value.span, "operators with
conflicting fixities need parentheses"));
                }
                if prior.assoc == Assoc::Right {
                    break;
                }
            }
            let prior = operators.pop().unwrap();
            self.reduce(&mut values, prior);
        }
        operators.push(Some(op));
        self.operand(value, &mut values, &mut operators)?;
    }
    while let Some(op) = operators.pop() {
        self.reduce(&mut values, op);
    }
    Ok(values.pop().unwrap())
}

```

Sections become lambdas with generated names that source identifiers cannot spell. The current primitive operators remain direct arithmetic nodes when unshadowed. User operators become ordinary curried applications. Core therefore needs no new expression forms.

Checking signatures

Declarations are collected before signatures are attached to bindings. The checker first infers and generalizes a binding, then tests its signature in a trial store. A successful check specializes the live scheme used by subsequent occurrences. Recursive calls remain monomorphic during inference. Signature-guided polymorphic recursion arrives with the later signature checker.

```

fn check_signature(
    &mut self,
    inferred: &Scheme,
    declared: &Scheme,
    expr: &Expr,
) → Result<Scheme, Diagnostic> {
    fn rigid(ty: &Type) → Type {
        match ty {
            Type::Var(n) ⇒ Type::Skolem(*n),
            Type::Arrow(a, b) ⇒ Type::arrow(rigid(a), rigid(b)),
            t ⇒ t.clone(),
        }
    }
    fn has_skolem(ty: &Type) → bool {
        match ty {
            Type::Skolem(_) ⇒ true,
            Type::Arrow(a, b) ⇒ has_skolem(a) || has_skolem(b),
            _ ⇒ false,
        }
    }
    let mut trial = self.clone();
    let actual = trial.instantiate(inferred);
    let expected = rigid(&declared.ty);
    let invalid = trial.unify(&actual, &expected, expr).is_err()
        || (0..self.metas.len()).any(|n| {
            self.metas[n].level ≤ self.level
            && has_skolem(&trial.zonk(&Type::Var(n as u32)))
        });
    if invalid {
        return Err(Diagnostic::error(
            ErrorCode::Signature,
            format!(
                "signature {} is not an instance of inferred type {}",
                declared.ty,
                inferred.ty.canonical()
            ),
            expr.span,
        ));
    }
    // Specialization can constrain variables captured from an outer scope.
    self.level += 1;
    let actual = self.instantiate(inferred);
    let expected = self.instantiate(declared);
    let result = self.unify(&actual, &expected, expr);
    self.level -= 1;
    result?;
    Ok(self.generalize(expected))
}

```

Formatting

The formatter uses `marginalia`'s document algebra and attached comment map. Its canonical output uses explicit blocks and keeps source parentheses. Comments move to separate leading lines at token boundaries, preserving their text and order. This avoids inserting punctuation inside a line comment.

```
pub fn format(sess: &mut Session, file: FileId) → Result<String, Stop> {
    let parsed = parse::parse(sess, file)?;
    crate::fixity::resolve(sess, parsed)?;
    let (semantic, trivia) = lexer::trivia(sess, file)?;
    let tokens = layout::layout(sess, file)?;
    let source = &sess.sources.get(file).text;
    let mut comments = attach(
        source,
        &trivia,
        semantic.iter().map(|(lo, _, hi)| span(*lo, *hi)),
        AttachOptions::default(),
    );
    // Tokens are smaller anchors than statements. Put their comments on
    // separate leading lines so a trailing line comment cannot swallow a
    // newly inserted separator or a following `where`.
    for (lo, _, hi) in &semantic {
        let entry = comments.entry(span(*lo, *hi));
        entry.leading.append(&mut entry.trailing);
    }
    let mut cursor = 0;
    let doc = document(source, &tokens, &mut cursor, false);
    let rendered = pretty::render(&doc, &comments, RenderOpts::default());
    Ok(format!("{}", rendered.trim_end()))
}
```

The tests compare parsed trees after removing spans, rather than merely comparing results of evaluation. Every syntactically valid corpus program must round-trip, and formatting the output again must produce exactly the same bytes. Dedicated cases retain nested block comments and comments beside `where`.

Running It

```
$ cd crates/ch08_layout
$ cargo run -q -- run tests/programs/layout/right-fixity.fun
33
$ cargo run -q -- dump --pass=fixity tests/programs/layout/negation.fun
{
  main = -(2 * 3) + 4;
}
$ cargo run -q -- fmt example.fun
$ cargo run -q -- fmt example.fun --check
```

The three dumps make a useful little tour. `--dump=layout` reveals the inserted punctuation, `--dump=parse` shows operator chains before grouping, and `--dump=fixity` shows the chosen grouping. Try changing only the indentation or a fixity declaration and watch which stage first changes.

Exercises

1. **Introductory.** predict the grouping of three operators with distinct fixities, then compare the fixity dump with your answer.
2. **Intermediate.** Explain why `(1 + 2 *)` is rejected and repair it with parentheses.
3. **Intermediate.** Add a formatter test with nested comments between an operator and its operand. Check both tree equality and comment retention.
4. **Advanced.** Construct a local signature whose skolem would escape through a captured lambda parameter. Trace the trial-store check that rejects it.
5. **Advanced.** Compare the restricted in rule with the Report's general `parse-error(t)` rule. Identify syntax for which the latter would do more work.

Further Reading

The Report specifies layout and fixity resolution precisely (S. Marlow 2010). Damas and Milner establish the principal-type property used by signature checks (Damas and Milner 1982). Pierce develops the distinction between polymorphic types and their instances in a broader type-theoretic setting (Pierce 2002).

9

The Renamer

New files

- `src/rename.rs`
- `src/scc.rs`
- `src/renamed.rs`

Changed files

- `src/names.rs`
- `src/typecheck.rs`
- `src/ast.rs`
- `src/core.rs`
- `src/elaborate.rs`
- `src/eval.rs`
- `src/fixity.rs`
- `src/lib.rs`
- `src/main.rs`
- `src/parse.rs`
- `src/types.rs`

Motivation

Two variables can both be called `x` and mean entirely different things. That's easy enough to follow in a small expression. It gets less comfortable when the compiler starts moving expressions around. We'd like to settle which binding each occurrence means once, then carry that answer through the rest of the compiler.

That’s the renamer’s first job. Its second is to free us from writing helpers before the functions that use them. Some functions even call each other, so there is no possible ordering that puts every helper first. We’ll find those groups in a dependency graph and give the type checker the whole group at once.

The Language, Now

Top-level and local declarations may refer forward within their block. These two functions belong to the same recursive component:

```
even n = if n == 0 then True  else odd  (n - 1)

odd  n = if n == 0 then False else even (n - 1)

main = even 100
```

Each binding block is one lexical scope. A local definition shadows a global with the same spelling. Two definitions in the same block are an E0102 error. Every member of a recursive component must have a function right-hand side. Thus `a = b; b = a` is rejected with E0120, even if neither value is used.

Fixities also receive a scope check. A top-level fixity declaration must have a matching definition in the file (E0130). An unbound variable reports E0101 and suggests an in-scope spelling when its edit distance is at most two.

Theory

A binding identity consists of an interned spelling and a fresh integer. The spelling serves diagnostics. The unique integer separates shadowed binders. Inference and Core use these identities, so neither has to repeat lexical lookup. This is the compiler counterpart of the binding distinctions explored in Chapter 3 (Pierce 2002).

Draw an arrow from a function to each binding in its block that it uses. If `even` calls `odd` and `odd` calls `even`, we can follow arrows from either one back to the other. They belong together. A *strongly connected component* is a maximal group with this mutual-reachability property.

Now treat each group as one node. The resulting graph has no cycles, so we can process its dependencies first. That’s the order our type checker needs. A helper outside a recursive group can have its type generalized before its callers use it (S. Marlow 2010).

Inside a recursive component, allocate one provisional monotype per binder and infer all bodies against them. Unify those provisional types with the results. Only then generalize the component. Outside the component, each occurrence can instantiate the resulting schemes independently. Sig-

natures still check inferred types here. Signature-directed polymorphic recursion remains Chapter 14's work.

Implementation

Name resolution

The surface tree retains a whole local declaration block until renaming. The renamer allocates all of that block's identities before visiting any right-hand side. Each lambda introduces a smaller scope containing its parameter. Lookup searches scopes from the innermost outward.

```

pub fn rename(
    sess: &mut Session,
    program: ast::Program,
) → Result<Program, Stop> {
    let program = crate::fixity::resolve(sess, program)?;
    for fixity in &program.fixities {
        for op in &fixity.operators {
            if !program.bindings.iter().any(|b| b.name == *op) {
                sess.emit(Diagnostic::error(
                    ErrorCode::OrphanFixity,
                    format!("fixity without a definition: {op}"),
                    fixity.span,
                ));
            }
        }
    }
    let mut primitives = Vec::new();
    let mut prelude = BTreeMap::new();
    if !sess.options.no_prelude {
        for op in [
            Op::Div,
            Op::Mod,
            Op::Add,
            Op::Sub,
            Op::Mul,
            Op::Append,
            Op::Eq,
            Op::Lt,
            Op::Le,
        ] {
            let id = sess.fresh_id(op.text());
            prelude.insert(op.text().into(), id.clone());
            primitives.push((op, id));
        }
    }
    let mut renamer = Renamer {
        sess,
        scopes: vec![prelude],
    };
    let groups = renamer.bindings(program.bindings);
    let main = renamer.lookup("main", program.span);
    renamer.sess.finish(Program {
        groups,
        primitives,
        main,
        span: program.span,
    })
}

```

The renamed tree is a separate IR. Its occurrences contain an `Id`, which keeps a `Name` and its diagnostic spelling together. Dumps display names as `x#12`. This small amount of repeated text keeps dumps readable after the source session has gone away. Comparisons still include the allocated identity.

Errors accumulate during the walk. An unresolved occurrence receives a fresh placeholder so other bodies can still be checked for scope errors. The driver stops after the renaming pass, before any placeholder can reach evaluation.

Dependency components

Tarjan's search assigns each vertex an index and records the lowest reachable index still active on the search stack. When the two indices agree, the vertex is the root of a complete component, which can be popped together.

```
fn visit(&mut self, v: usize) {
    self.index[v] = Some(self.next);
    self.low[v] = self.next;
    self.next += 1;
    self.stack.push(v);
    self.active[v] = true;
    for &w in &self.edges[v] {
        if self.index[w].is_none() {
            self.visit(w);
            self.low[v] = self.low[v].min(self.low[w]);
        } else if self.active[w] {
            self.low[v] = self.low[v].min(self.index[w].unwrap());
        }
    }
    if self.low[v] == self.index[v].unwrap() {
        let mut group = Vec::new();
        loop {
            let w = self.stack.pop().unwrap();
            self.active[w] = false;
            group.push(w);
            if w == v {
                break;
            }
        }
        group.sort_unstable();
        self.result.push(group);
    }
}
```

Our edges point toward dependencies, so completed components arrive in the order inference needs. We visit source indices in a fixed order to keep dumps reproducible. A group containing just one binding is recursive only if that binding refers to itself. As a check on Tarjan's bookkeeping, the tests also find components by the much slower but direct method of testing mutual reachability.

Inference by group

The environment contains all provisional types while a recursive component is being inferred. Generalization waits until every body has contributed constraints. Signatures are checked against the inferred schemes, and specialization propagates through the shared component before the exported schemes are finalized.

```
fn group(
    &mut self,
    env: &mut Vec<Id, Scheme>,
    group: &BindGroup,
) → Result<(), Diagnostic> {
    self.level += 1;
    let size = env.len();
    let provisional: Vec<_> = group
        .binds
        .iter()
        .map(|b| (b.id.clone(), self.fresh()))
        .collect();
    if group.recursive {
        env.extend(
            provisional
                .iter()
                .map(|(id, ty)| (id.clone(), Scheme::mono(ty.clone()))),
        );
    }
    let inferred = group
        .binds
        .iter()
        .zip(&provisional)
        .map(|(b, (_, expected))| {
            let ty = self.infer(env, &b.rhs)?;
            if group.recursive {
                self.unify(expected, &ty, &b.rhs)?;
            }
            Ok(ty)
        })
        .collect::<Result<Vec<_>, Diagnostic>>();
    env.truncate(size);
}
```

```

    self.level -= 1;
    let inferred = inferred?;
    for (binding, ty) in group.binds.iter().zip(&inferred) {
        if let Some(annotation) = &binding.annotation {
            let scheme = self.generalize(ty.clone());
            self.check_signature(&scheme, annotation, &binding.rhs)?;
        }
    }
    self.level += 1;
    for (binding, ty) in group.binds.iter().zip(&inferred) {
        if let Some(annotation) = &binding.annotation {
            let declared = self.instantiate(annotation);
            self.unify(ty, &declared, &binding.rhs)?;
        }
    }
    self.level -= 1;
    for (binding, ty) in group.binds.iter().zip(inferred) {
        env.push((binding.id.clone(), self.generalize(ty)));
    }
    Ok(())
}

```

That last step matters. If `f` and `g` call one another and `f` has an `Int → Int` signature, a shared argument type must also constrain `g`. Exporting an unconstrained polymorphic scheme for `g` would be unsound. A regression test rejects a subsequent `g True` call.

Recursive closures

The interpreter stores an immutable recipe for every recursive group: its lambda bodies and the outer environment. Entering one of these closures reconstructs the group's bindings before adding the function argument. No closure needs a mutable pointer to itself, and there is no reference-counting cycle.

```
fn recursive_values(bindings: Rc<Vec<Bind>>, env: Rc<Env>) → Env {
    let mut values = Vec::new();
    for binding in bindings.iter() {
        let Expr::Lam(parameter, body) = &*binding.rhs else {
            unreachable!("recursive lambda invariant")
        };
        let value = Value::Closure {
            parameter: parameter.clone(),
            body: Rc::new(*body.clone()),
            env: env.clone(),
            recursive: Some(bindings.clone()),
        };
        values.push((binding.name.clone(), value));
    }
    values
}
```

Top-level value initializers run once in dependency order. Functions are closures. Creating one does not run its body. Core lint now checks unique binder identities as well as scope, primitive saturation, and the recursive-lambda restriction.

Running It

```
$ cd crates/ch09_rename
$ cargo run -q -- run tests/programs/rename/mutual.fun
True
$ cargo run -q -- dump --pass=rename tests/programs/fun-forward.fun
nonrec { later#10 = 1; }
nonrec { main#9 = later#10; }
$ cargo xtask check-inherit 09
chapter 09: 101 inherited programs verified and chapter tests passed
```

Try moving the helper in a small program below its caller. It now works. Then make the two functions call each other and inspect the rename dump. You should see one recursive group, which tells the checker to infer them together.

Exercises

1. **Introductory.** annotate each occurrence of `x` in a nested-shadowing example with the binder it references. Compare with the rename dump.
2. **Intermediate.** Draw the dependency graph of a three-function recursive group with one non-recursive helper, then predict the emitted component order.

3. **Intermediate.** Add a diagnostic test with two independent misspelled names and verify that the pass reports both.
4. **Advanced.** Explain why the immutable closure recipe has no reference-counting cycle even when one of its functions returns another closure.
5. **Advanced.** Identify where signature-directed polymorphic recursion will require the current component construction and checking strategy to change.

Further Reading

The Haskell Report describes binding groups and dependency analysis (S. Marlow 2010). Pierce supplies the underlying treatment of variables, scope, and binding (Pierce 2002).

10

Typed Core and Core Lint

New files

- `src/core_lint.rs`
- `src/core_type.rs`

Changed files

- `src/core.rs`
- `src/elaborate.rs`
- `src/typecheck.rs`
- `src/eval.rs`
- `src/lib.rs`
- `src/main.rs`
- `src/rename.rs`
- `src/renamed.rs`

Motivation

The type checker has accepted our program. Excellent. Now the compiler gets to rewrite it, which gives us plenty of opportunities to introduce mistakes of our own. We might pass the wrong argument, forget a type instantiation, or reuse a variable identity. The source program being correct doesn't help if we translate it incorrectly.

We'll give Core explicit types and write a second checker for it. This checker has an easier job than inference. All the answers are already in the tree, so it only needs to verify that they fit together. When a transformation goes wrong, we can catch it close to where it happened. A compiler that checks its own homework is a rather useful thing to have.

The Language, Now

We can still write polymorphic functions without spelling out their type arguments.

```
id x = x

number = id 42

main = id True
```

In Core, the definition of `id` abstracts over a type variable. Its two uses apply it first to `Int` and then to `Bool`. Those applications are real IR nodes rather than comments on an untyped expression. The evaluator ignores their types, so this program still prints `True`.

Theory

The source expression `id 42` hides a small decision. We’re using identity at `Int`. System F gives us a way to write that decision into the program. A *type abstraction* introduces a type parameter, and a *type application* chooses a type for it. Girard and Reynolds developed this explicit treatment of polymorphism (Girard 1972; Reynolds 1974).

Think of the two new forms alongside ordinary lambdas and calls. An ordinary lambda waits for a value. A type abstraction abstracts over a type. This analogy helps us read the rules, although the type arguments will disappear before runtime.

$$\frac{\Gamma, \alpha \vdash e : \tau \quad \alpha \notin \text{ftv}(\Gamma)}{\Gamma \vdash \Lambda \alpha. e : \forall \alpha. \tau} \text{ (Type abstraction)}$$

$$\frac{\Gamma \vdash e : \forall \alpha. \tau \quad \Gamma \vdash \sigma : *}{\Gamma \vdash e[\sigma] : \tau[\sigma/\alpha]} \text{ (Type application)}$$

Every type binder has kind `*` in this chapter. Higher-kinded constructors will extend the kind judgment later. Ordinary application still requires an arrow type. A universally quantified function must first receive its type arguments.

Core supports full System F. Fun still exposes only HM polymorphism. Hand-built Core terms test that distinction without extending source syntax (Pierce 2002).

Implementation

The Core Datatypes

We can write the same language as a small grammar before looking at the Rust. Read each vertical bar as “or”. An expression e is one of the forms below, and the right-hand column tells us what that form does.

$e = x$	local variable
$ g$	global reference
$ \ell$	literal
$ e_1 e_2$	application
$ \lambda x : \tau. e$	term abstraction
$ \Lambda \alpha. e$	type abstraction
$ e[\tau]$	type application
$ \text{let } b \text{ in } e$	binding group
$ \text{if } e_1 \text{ then } e_2 \text{ else } e_3$	conditional
$ p[\bar{\tau}](\bar{e})$	primitive operation

Here x and g name local and global values, ℓ is a literal, and p is a primitive. An overbar means a sequence, so $\bar{e}$ stands for the primitive's operands in evaluation order. We leave source spans out of the notation, though the Rust representation keeps them for diagnostics.

The types and binding groups have their own grammar.

$\tau = \text{Int} \mid \text{Bool} \mid \text{Char} \mid \text{String}$	base types
$ \alpha$	type variable
$ \tau_1 \rightarrow \tau_2$	function type
$ \forall \alpha. \tau$	universal type
$b = x : \tau = e$	nonrecursive binding
$ \text{rec } \{\bar{x} : \tau = \bar{e}\}$	recursive group

A recursive group's names are in scope throughout its right-hand sides. Lint also checks that those right-hand sides describe functions, possibly wrapped in type abstractions. In a nonrecursive binding, the new name is available only in the body of the `let`.

Notice the two different lambdas. In $\Lambda \alpha. \lambda x : \alpha. x$, the capital Λ chooses no value at all. It introduces the type parameter used by the ordinary λ . Applying this identity to a type and then a value gives $(\Lambda \alpha. \lambda x : \alpha. x)[\text{Int}]$ 42. That's the explicit version of the source-level `id` 42 we started with.

A Core expression is a Rust enum. Each variable and term binder carries an `Id` with its type, so lint can check the tree without rerunning inference.

```

/// A Core identifier carries its independently checkable type.
#[derive(Clone, Debug, PartialEq, Eq)]
pub struct Id {
    pub name: crate::names::Id,
    pub ty: Type,
}

```

The expression constructors distinguish term-level functions (Lam, App) from type-level abstraction and instantiation (TyLam, TyApp). Let keeps the recursive components produced by the renamer. Prim carries explicit type arguments, evaluated operands, and a source span for failures.

```

#[derive(Clone, Debug)]
/// System F Core makes type abstraction and instantiation explicit.
pub enum Expr {
    Var(Id),
    Global(Id),
    TyLam(u32, Box<Expr>),
    TyApp(Box<Expr>, Type),
    Lit(Lit),
    Lam(Id, Box<Expr>),
    App(Box<Expr>, Box<Expr>),
    Let(BindGroup, Box<Expr>),
    If(Box<Expr>, Box<Expr>, Box<Expr>),
    Prim(Op, Vec<Type>, Vec<Expr>, Span),
}

```

Binding groups hold typed names and right-hand sides. A program is an ordered list of these groups together with the entry point:

```

#[derive(Clone, Debug)]
/// A recursive binding is guaranteed to have a lambda right-hand side.
pub struct Bind {
    pub name: Id,
    pub rhs: Box<Expr>,
}
#[derive(Clone, Debug)]
/// An ordered component; recursive members are in scope in every right-hand
side.
pub struct BindGroup {
    pub recursive: bool,
    pub binds: Vec<Bind>,
}
#[derive(Clone, Debug)]
/// Globals are initialized in dependency order, with explicit recursive
components.
pub struct Program {
    pub groups: Vec<BindGroup>,
    pub main: Id,
}

```

This chapter still has `If`. Datatype constructors and pattern matches arrive in Chapter 11, and flat cases and join points arrive in Chapter 16. The smaller enum here lets us establish typed elaboration and lint before adding those forms.

Recording inference results

Renamed expressions now have stable node numbers. The checker records the fresh type arguments used at each variable occurrence, the scheme of each binding, and the monotype of each lambda parameter. Before returning, it zonks all these types, so elaboration never needs access to the mutable metavariable store.

A variable occurrence inferred inside a recursive group initially sees a provisional monotype. Once the component is generalized, elaboration instantiates its recursive `Core` references at the corresponding generalized variables. Ordinary occurrences use the fresh arguments recorded during inference.

Elaborating polymorphism

Each generalized metavariable receives a fresh `Core` type identity. Binder types wrap the converted monotype in `Forall`. Right-hand sides wrap their terms in the matching `TyLams`. Local schemes can still mention type variables bound by an outer function’s abstraction.

```

fn group(
    &mut self,
    group: &renamed::BindGroup,
    outer: &BTreeMap<u32, u32>,
    global: bool,
) → BindGroup {
    for binding in &group.binds {
        self.binder(
            binding.id.clone(),
            &self.typed.schemes[&binding.id],
            outer,
            global,
        );
    }
    let binds = group
        .binds
        .iter()
        .map(|binding| {
            let binder = self.binders[&binding.id].clone();
            let mut variables = outer.clone();
            variables.extend(binder.quantifiers.iter().copied());
            let mut rhs = self.expression(&binding.rhs, &variables);
            for (_, variable) in binder.quantifiers.iter().rev() {
                rhs = Expr::TyLam(*variable, Box::new(rhs));
            }
            Bind {
                name: binder.id,
                rhs: Box::new(rhs),
            }
        })
        .collect();
    BindGroup {
        recursive: group.recursive,
        binds,
    }
}

```

Occurrences gain TyApps, and lambda parameters carry their converted types. Core distinguishes global references from local variables. Its binder annotations are checked against the declared environment, so attaching a plausible-looking type to an otherwise wrong occurrence cannot evade lint.

An internal variable absent from every enclosing scheme needs a closed witness: $(\lambda f \rightarrow 42)$ $(\lambda x \rightarrow x)$ is an example. We use `Int` for these erased annotations. This does not default source expressions or resolve ambiguous class constraints.

Independent lint

The checker maintains term and type scopes, plus sets of binder identities already used. It checks arrow applications, primitive saturation, conditional branches, recursive function groups, and binder annotations. These are the explicit type abstraction and application rules:

```
Expr::TyLam(variable, body) => {
  if !self.seen_types.insert(*variable) {
    return Err(format!(
      "duplicate Core type binder t{variable}"
    ));
  }
  self.types.insert(*variable);
  let body = self.expression(body)?;
  self.types.remove(variable);
  Ok(Type::Forall(*variable, Box::new(body)))
}
Expr::TyApp(fun, argument) => {
  self.well_formed(argument)?;
  let function = self.expression(fun)?;
  let Type::Forall(variable, body) = function else {
    return Err(format!(
      "Core type application of non-polymorphic type {function}"
    ));
  };
  Ok(body.substitute(variable, argument))
}
```

Type application substitutes into the quantified body. Substitution must avoid capturing a free type variable in the argument, even for Core terms constructed outside our elaborator. The implementation freshens a conflicting binder:

```

pub fn substitute(&self, variable: u32, replacement: &Type) → Type {
  match self {
    Self::Var(v) if *v == variable ⇒ replacement.clone(),
    Self::Arrow(a, b) ⇒ Self::arrow(
      a.substitute(variable, replacement),
      b.substitute(variable, replacement),
    ),
    Self::Forall(v, _) if *v == variable ⇒ self.clone(),
    Self::Forall(v, body) ⇒ {
      let (v, body) = if replacement.free().contains(v) {
        let fresh = self
          .largest()
          .max(replacement.largest())
          .max(variable)
          .checked_add(1)
          .expect("type variable supply exhausted");
        (fresh, body.substitute(*v, &Self::Var(fresh)))
      } else {
        (*v, *body.clone())
      };
      Self::Forall(
        v,
        Box::new(body.substitute(variable, replacement)),
      )
    }
    _ ⇒ self.clone(),
  }
}

```

Type equality is alpha equivalence, so two forall types may use different fresh identities and still denote the same type. In contrast, a free type variable must refer to the same enclosing binder.

```

pub fn lint(program: &Program) → Result<(), String> {
  let mut checker = Checker::default();
  for group in &program.groups {
    checker.group(group, true)?;
  }
  checker.expression(&Expr::Global(program.main.clone()))?;
  Ok(())
}

```

ASIDE A compiler bug that drops TyApp from an application of polymorphic identity produces Core application of non-function type forall t0. t0 → t0. The source program may be valid. The generated Core is not. The malformed-Core tests check this failure directly, without rerunning source inference.

Erasure and strict execution

TyLam and TyApp do not allocate runtime objects. The interpreter steps through them without substituting runtime values. This is erasure, rather than a separate runtime evaluation strategy for System F.

In particular, a type abstraction does not delay a strict initializer. The unused polymorphic binding below still fails before main is selected:

```
unused = (\x → \y → y) (div 1 0)

main = 42
```

Chapter 18 introduces an explicit erased IR. Corpus runs already enable `--lint`.

Running It

```
$ cd crates/ch10_core
$ cargo run -q -- run tests/programs/core/self-instantiation.fun --lint
42
$ cargo run -q -- dump --pass=core --no-prelude tests/programs/layout/
signature.fun
nonrec { id#0 : forall t0. t0 → t0 = (type-lambda t0 (lambda x#2 : t0
x#2)); }
nonrec { main#1 : Int = (app (type-app @id#0 Int) 42); }
$ cargo test -p ch10_core --test system_f
```

For a useful experiment, remove a type application from a small Core term and run lint. The source checker has no role in this test. We're deliberately feeding the second checker a mistake that only the compiler could have made. It should reject the term before evaluation.

Exercises

1. **Introductory.** identify every type abstraction and type application in the Core dump for `id id 42`.
2. **Intermediate.** Explain why a captured outer type variable remains free in a local binding's scheme but is bound in the complete Core program.

3. **Intermediate.** Construct a malformed Core primitive call with the right number of arguments but the wrong operand type.
4. **Advanced.** Demonstrate a capture bug in type substitution if freshening is removed, then explain why alpha equivalence alone does not repair it.

Further Reading

Girard's thesis and Reynolds's paper are the original sources for explicit polymorphism (Girard 1972; Reynolds 1974). Pierce develops System F's typing and substitution rules (Pierce 2002).

11

Datatypes and Pattern Matching

New files

- `src/data.rs`
- `src/syn.rs`
- `src/desugar.rs`

Changed files

- `src/typecheck.rs`
- `src/core.rs`
- `src/elaborate.rs`
- `src/ast.rs`
- `src/core_lint.rs`
- `src/core_type.rs`
- `src/eval.rs`
- `src/fixity.rs`
- `src/layout.rs`
- `src/lexer.rs`
- `src/lib.rs`
- `src/main.rs`
- `src/parse.rs`
- `src/rename.rs`
- `src/renamed.rs`
- `src/session.rs`
- `src/types.rs`

Motivation

Integers and functions have taken us a long way, but it's time to build some proper data structures. A binary tree can be empty or contain a value and two subtrees. A lookup can fail or return a result. Algebraic datatypes let us say these things directly, without inventing special values to mean “nothing here”.

Pattern matching gives us an equally direct way to take the values apart. The shape of the data becomes the shape of the program. This is one of the beautiful parts of functional programming, and we'll use it constantly from here on.

We'll first implement matching by trying patterns in source order. That gives us a clear account of what a match means. In Chapter 16 we'll turn the same matches into decision trees suitable for native code.

The Language, Now

Declarations introduce nominal types and their constructors. Constructor arguments are strict, and constructors can be passed around as curried functions.

```
data Tree a = Empty | Node (Tree a) a (Tree a)

size Empty = 0
size (Node left _ right) = 1 + size left + size right

main = size (Node Empty 42 Empty)
```

Read `size` beside the datatype. `Empty` contributes zero. A `Node` contributes one, plus the sizes of its two children. The definition follows the data almost word for word. That's beautiful, and it scales to much more interesting trees.

We can mirror a tree just as directly.

```
data Tree a = Empty | Node (Tree a) a (Tree a)

mirror Empty = Empty
mirror (Node left x right) = Node (mirror right) x (mirror left)

main = mirror (Node Empty 1 (Node Empty 2 Empty))
```

Notice that `mirror` never examines `x`. Its type is `Tree a → Tree a`, so it works for trees of any element type. The recursion changes the shape while leaving the values alone.

Equations may have nested constructor patterns, literals, wildcards, and as-patterns. Lists, pairs, triples, and unit have dedicated syntax. Guards must have type `Bool`. If all guards fail, matching continues with the next equation.

```
select all@(x : xs) | x < 0 = xs
select all = all

main = select [-1,2,3]
```

A `where` block belongs to its entire equation, including every guard. A `case` alternative may also have guards and a `where` block. Pattern lambdas and strict, non-recursive pattern bindings work as well: `let (x,y) = (20,22) in x + y`. Types remain first-order here. A datatype parameter has kind `*`, so `Tree Int` is valid but applying a parameter as in `f Int` waits for Chapter 13.

The initial Prelude is embedded from this actual Fun source:

```
data Bool = False | True

data Ordering = LT | EQ | GT

data Maybe a = Nothing | Just a

data Either a b = Left a | Right b

data [] a = [] | (:) a [a]

data () = ()

data (,) a b = (,) a b

data (,,) a b c = (,,) a b c
```

Theory

A datatype declaration supplies a polymorphic scheme for each constructor. For example, `Node` has type `Tree a → a → Tree a → Tree a`, quantified over `a`. Each occurrence instantiates that scheme with fresh metavariables, just as a polymorphic function occurrence does (S. Marlow 2010; Pierce 2002).

For a constructor with instantiated field types, a pattern judgment returns the bindings introduced by its subpatterns:

$$\frac{K : \tau_1 \rightarrow \tau_2 \rightarrow T \quad \Gamma \vdash p_1 : \tau_1 \Rightarrow \Delta_1 \quad \Gamma \vdash p_2 : \tau_2 \Rightarrow \Delta_2}{\Gamma \vdash K \ p_1 \ p_2 : T \Rightarrow \Delta_1, \Delta_2} \text{ (Constructor pattern)}$$

The combined pattern environment must not bind a name twice. Each row is checked against the same scrutinee types, and every successful right-hand side must have the same result type. Guards additionally unify with `Bool`.

`Fun` evaluates fields before constructing a value. A failed match raises a runtime error. There are no lazy patterns or suspended constructor fields. Exhaustiveness and redundancy warnings require a separate analysis. They arrive with match compilation (Maranget 2007).

Implementation

Declaring names before checking fields

`data.rs` first allocates every type head, then converts constructor fields. This permits mutually recursive datatypes. The immutable table stores each type’s arity, each constructor’s scheme, and declaration-order tags. Duplicate names, unbound field variables, and incorrect type arities are diagnosed before expression inference begins.

The renamer resolves constructor names in expressions and patterns separately from ordinary variables. Pattern variables receive fresh identities for each row. Its scope then contains those variables and the row’s local declarations while visiting both guards and right-hand sides.

Lowering syntax into Syn

`desugar.rs` introduces an explicit pass between renaming and inference. Lists and tuples become constructor applications. `if` becomes a two-row match on `True` and `False`. Infix `&&` and `||` first become conditionals. As first-class values those operators are ordinary eta-expanded functions, whose arguments are evaluated by call-by-value application.

A single equation containing only variable patterns can use ordinary lambdas and `lets`. Other equations become a lambda for each argument followed by a match:

```

let params: Vec<_> = (0..rows[0].patterns.len())
  .map(|_| self.sess.fresh_id("$match"))
  .collect();
let scruts = params
  .iter()
  .map(|id| self.node(Kind::Var(id.clone()), span))
  .collect();
let mut body = self.node(Kind::Match(scruts, rows), span);
for id in params.into_iter().rev() {
  body = self.node(Kind::Lam(id, Box::new(body)), span);
}
body

```

Strict pattern bindings use one checked bundle of their bound values, represented by nested pairs, followed by projections. The right-hand side is evaluated once. Its newly bound names are hidden while renaming that right-hand side, so an outer binding of the same spelling remains accessible. An unused top-level pattern binding still executes and can fail.

Inferring constructor patterns

The checker instantiates a constructor scheme, separates its field arrows from its result type, and unifies the result with the scrutinee. It then checks each subpattern against the corresponding instantiated field:

```

PatKind::Con(id, patterns) => {
  let info = self.data.con(id).clone();
  if patterns.len() != info.fields.len() {
    return Err(Diagnostic::error(
      ErrorCode::Mismatch,
      format!(
        "{} expects {} pattern fields",
        id.name,
        info.fields.len()
      ),
      pattern.span,
    ));
  }
  let mut ty = self.instantiate(&info.scheme);
  let mut fields = Vec::new();
  for _ in patterns {
    let Type::Arrow(a, b) = ty else {
      unreachable!("constructor arity")
    };
    fields.push(*a);
    ty = *b;
  }
  self.unify(expected, &ty, &site)?;
  for (pattern, ty) in patterns.iter().zip(fields) {
    self.pattern(env, pattern, &ty)?;
  }
}

```

Pattern variables enter the row environment monomorphically. Local bindings retain ordinary let-polymorphism. The checker records expression and pattern-binder types alongside the instantiation metadata already needed by System F elaboration.

Extending Core and lint

Elaboration eta-expands constructor values into lambdas whose bodies are saturated Con nodes. Strict lets evaluate match scrutinees once and in source order. Core keeps nested patterns until the decision-tree compiler is introduced:

```

/// Nested patterns remain until the decision-tree compiler in Chapter 16.
#[derive(Clone, Debug)]
pub enum Pat {
    Var(Id),
    Wild,
    As(Id, Box<Pat>),
    Con(crate::data::ConId, Vec<Pat>),
    Lit(Lit),
}

/// A guarded row falls through when all its Boolean tests fail.
#[derive(Clone, Debug)]
pub enum RowBody {
    Plain(Box<Expr>),
    Guarded(Vec<(Expr, Expr)>),
}

/// Row-local declarations run after patterns match and are shared by the
guards.
#[derive(Clone, Debug)]
pub struct Row {
    pub pats: Vec<Pat>,
    pub locals: Vec<BindGroup>,
    pub body: RowBody,
    pub span: Span,
}

/// Scrutinees name strict preceding lets; the match records its result type
and failure site.
#[derive(Clone, Debug)]
pub struct Match {
    pub scruts: Vec<Id>,
    pub rows: Vec<Row>,
    pub result: Type,
    pub span: Span,
}

```

Lint independently checks constructor type arguments, field counts and field types. For a constructor pattern it instantiates the declaration under the scrutinee's type arguments, then checks each binder. Guards must be Boolean, and every row must produce the declared match result. Malformed-Core tests exercise these checks without asking the surface inferencer to validate their inputs.

Executing rows

The interpreter evaluates constructor fields left to right. Matching already evaluated values recursively tests constructors and literals, extending a private row environment with variables and as-patterns:

```
fn matches_pattern(p: &core::Pat, v: &Value, env: &mut Env) → bool {
  match p {
    core::Pat::Wild ⇒ true,
    core::Pat::Var(id) ⇒ {
      env.push((id.clone(), v.clone()));
      true
    }
    core::Pat::As(id, p) ⇒ {
      env.push((id.clone(), v.clone()));
      matches_pattern(p, v, env)
    }
    core::Pat::Lit(l) ⇒ matches!(v, Value::Lit(actual) if l==actual),
    core::Pat::Con(id, ps) ⇒ match v {
      Value::Con(actual, vs)
        ⇒ if id == actual && ps.len() == vs.len() ⇒
            {
              ps.iter().zip(vs).all(|(p, v)| matches_pattern(p, v, env))
            }
        ⇒ false,
      _,
    },
  }
}
```

A failed pattern discards that environment and tries the next row. A successful pattern evaluates its where groups once, then tests guards in order. If all fail, execution continues with the next row. Exhausting the rows reports `fun: pattern match failure at FILE:LINE:COL` and exits with status 1.

The value printer uses constructor precedence, so a nested application prints as `Just (Left 3)`. Lists and tuples use their surface notation. A character list prints as `['a', 'b']`, while primitive strings keep double quotes.

Running It

```
$ cd crates/ch11_data
$ cargo run -q -- run tests/programs/data/tree.fun --lint
(True,False)
$ cargo run -q -- dump --pass=syn tests/programs/data/bool-data.fun
nonrec { not#9 = (lambda $match#11 (match $match#11 [ (True) → False ]
[ (False) → True ])); }
nonrec { main#10 = (app (app (,) (app not#9 True)) (app not#9 False)); }
$ cargo test --test data
```

Guard fallthrough is worth experimenting with. Make the first pattern match but its guard fail, then check that the next equation runs. Also try an unused top-level pattern binding that fails. Strict evaluation means “unused” doesn’t make the match disappear. These examples exercise the parts that an ordinary successful tree traversal might miss.

Exercises

1. **Introductory.** define a binary tree mirror function and test it on an asymmetric tree.
2. **Intermediate.** Explain the type errors from a constructor pattern with too few fields and from two equations returning different types.
3. **Intermediate.** Write two guards that fall through to a later equation. Move their shared computation into where and inspect the Syn dump.
4. **Advanced.** Compare strict pattern binding with Haskell’s ordinary pattern binding. Construct a program whose termination or error behavior differs.
5. **Advanced.** Design a lint test that swaps two differently typed constructor fields while preserving the field count.

Further Reading

The Haskell Report specifies datatype declarations, patterns, guards, and local scopes (S. Marlow 2010). Pierce develops the connection between data introductions and eliminations (Pierce 2002). Maranget’s papers explain the later steps: warning analysis and decision-tree compilation (Maranget 2007; 2008).

12

Records

New files

None.

Changed files

- `src/typecheck.rs`
- `src/data.rs`
- `src/lexer.rs`
- `src/elaborate.rs`
- `src/ast.rs`
- `src/core.rs`
- `src/desugar.rs`
- `src/eval.rs`
- `src/fixity.rs`
- `src/format.rs`
- `src/main.rs`
- `src/parse.rs`
- `src/rename.rs`
- `src/renamed.rs`

Motivation

What does the third argument to a constructor mean? With a small point or pair, you can probably remember. With a configuration value containing half a dozen integers, it becomes a small memory test every time you read the code. Let's give those fields names.

Fun's records build on the datatypes we already have. We'll use dot syntax to read a field and require every record type to have just one constructor. This avoids two awkward properties of tradi-

tional Haskell selectors. Access cannot fail because we picked the wrong constructor, and field names don't consume ordinary function names. Two record types can both have an `x` field (S. Marlow 2010; Mitchell and Fletcher, n.d.).

The interesting work is in inference. When we see `p.x`, how much do we know about `p`? We'll arrange for information elsewhere in the expression to help answer that question.

The Language, Now

```
data Point = Point { x :: Int, y :: Int }

move p = p { x = p.x + 1 }

main = move (Point { y = 4, x = 3 })
```

Construction must supply every field exactly once. Its expressions run in written order, even when that differs from declaration order. Positional construction and patterns remain available. A record pattern may omit fields, which become wildcards: `Point { x = n }` binds only the first coordinate.

Projection and update bind more tightly than application. Thus `f p.x` means `f (p.x)`, and `f p { x = 3 }` means `f (p { x = 3 })`. The section `(.x)` is shorthand for a lambda. Chained access such as `shape.origin.x` projects twice. An update preserves the receiver's complete type, including its type arguments.

```
data Label = Label { x :: String }

coordinate :: Point → Int
coordinate p = p.x
```

The signature identifies which `x` is meant. The parenthesized expression annotation `(p :: Point).x` works too. If only one record declares a field, the checker can infer its receiver type without an annotation.

Theory

Two declarations with identical fields still describe different types. That is what *nominal* means here. A `Point` doesn't become some other record merely because both happen to contain `x` and `y`. A structural or row-polymorphic system could instead let a function accept any record with the fields it needs (Pierce 2002). We're choosing the nominal design so field access can resolve to a particular constructor and slot before runtime.

Projection introduces a wanted constraint linking the receiver and result types:

$$\frac{\Gamma \vdash e : \tau \quad \text{Has}(\tau, x, \alpha)}{\Gamma \vdash e.x : \alpha} \text{ (Projection)}$$

`Has` is an internal checking obligation. It is never generalized into a scheme. For a unique field name, the checker immediately unifies the receiver with that record type, using fresh parameters. Otherwise it waits for the receiver’s nominal head to become known. Before generalizing a binding group it repeats resolution to a fixpoint. Any remaining field wanted produces `E0245` and an annotation hint.

GHC’s record-dot proposal is useful prior art for the surface syntax. Its `HasField` mechanism supports a different form of overloaded access. Fun’s field solver always chooses a concrete nominal record before elaboration (Mitchell and Fletcher, n.d.).

Implementation

Classifying the dot

Whitespace is significant for this token. `f . g` means projection, while `f . g` uses the ordinary composition operator. The post-pass compares original token spans before trivia capture and layout can obscure adjacency:

```

/// Classify dot adjacency before trivia capture and layout can erase spacing.
fn projection_dots(
    file: FileId,
    tokens: &mut [Spanned],
    errors: &mut Vec<Diagnostic>,
) {
    for i in 1..tokens.len().saturating_sub(1) {
        if !matches!(&tokens[i].1, Token::Operator(op) if op == ".")
            || tokens[i].2 != tokens[i + 1].0
            || !matches!(tokens[i + 1].1, Token::Ident(_))
        {
            continue;
        }
        if tokens[i - 1].2 != tokens[i].0 {
            continue;
        }
        if tokens[i - 1].1 == Token::Left {
            tokens[i].1 = Token::ProjSection;
            if !matches!(
                tokens[i + 2..].iter().find(|(&t, _)| !matches!(
                    t,
                    Token::LineComment(_) | Token::BlockComment(_)
                )),
                Some((_, Token::Right, _))
            ) {
                errors.push(Diagnostic::error(
                    ErrorCode::ProjectionDot,
                    "a projection section must have the form (.field)",
                    Span::new(file, tokens[i].0, tokens[i].2),
                ));
            }
        } else if matches!(
            tokens[i - 1].1,
            Token::Ident(_)
            | Token::Right
            | Token::BracketClose
            | Token::Close
            | Token::Int(_)
            | Token::Char(_)
            | Token::String(_)
        ) {
            tokens[i].1 = Token::Proj;
        }
    }
}

```

The token preceding a projection must be an identifier, literal, or closing delimiter. A dot immediately following (and preceding a variable identifier is a projection-section token. Other dots remain operators. In particular, the same rule leaves `x .y` as an ordinary operator occurrence.

E0004 diagnoses malformed recognized sections such as `(.x y)`. The formatter keeps dot and label tokens adjacent and moves attached comments outside that sequence. Both decisions are tested because a whitespace-only rewrite could otherwise change the program's meaning.

Keeping field metadata nominal

Each record constructor retains field labels alongside its field types. Datatype validation rejects a record mixed with other constructors (E0240) and repeated declaration labels (E0104). Named construction checks missing, duplicate, and unknown fields with E0241, E0242, and E0243 respectively.

Record patterns are converted to positional constructor patterns during renaming. The renamer looks up each label's slot and fills every omitted slot with a wildcard. Consequently the pattern checker and Core lint need no new record-pattern rule. No field is added to the value environment.

Solving field obligations

A wanted records the receiver type, label, fresh field type, source site, and inference level. Sites distinguish individual fields within an update:

```
/// A field site's position distinguishes updates of several named slots.
pub type FieldSite = (u32, usize);

/// Field constraints never enter a polymorphic scheme.
#[derive(Clone)]
struct FieldWanted {
    record: Type,
    label: String,
    field: Type,
    site: FieldSite,
    span: crate::source::Span,
    level: u32,
}
```

The projection inference branch creates that obligation and returns its fresh result type. Successful resolution later unifies the result with the instantiated field declaration:

```
Kind::Project(e, label) => {
  let record = self.infer(env, e)?;
  let field = self.fresh();
  self.field(
    record,
    label.clone(),
    field.clone(),
    (expr.node, 0),
    expr,
  )?;
  Ok(field)
}
```

`solve_field` either identifies a constructor and field slot, reports a type with no such field (E0244), or leaves an unknown receiver unresolved. The surrounding loop repeats because one resolved field may identify the receiver of another:

```

fn solve_fields(
    &mut self,
    complete: Option<u32>,
) → Result<(), Diagnostic> {
    loop {
        let pending = std::mem::take(&mut self.wanted);
        let count = pending.len();
        for wanted in pending {
            if !self.solve_field(&wanted)? {
                self.wanted.push(wanted);
            }
        }
        if self.wanted.len() == count {
            break;
        }
    }
    if let Some(level) = complete
        && let Some(w) = self.wanted.iter().find(|w| w.level > level)
    {
        let mut error = Diagnostic::error(
            ErrorCode::AmbiguousField,
            format!("ambiguous field access .{}", w.label),
            w.span,
        );
        error.notes.push(
            "add a type annotation to identify the record type".into(),
        );
        return Err(error);
    }
    Ok(())
}

```

Signature-directed checking supplies lambda parameter types before their bodies are visited. This matters when a local binding accesses a shared field. The outer signature must be available before that local group is generalized. The existing rigid signature check still rejects an annotation that claims too much polymorphism.

Elaborating strict construction and access

Field order in source is independent of field order in memory. Construction therefore introduces a strict let for each supplied expression, then builds a constructor using those variables in declaration order:

```

Kind::Record(id, fields) => {
  let declaration =
    self.constructors[id.index as usize].clone();
  let arguments = self.typed.instantiations[&expr.node]
    .iter()
    .map(|t| convert(t, variables))
    .collect::<Vec<_>>();
  let mut values = vec![Expr::Lit(Lit::Int(0)); fields.len()];
  let mut locals = vec![];
  for (label, e) in fields {
    let index = declaration
      .labels
      .as_ref()
      .unwrap()
      .iter()
      .position(|n| n == label)
      .unwrap();
    let parameter = Id {
      name: self.sess.fresh_id("$field"),
      ty: instantiate(
        &declaration.fields[index],
        &declaration.parameters,
        &arguments,
      ),
    };
    locals.push(BindGroup {
      recursive: false,
      binds: vec![Bind {
        name: parameter.clone(),
        rhs: Box::new(self.expression(e, variables)),
      }],
    });
    values[index] = Expr::Var(parameter);
  }
  let mut body = Expr::Con(id.clone(), arguments, values);
  for local in locals.into_iter().rev() {
    body = Expr::Let(local, Box::new(body));
  }
  body
}

```

Projection evaluates its receiver once, matches its sole constructor, and returns the selected field binder. Update uses the same match, evaluates replacement expressions in source order, and rebuilds

the constructor with unchanged fields in their original slots. The result type equals the receiver type, so an update cannot turn `Box Int` into `Box Bool`.

Core sees only constructs introduced in Chapter 11. Independent lint checks their types, and the interpreter executes ordinary matches and constructors. Its value printer consults the constructor’s labels to print `Point {x = 3, y = 4}` in declaration order. Later native projection will become a field load.

Running It

```
$ cd crates/ch12_records
$ cargo run -q -- run tests/programs/records/nested.fun --lint
(3,Outer {inner = P {x = 10, y = 4}})
$ cargo run -q -- dump --pass=types tests/programs/records/shared-
signature.fun
main : Int
$ cargo test --test records
```

Try giving a second record an `x` field and remove the annotation from a projection. The ambiguity error shows why field names alone aren’t always enough. Another useful experiment is to put different errors in a receiver and an updated field. The first error should follow source evaluation order, even though the final fields are stored in declaration order.

Exercises

1. **Introductory.** define a polymorphic `Box` a record and infer the type of its projection section.
2. **Intermediate.** Add another record with the same field label, observe the ambiguity error, and resolve it with a signature.
3. **Intermediate.** Explain why another uniquely named field can disambiguate an earlier unresolved access in the same binding group.
4. **Advanced.** Write a construction whose source field order differs from its declaration order and whose first failure reveals the evaluation order.
5. **Advanced.** Identify the parts of inference and generalization that would need to change to support row-polymorphic access. Do not add that feature here.

Further Reading

The Haskell Report describes traditional record construction, update, and selector functions (S. Marlow 2010). GHC proposal 282 motivates record-dot syntax and its relationship to overloaded field access (Mitchell and Fletcher, n.d.). Pierce develops structural record typing, a useful comparison with Fun’s nominal choice (Pierce 2002).

13

Kinds and Type Constructors

New files

- `src/kinds.rs`

Changed files

- `src/types.rs`
- `src/typecheck.rs`
- `src/core_lint.rs`
- `src/ast.rs`
- `src/core.rs`
- `src/core_type.rs`
- `src/data.rs`
- `src/elaborate.rs`
- `src/lexer.rs`
- `src/lib.rs`
- `src/main.rs`
- `src/parse.rs`
- `src/rename.rs`

Motivation

`Int` describes a value. `Maybe Int` does too. But `Maybe` on its own is waiting for another type before it can describe a value. We already handle this by counting a datatype's arguments. What happens when one of those arguments is itself something like `Maybe`?

Consider a field with type $f\ a$. Here f must accept a type, while a fills that argument. We need a way to describe their different jobs. These descriptions are called *kinds*. They look reassuringly familiar, using arrows and $*$ instead of the richer collection of ordinary types.

Much of the machinery we need is already familiar too. We'll infer kinds by solving equations, just as we inferred types. It's a nice example of a compiler idea earning its keep a second time.

The Language, Now

```
data Wrap f a = Wrap (f a)

unwrap :: Wrap Maybe Int → Maybe Int
unwrap (Wrap x) = x

main = unwrap (Wrap (Just 42))
```

Wrap has kind $(* \rightarrow *) \rightarrow * \rightarrow *$. A recursive declaration such as `data Fix f = Fix (f (Fix f))` gives Fix kind $(* \rightarrow *) \rightarrow *$. Declaration dependencies are grouped into strongly connected components. Each component is inferred before its unconstrained kind variables default to $*$.

```
type Pair a = (a,a)

type Count = Int

addPair :: Pair Count → Count
addPair (x,y) = x + y

main = addPair (20,22)
```

A synonym introduces no constructor or runtime representation. It must be fully applied, even where a partially applied datatype would have the right kind. Cycles are rejected, including unused cycles. E0301 reports a kind mismatch, E0302 a partial synonym, and E0303 a synonym cycle. The type constructors `[]`, `(,)`, `(,,)`, and `(→)` may appear unapplied where their kinds permit.

Theory

Read `Maybe :: * → *` much like a function type. It accepts one type of values and produces another. Applying it to `Int :: *` gives `Maybe Int :: *`. The kind application rule writes down that same step.

$$\frac{\Gamma \vdash f : \kappa \rightarrow \sigma \quad \Gamma \vdash a : \kappa}{\Gamma \vdash f\ a : \sigma} \text{ (Application)}$$

Function arguments and results must each have kind $*$. A constructor field must also have kind $*$: a value cannot inhabit the unapplied type constructor `Maybe`. The distinction between kinds and types follows the presentation of higher-order type operators in Pierce (Pierce 2002). `Fun` infers kinds but does not add type-level lambdas or explicit kind annotations.

Synonym substitution follows the Haskell report's fully applied convention (S. Marlow 2010). Expanding arguments before entering a synonym permits `Id (Id Int)` without mistaking the repeated name for a declaration cycle. A stack of definitions detects actual recursion during expansion.

Implementation

Kinds classify type constructors in the same way that types classify terms. The implementation combines kind unification, an occurs check, and dependency ordering to infer those classifications.

```
pub enum Kind {  
    Star,  
    Arrow(Box<Kind>, Box<Kind>),  
    Meta(u32),  
}
```

`Store::unify` follows solutions, compares arrow structure, and rejects a metavariable occurring inside its own proposed solution. `finish` defaults unsolved cells only after all fields in a declaration component were checked.

```

pub fn unify(
    &mut self,
    a: &Kind,
    b: &Kind,
    span: Span,
) → Result<(), Diagnostic> {
    let (a, b) = (self.zonk(a), self.zonk(b));
    match (&a, &b) {
        _ if a == b ⇒ Ok(()),
        (Kind::Meta(n), k) | (k, Kind::Meta(n))
            if !self.occurs(*n, k) ⇒
        {
            self.solutions[*n as usize] = Some(k.clone());
            Ok(())
        }
        (Kind::Arrow(a, b), Kind::Arrow(c, d)) ⇒ {
            self.unify(a, c, span)?;
            self.unify(b, d, span)
        }
        _ ⇒ Err(Diagnostic::error(
            ErrorCode::Kind,
            format!("cannot unify kind {a} with {b}"),
            span,
        )),
    }
}

```

The declaration pass first installs every head, then reuses Chapter 9’s Tarjan implementation. Signatures infer implicit variable kinds against that closed table. The immutable datatype environment validates synonym use and converts the expanded syntax to inference types.

The type store now records a kind on each metavariable. Instantiating a scheme copies its quantified kinds as well as freshening its variables. Unification of `f a` with `Maybe Int` separates the final application from each side, solving `f = Maybe` and `a = Int` without requiring either to be a value type.

Core keeps a kind table keyed by unique type binder identities. Lint independently checks applications, constructor arguments, and function operands against it. Internal type variables absent from every enclosing scheme receive a closed witness of the right kind. These compiler-only type constants have no values or source syntax and disappear at erasure. This extends the `Int` witness used in Chapter 10 to kinds for which `Int` is unsuitable.

Running It

```
$ cd crates/ch13_kinds
$ cargo run -q -- run tests/programs/kinds/wrapper.fun --lint
Just 42
$ cargo run -q -- dump --pass=types tests/programs/kinds/wrapper.fun
main : Maybe Int
$ cargo run -q -- check tests/programs/kinds/synonym-self.fun
E0303
```

The last command fails and points to the bad kind application. Try a partially applied datatype constructor next, then a partially applied synonym. The former can be useful at a higher kind, while the latter is rejected by our synonym rules. Having the same apparent number of missing arguments doesn't make the two declarations interchangeable.

Exercises

1. **Warm-up.** Infer the kind of `data Compose f g a = Compose (f (g a))`.
2. **Intermediate.** Explain why `data Bad f = Bad (f f)` fails the occurs check.
3. **Intermediate.** Trace kind inference through mutually dependent declarations.
4. **Advanced.** Find a term whose elaboration needs an internal witness of kind $* \rightarrow *$, and inspect the corresponding Core application.

Further Reading

- Pierce's treatment of type operators (Pierce 2002).
- The Haskell report's datatype and synonym rules (S. Marlow 2010).
- Higher-order type operators and System F-omega (Pierce 2002).

14

Type Classes

New files

- `src/classes.rs`
- `src/evidence.rs`
- `src/elaborate/dictionaries.rs`
- `src/primitives.rs`

Changed files

- `src/typecheck.rs`
- `src/elaborate.rs`
- `src/types.rs`
- `src/ast.rs`
- `src/core.rs`
- `src/core_lint.rs`
- `src/core_type.rs`
- `src/data.rs`
- `src/desugar.rs`
- `src/eval.rs`
- `src/fixity.rs`
- `src/kinds.rs`
- `src/lexer.rs`
- `src/lib.rs`
- `src/main.rs`
- `src/parse.rs`
- `src/rename.rs`

- `src/renamed.rs`
- `src/syn.rs`

Motivation

How do we compare two lists? We compare their elements, which means we need to know how to compare those elements. We could pass an equality function into every list comparison. That works, but we'd soon spend a lot of our time passing around the same handful of operations.

Type classes let the compiler do that plumbing. We describe the operations in a class and provide implementations in instances. The compiler works out which implementation a call needs and passes it along.

The lovely part is that we don't need a mysterious new runtime feature. A class becomes a record of functions, usually called a *dictionary*. Once we've made those arguments explicit, our existing Core interpreter can run the result. In this chapter we'll follow that translation all the way through.

The Language, Now

Let's define a class and give it an instance for integers.

```
class Value a where
  value :: a → Int

instance Value Int where
  value x = x + 1

main = value 41
```

Class parameters may have higher kinds. A method `unwrap :: f a → a` gives its class parameter `f` kind `* → *`. The remaining variable `a` is independently polymorphic. Classes may have superclasses and default method bodies. Instance methods accept ordinary equations, guards, and local where bindings.

The Prelude now defines `Eq`, `Ord`, `Show`, `Num`, `Fractional`, `Semigroup`, and `Monoid`. Arithmetic and comparisons resolve to methods. Literals remain monomorphic. `1` has type `Int`, while `1.0` has type `Double`. Consequently `1 + 2.0` fails. `toDouble 1 + 2.0` succeeds. There is no defaulting.

```
same :: Ord a ⇒ a → a → Bool
same x y = x == y

main = same [1,2] [1,2]
```

`Ord` provides an `Eq` superclass, so the signature supplies enough evidence. An absent instance reports E0210. Malformed or overlapping instances and missing methods report E0211. A constraint whose variable is absent from the result type reports E0230: for example, `show Nothing` cannot choose the element's `Show` instance. An explicit `Maybe Int` annotation resolves that ambiguity.

`Double` adds floating arithmetic, comparisons, conversions, and mathematical functions. Primitive names remain private to the embedded Prelude. The source still has no IO, so `run` prints the value of `main`. `++` still appends strings in this chapter. `List ++` belongs to the complete Prelude chapter. `<` already supports both strings and lists.

Theory

The type `Eq a ⇒ a → a → Bool` says we can compare two values of `a` if someone supplies equality for `a`. The part before `⇒` is the *context*. When we use a function with this type, inference creates a request for that equality operation. We call this a *wanted* constraint.

Some requests have immediate answers, such as the instance for `Int`. Others remain requests about a type variable. When we generalize a binding, those can become part of its scheme so callers supply the answer. Constraints involving an outer variable instead belong to the enclosing scope. Jones gives an executable account of these operations (Jones 1999).

Finding an instance is only half the job. We also need something to pass at runtime. That is the *evidence* for a constraint. An `Ord a` dictionary already contains its `Eq a` superclass dictionary. Given `Eq a`, the list instance can build a dictionary for `Eq [a]`. Following these steps constructs an ordinary expression, made from dictionary variables, field selections, and instance applications. There is no runtime search left to perform.

Signatures are checked with rigid skolems and their context as givens. Signed bindings may be used polymorphically inside their own bodies. Following the Haskell report (S. Marlow 2010), type-inference dependencies omit edges to signed bindings. Runtime dependencies retain those edges, since recursive functions still need a shared recursive environment. These are two different uses of the same source dependency graph.

Generalization remains unrestricted. A constrained value such as `comparison = (==)` elaborates to a function taking a dictionary. Its body is evaluated when that dictionary is supplied, so source-level sharing is lost. `Fun` accepts this consequence of dictionary passing. Purity makes it sound.

Implementation

`classes.rs` implements superclass search, instance matching, and entailment. The main inferencer continues to use mutable, level-based metavariables. The small immutable map used to match an instance head is local to that lookup. It is not an alternative substitution-based type checker.

```
pub enum Evidence {
    Given(usize, Vec<usize>),
    Instance(usize, Vec<Type>, Vec<Evidence>),
}
```

A given stores its position in the lexical dictionary environment and a path through superclass fields. An instance proof stores the selected instance, its type arguments, and proofs of its prerequisites.

```
pub fn resolve(
    &self,
    goal: &Predicate,
    givens: &[Predicate],
    span: Span,
) → Result<Evidence, Diagnostic> {
    for (i, given) in givens.iter().enumerate() {
        if given.ty == goal.ty
            && let Some(path) =
                self.super_path(&given.class, &goal.class)
        {
            return Ok(Evidence::Given(i, path));
        }
    }
    let (i, args, context) =
        self.by_instance(goal).ok_or_else(|| {
            error(
                ErrorCode::NoInstance,
                format!("no instance for {} {}", goal.class, goal.ty),
                span,
            )
        })?;
    let proofs = context
        .iter()
        .map(|p| self.resolve(p, givens, span))
        .collect::<Result<Vec<_>, _>>()?;
    Ok(Evidence::Instance(i, args, proofs))
}
```

Class validation checks superclass cycles, method membership, extra method constraints, distinct variables in nominal instance heads, overlap, and superclass availability. Kind inference processes superclass dependencies before defaulting unconstrained class kinds. Instance contexts may mention only the head's variables. Solving therefore descends into structurally smaller types.

Field constraints still run before class constraints at generalization. After zonking, `evidence.rs` records the class proofs at variable occurrences while their lexical givens are available. Elaboration consumes these checked proofs.

```
pub struct EvVar {
    pub node: u32,
    pub index: usize,
}
```

`elaborate/dictionaries.rs` declares one nominal Core dictionary type per class. Its fields contain superclass dictionaries followed by methods. Method-local quantifiers become System F polymorphism inside the field type. This does not add rank-N signatures to the source language. Each instance becomes a global value, abstracted over its type variables and context dictionaries. Method selectors are ordinary matches on the dictionary constructor.

```
pub(super) fn dictionary_type(&self, class: &str, ty: Type) → Type {
    let c = &self.constructors[self.dicts[class]];
    instantiate(&c.result, &c.parameters, &[ty])
}
```

Defaults get named globals, and each missing implementation is checked from a copy of its default body specialized to the instance. This keeps strict dictionary construction from tying a recursive value knot. Dictionary-introduced dependencies are ordered before evaluation. Recursive non-function initialization is rejected. The evaluator installs function closures before evaluating values, and each unconstrained top-level value is still evaluated once.

`primitives.rs` supplies fixed primitive signatures and implementations. Integer operations retain 63-bit wrapping. Floating display uses shortest round-trip digits with the specified positional range, signed exponents, and explicit special values. Tests cover negative zero, boundary magnitudes, Unicode strings, and failed conversions. The shared runtime chapter will move the non-trivial operations into `fun-rt`.

Running It

```
$ cd crates/ch14_classes
$ cargo run -q -- run tests/programs/classes/minimal.fun --lint
42
$ cargo run -q -- dump --pass=types tests/programs/classes/principal-type.fun
main : Num a => a -> a -> a
$ cargo run -q -- check tests/programs/classes/mixed-literals.fun
E0201
```

The failing command also prints a labelled diagnostic. `--dump=core` exposes `$D` dictionary types, `$f` instance globals, and explicit dictionary applications. The small dictionary snapshots disable the Prelude to keep those examples readable. Ordinary builds compile the embedded Prelude before the user's scope. Shadowing a Prelude name cannot change the Prelude's own references.

Expression annotations also work without enclosing parentheses. There is one place to be careful. Inside a case guard, parenthesize an annotation so its type arrow can't be confused with the arrow leading to the alternative's body.

Exercises

1. **Warm-up.** Inspect the dictionary arguments for equality on `[Maybe Int]`.
2. **Intermediate.** Write an instance whose context is insufficient for its superclass, then identify the missing proof.
3. **Intermediate.** Explain why `show Nothing` is ambiguous but `show (Nothing :: Maybe Int)` is not.
4. **Advanced.** Compare the type-inference and runtime dependency components of the polymorphic-recursion fixture.
5. **Advanced.** Corrupt a superclass dictionary's Core type and add a lint regression that does not invoke surface inference.

Further Reading

- Typing Haskell in Haskell (Jones 1999).
- Constructor classes and higher-kinded parameters (Jones 1993).
- Classes and instances in the language reference (S. Marlow 2010).
- The Haskell report's binding groups and instance restrictions (S. Marlow 2010).

15

Monads, Do Notation and IO

New files

- `src/io_lower.rs`

Changed files

- `src/desugar.rs`
- `src/eval.rs`
- `src/typecheck.rs`
- `src/ast.rs`
- `src/data.rs`
- `src/elaborate.rs`
- `src/fixity.rs`
- `src/kinds.rs`
- `src/layout.rs`
- `src/lexer.rs`
- `src/lib.rs`
- `src/main.rs`
- `src/primitives.rs`
- `src/rename.rs`
- `src/session.rs`

Motivation

So far, our programs have had a rather limited relationship with the outside world. The driver evaluates `main`, prints the answer, and goes home. Let's write programs that can ask a question, read a response, and decide what to do next.

We want to keep ordinary expressions pure, so evaluating an IO value won't perform its effects. We'll represent an action as a value and run it at an explicit boundary. That distinction lets us build and combine actions using the functions we already know.

The type classes from Chapter 14 give us a way to describe these combinations. Functor, Applicative, and Monad work with IO, but they also describe useful operations on Maybe, Either, and lists. We'll start with what the operations do and expand the syntax by hand. There's plenty to understand here without adding mystique to the vocabulary.

The Language, Now

A complete program now has `main :: IO ()`, whether that signature is written or inferred. The smallest example calls the new Prelude printer:

```
main = putStrLn "hello"
```

do notation also works for ordinary data. Here the two binds extract values from Just, and pure constructs the result in the same Maybe instance:

```
main = print (do { x ← Just 20; y ← Just 22; pure (x+y) })
```

A block must end with an expression. A statement `x ← action` binds `x` only in the remaining statements. A statement `let` introduces local declarations without an `in`. Patterns may be nested, including tuples, lists, as-patterns, and nominal record patterns. A failed refutable pattern calls error with its source location. Fun has no MonadFail, so failure is a runtime error even when the surrounding monad is Maybe.

`putStr`, `putStrLn`, `getLine`, `print`, and `exitWith` provide the program's external interface. `mapM`, `mapM_`, `sequence_`, `when`, and `unless` express common sequencing patterns. A missing or wrongly typed entry point reports E0250. A body that violates its explicit signature can instead fail during signature checking with E0203.

Theory

Suppose an action produces an `a`, and the rest of our program knows how to use that `a` to make the next action. Bind connects the two. Its second argument is a function because the next action may depend on the first result. Read this type slowly from left to right (Jones 1993).

$$\text{Monad } m \Rightarrow m \alpha \rightarrow (\alpha \rightarrow m \beta) \rightarrow m \beta$$

For IO, the source checker treats `IO` as an abstract nominal constructor. Core represents an action as a function from unit to its result:

$$IO\ \alpha \Rightarrow () \rightarrow \alpha$$

The unit argument gives us somewhere to put the delay. Constructing the function doesn't run its body. `returnIO` builds a function that returns its saved value, while `bindIO` builds one that runs the first action and passes the result to the next. The effectful primitives sit inside those bodies.

Only the trusted Prelude and compiler can use this representation. Source programs see abstract `IO a` values, so they can't apply an action to `()` and run it wherever they please. That boundary is what lets ordinary Fun expressions remain pure.

Strictness still applies. Constructing `putStrLn "hello"` evaluates its string argument and returns an action. It does not print. Constructing an action with `error "bad"` as an eagerly demanded argument can fail before the action runs. Sequencing controls effects, not evaluation of arbitrary pure arguments. The Haskell report describes the analogous `do` translations (S. Marlow 2010), but its lazy evaluation must not be imported into Fun's operational account.

The monad laws describe intended instance behavior. The compiler checks types, not those equations. In particular, equations involving nontermination need care in a strict language. Dictionary passing remains the mechanism from Chapter 14: no new class dispatch appears in the evaluator.

Implementation

The parser calls `desugar :: do_block` before lexical renaming. Processing the statements backwards makes the remaining block the continuation of each `bind`. A variable pattern becomes a lambda directly. A refutable pattern introduces a case with a failure alternative. Its temporary name cannot be written as a source identifier.

```
/// Expand do syntax into ordinary applications, matches, and local bindings.
pub fn do_block(
  statements: Vec<Statement>,
  span: Span,
) → Result<crate::ast::Expr, Diagnostic> {
  use crate::ast::{
    Expr as E, Kind as K, Pat as P, PatKind as PK, Row as R,
    RowBody as B,
  };
  let node = |kind| E { kind, span };
  let app = |a, b| node(K::App(Box::new(a), Box::new(b)));
  let mut statements = statements;
  let Some(Statement::Expr(mut body)) = statements.pop() else {
    return Err(Diagnostic::error(
      ErrorCode::Parse,
      "do must end with an expression",
    ));
  };
  app(body, node(K::Block(statements)))
}
```

```

        span,
    ));
};
for statement in statements.into_iter().rev() {
    body = match statement {
        Statement::Expr(e) => {
            app(app(node(K::Var(">>".into()))), e), body)
        }
        Statement::Let(bs) => node(K::LetGroup(bs, Box::new(body))),
        Statement::Bind(p, e) => {
            let continuation = match &p.kind {
                PK::Var(n) => node(K::Lam(n.clone(), Box::new(body))),
                PK::Wild => node(K::Lam("$do".into(), Box::new(body))),
                _ => {
                    let site = p.span;
                    let rows = vec![
                        R {
                            patterns: vec![p],
                            body: B::Plain(Box::new(body)),
                            locals: vec![],
                            span: site,
                        },
                        R {
                            patterns: vec![P {
                                kind: PK::Wild,
                                span: site,
                            }],
                            body: B::Plain(Box::new(E {
                                kind: K::DoFail,
                                span: site,
                            })),
                            locals: vec![],
                            span: site,
                        },
                    ];
                    node(K::Lam(
                        "$do".into(),
                        Box::new(node(K::Case(
                            Box::new(node(K::Var("$do".into()))),
                            rows,
                        ))),
                    ))
                }
            };
            node(K::Lam(
                "$do".into(),
                Box::new(node(K::Case(
                    Box::new(node(K::Var("$do".into()))),
                    rows,
                ))),
            ))
        }
    }
}

```

```

        };
        app(app(node(K::Var("»=">".into()), e), continuation)
    }
};
}
Ok(body)
}

```

This placement differs from a separate post-renaming desugaring pass: generated bindings go through the same scope checker as handwritten ones. The helper `do_pattern` validates the pattern-shaped syntax on the left of `←`. The span-bearing `DoFail` node is later translated into an ordinary error call with its file, line, and column. No `do` node reaches `Core`.

Only the Prelude's trusted `FileId` lets the checker relate its IO implementation to unit-taking functions. User-visible schemes remain nominal. After elaboration, `io_lower::lower` traverses all `Core` types before `lint`: binder types, type applications, constructor fields, match result types, and dictionary types. Replacing only `IO a` at the outermost level would miss higher-kinded uses such as a `Monad IO` dictionary. The traversal also replaces the unapplied constructor `IO` with the partially applied function constructor `(→) ()`.

The runner first installs function closures and evaluates strict top-level values in dependency order, then applies the entry point to unit:

```

/// Initialize globals, then run the checked IO action exactly once.
pub fn run_io(program: &core::Program, fuel: usize) → Result<(), Failure> {
    execute(program, fuel, true).map(|_| ())
}

```

An unused top-level action is still a value initialized at startup. Its body is not entered, as this regression demonstrates:

```

unused = putStrLn "bad"

main = putStrLn "ok"

```

The evaluator distinguishes a language error from a requested process exit. This distinction prevents `exitWith 7` from being printed as an error message:

```

/// Language termination is separate from an ordinary runtime failure.
#[derive(Debug, Clone, PartialEq, Eq)]
pub enum Failure {
    Runtime(String),
    Exit(i32),
}

```

The driver flushes stdout at completion and before runtime diagnostics. Input primitives flush before reading, so a prompt reaches the user before the program blocks. The test runner accepts .in files and sends identical input to each engine. A pipe-handshake regression checks flushing independently of terminal buffering.

Existing programs need a small adjustment here. The driver no longer prints main for us, so we put the printing in the program itself. If a value has no Show instance, we'll need to supply one or choose something printable from it. The old value printer remains available through --print-value for inspecting compiler fragments.

Running It

From the repository root, enter the chapter crate. These commands print an explicit action's output and its inferred entry-point type:

```

$ cd crates/ch15_io
$ cargo run -q -- run tests/programs/io/hello.fun --lint
hello
$ cargo run -q -- dump --pass=types tests/programs/io/hello.fun
main : IO ()
$ cargo run -q -- run tests/programs/io/deferred.fun --lint
ok

```

Use --pass=core to find the dictionary arguments and the lambda taking unit. It's worth comparing an unused print action with one that is actually sequenced by main. Both are values, but only the latter's body runs. The Core interpreter is still our execution engine in this chapter.

Exercises

1. **Warm-up.** Replace the second Just in the example with a suitably annotated Nothing. Explain why its continuation is not called.
2. **Warm-up.** Compare constructing an unused print action with constructing one whose string argument calls error.
3. **Intermediate.** Trace the generated scopes for a do block containing a let, a tuple bind, and a shadowing bind.

4. **Intermediate.** Inspect a `Monad IO` dictionary in `Core` and identify every type affected by IO lowering.
5. **Advanced.** Extend the pipe-handshake test to issue two prompts and verify the order without relying on a terminal's line buffering.

Further Reading

- The report's `do` expressions and monadic library (S. Marlow 2010).
- Higher-kinded constructor classes (Jones 1993).
- Dictionary passing and qualified types (Jones 1999).

16

Pattern-Match Compilation

New files

- `src/match_compile.rs`
- `src/usefulness.rs`
- `src/env.rs`

Changed files

- `src/core.rs`
- `src/core_lint.rs`
- `src/eval.rs`
- `src/io_lower.rs`
- `src/lib.rs`
- `src/main.rs`

Motivation

Our interpreter tries patterns from top to bottom, which makes their meaning easy to follow. A native compiler needs to turn that process into instructions that inspect tags, load fields, and branch. We could translate each pattern separately, but we'd often inspect the same value again and again.

Instead, we'll look at the patterns together. Put them in a matrix, choose a useful test, and divide the remaining work according to its result. This gives us a decision tree. Join points let different paths share the code they reach, including the next clause when a guard fails.

We can also ask whether any inputs slip through the patterns, or whether an earlier clause has already caught everything a later clause could match. It's a pleasing connection. Understanding the shapes of values helps us both run patterns and explain mistakes in them.

The Language, Now

There is no new source syntax. Earlier programs keep the same results, including guards that fail after their patterns match:

```
f (Just x) | x < 0 = 0
f (Just x) = x
f Nothing = 99

main = print (f (Just 42))
```

The first clause recognizes `Just 42`, but its guard fails. Execution must try the second clause and return 42. It must not report pattern failure or skip to the `Nothing` clause. Local where bindings remain in scope for the guards and right-hand side of their own equation.

The checker can now show us holes in a match. `W0001` includes example inputs that aren't covered, and `W0002` points out a clause that earlier clauses have made unreachable. These are warnings, so the program can still run. Encountering an uncovered input at runtime still reports `fun: pattern match failure at FILE:LINE:COL`.

Theory

Think of a match as a matrix whose rows are clauses and whose columns are scrutinees. Specializing a column for constructor `C` keeps rows that can match `C`, removes that constructor, and inserts its fields as new columns. A default matrix retains rows whose selected pattern can match any other value. This is the decision-tree approach described by Maranget (Maranget 2008).

Fun chooses the first column containing a refutable pattern in the first row. That simple heuristic is deterministic and easy to trace. It is not a claim to minimize the number of tests for every matrix. Wildcards, variables, and as-patterns do not inspect a value. Their bindings are recorded for the eventual clause body. Constructor patterns expose fields only after their tag has matched.

Finite datatypes have known constructor families. Integer and character literals need a default path because a finite set of tested literals cannot exhaust their value domain. String patterns become equality tests. Double patterns also use primitive equality, preserving the language's floating comparison behavior, including equality of positive and negative zero.

Usefulness asks whether a candidate pattern vector admits a value not matched by preceding covering rows (Maranget 2007). A useless source row is redundant. A useful all-wildcard candidate supplies a missing-case witness. Guards complicate coverage because a matching pattern may still fall through. The implementation treats a guarded row as covering only when one guard is literally `True`. It does not attempt to prove arbitrary Boolean expressions.

One Question, Two Checks

There's a neat way to turn both warnings into the same question. Does this pattern match anything the earlier patterns missed? Maranget calls such a pattern *useful*. We'll write $U(P, q)$ for this question, with earlier rows P and candidate pattern vector q . A witness is an actual shape that slips past P and matches q (Maranget 2007).

For reachability, P contains the covering rows before the current clause and q is that clause's pattern vector. For exhaustiveness, P contains all covering rows and q is an all-wildcard vector. The first query asks whether a clause adds any cases. The second asks whether anything is still missing. It's the rows together that determine coverage. A clause can be unreachable even when no single earlier clause covers it completely.

Consider a function with two arguments, `Maybe Bool` and `Bool`:

```
choose Nothing    _      = 0
choose (Just True) _      = 1
choose (Just _)   False = 2
```

Ignore the right-hand sides and retain the argument columns:

Row	First Argument	Second Argument
1	Nothing	_
2	Just True	_
3	Just _	False

Ask whether $(_, _)$ is useful. The first column contains both constructors of `Maybe`, so examine each constructor family member. Specializing for `Nothing` discards rows 2 and 3 and leaves a wildcard for the second argument. That branch is covered. Specializing for `Just` discards row 1 and replaces the first column with the constructor's single field:

Row	Value Inside Just	Second Argument
2	True	_
3	_	False

The `True` branch is covered by row 2. In the `False` branch, only row 3 remains, and it covers a second argument of `False`. The uncovered second argument is `True`. Rebuilding the constructor around the field gives the witness `(Just False, True)` for the original argument vector. Adding `choose (Just False) True = 3` completes this match. Any subsequent clause is unreachable, even though no original clause alone covered every input.

Specialization and the Default Matrix

`usefulness.rs` implements the search on the first column. The selected pattern and constructor family determine the next problem:

1. **Constructor candidate.** For $C(p_1, \dots, p_k)$, keep rows headed by C or a wildcard. Replace C by its fields. Expand a wildcard to k wildcards. Drop rows headed by other constructors, and make the same expansion in the query.
2. **Wildcard candidate, complete family.** If every constructor of the column's datatype appears, specialize for each. Any useful branch supplies a witness.
3. **Wildcard candidate, incomplete family.** Search the default matrix: retain only wildcard-headed rows and drop their first column. A useful suffix can be prefixed by a missing constructor. Literal domains are handled through this path, choosing a literal absent from the tested heads.

The zero-column case distinguishes an empty matrix from an empty row. With no rows, nothing covers the remaining empty vector, so it is useful. With an empty row, an earlier clause already matches everything left, so it is not useful. These small base cases let the search handle nested constructors without enumerating all values of a recursive datatype.

On return, the algorithm rebuilds each stripped constructor around the witness fields. `Fun`'s `useful` therefore returns an optional pattern vector, rather than just a Boolean. The warning driver can print a concrete missing shape, add that shape to its temporary coverage matrix, and search again for up to three examples.

Guards and the Limits of Coverage

A guarded clause can be unreachable because earlier clauses already cover its patterns. It can't usually establish coverage itself, since its guard might fail. `Fun` checks the row against earlier coverage, then adds it only if it is unguarded or has a syntactically literal `True` guard. It does not solve arbitrary guard expressions. Consequently a reported gap can reflect this conservative policy, and a useful pattern does not prove that its guard will ever succeed.

The analysis uses typed constructor families and treats variables as wildcards. As-pattern names do not change coverage. It does not prove that recursive field types are inhabited. Compilation and checking remain separate. The decision tree must preserve clause order and guard fallthrough regardless of these warnings.

Sharing the Decision Tree

A join point names a continuation local to an expression. Its parameters carry the values needed by the shared body, and every jump is a fully applied tail transfer. Unlike an ordinary function, a join cannot escape or be passed as a value. This distinction anticipates the later use of joins in ANF (Maurer et al. 2017).

Implementation

The compiler's internal row separates the remaining patterns from the eventual right-hand side. `values` records the values bound along the current decision path. `target` and `params` identify the shared clause body.

```
#[derive(Clone)]
struct Row {
    pats: Vec<Pat>,
    values: BTreeMap<u32, Id>,
    target: names::Id,
    params: Vec<Id>,
}
```

Before building tests, `Compiler::compile` assigns each clause a parameterized right-hand-side join. It also compiles the residual matrix following that clause into a failure join. Guards run in order. If all fail, a jump enters that residual matrix. Several leaves can therefore reach one clause body without copying it, and several failed guards can use one fallback.

A constructor branch must bind fields at the scrutinee's actual type arguments. The helper looks up the declaration and substitutes its parameters:

```
fn fields(&self, con: &crate::data::ConId, ty: &Type) → Vec<Type> {
    let c = &self.constructors[con.index as usize];
    let args = match ty {
        Type::Con(_, args) ⇒ args.as_slice(),
        Type::Bool ⇒ &[],
        _ ⇒ unreachable!("typed data scrutinee"),
    };
    c.fields
        .iter()
        .map(|t| crate::elaborate::instantiate(t, &c.parameters, args))
        .collect()
}
```

For example, specializing `Just at Maybe Int` binds an `Int`, not the declaration's uninstantiated type variable. The same rule matters for a record projection: its single-constructor match now becomes a tag branch and ordinary field binders. Core lint independently checks these instantiated types.

`Compiler::tree` handles the matrix recursively. An empty matrix becomes `PatternFail`. A first row with no refutable patterns jumps to its clause join, passing the values accumulated for its binders. Otherwise the selected column produces a flat `Case`, or equality tests for string and `Double` literals, with specialized subtrees and a default subtree.

The warning pass uses the original nested patterns, before that representation is lost. Its specialization helper is the matrix operation traced above:

```
fn specialize(
    matrix: &[Vec<Pat>],
    head: &Pat,
    arity: usize,
) → Vec<Vec<Pat>> {
    matrix
        .iter()
        .filter_map(|row| {
            let mut fields = if wild(&row[0]) {
                vec![Pat::Wild; arity]
            } else if same(&row[0], head) {
                match peel(&row[0]) {
                    Pat::Con(_, ps) ⇒ ps.clone(),
                    _ ⇒ vec![],
                }
            } else {
                return None;
            };
            fields.extend_from_slice(&row[1..]);
            Some(fields)
        })
        .collect()
}
```

useful carries the candidate vector and column types alongside the matrix. Its driver repeatedly asks for useful witnesses and records diagnostics:

```
/// Warnings do not alter the decision tree; unmatched execution still fails
at runtime.
pub fn warn(
    sess: &mut Session,
    m: &Match,
    cons: &[ConDecl],
    truths: &std::collections::BTreeSet<u32>,
) {
    let types = m.scruts.iter().map(|i| i.ty.clone()).collect::<Vec<_>>();
    let mut matrix = vec![];
    for (index, row) in m.rows.iter().enumerate() {
        if useful(&matrix, &row.pats, &types, cons).is_none()
```

```

        && !(index > 0 && m.rows[index - 1].span == row.span)
    {
        let mut d = Diagnostic::error(
            ErrorCode::Redundant,
            "clause can never match",
            row.span,
        );
        d.severity = Severity::Warning;
        sess.emit(d);
    }
    if covers(&row.body, truths) {
        matrix.push(row.pats.clone())
    }
}
let mut missing = vec![];
for _ in 0..3 {
    let Some(p) =
        useful(&matrix, &vec![Pat::Wild; types.len()], &types, cons)
    else {
        break;
    };
    missing.push(
        p.iter()
            .map(ToString::to_string)
            .collect::<Vec<_>>()
            .join(" "),
    );
    matrix.push(p)
}
if !missing.is_empty() {
    let mut d = Diagnostic::error(
        ErrorCode::NonExhaustive,
        "non-exhaustive pattern match",
        m.span,
    );
    d.severity = Severity::Warning;
    d.notes.push(format!("missing: {}", missing.join("; ")));
    sess.emit(d)
}
}

```

Generated do-pattern fallback rows share the originating span. The diagnostic pass suppresses redundant warnings for those synthetic adjacent rows, while retaining warnings for source clauses.

Warning analysis does not choose the runtime branches. A mistake in a warning must not silently change evaluation.

Core lint now checks flat alternative shape, constructor field types, globally unique binders, and join argument types and arities. A jump must be in tail position relative to its join. Entering a non-tail subexpression clears the available outer joins. The subexpression may establish its own local joins. The Core interpreter keeps evaluation and return frames in an explicit vector. A source call changes the current expression and environment rather than calling Rust recursively. A jump unwinds frames to its lexical join and transfers control to that body. Closures borrow code through stable indices, so calling a function does not clone its syntax tree. Constructor and environment destruction also use a worklist, preventing a long list from overflowing the stack when released. Tests run non-tail recursion and list reclamation on a 256 KiB host stack.

Saved environments share an immutable balanced tree. Capturing a scope copies one Rc, while adding a binding copies only the search path. `env.rs` implements the update with a work vector. This keeps a growing scope from being copied in full whenever the interpreter saves a continuation. Recursive closures refer to a machine-owned group by index, so calls reuse the group without creating a reference-count cycle.

Running It

```
$ cd crates/ch16_match
$ cargo run -q -- run tests/programs/match/guard-fallback.fun --lint
42
$ cargo run -q -- dump --pass=types tests/programs/match/guard-fallback.fun
main : IO ()
```

For the transformation itself, dump the same file with `--pass=core` and then `--pass=match`. The former retains nested `Match`. The latter exposes flat `Case`, right-hand-side joins, and guard-failure jumps. The dumps include the Prelude, so the small no-Prelude cases in `tests/matches.rs` are useful for studying one decision tree at a time.

We check the translation by running matches both before and after compilation. Coverage needs its own check. For small Boolean patterns we can enumerate all possible inputs and compare that direct answer with the usefulness algorithm. This catches a different kind of mistake from a wrong runtime branch.

Exercises

1. **Warm-up.** Draw the decision tree for the example and mark the failing guard's destination.
2. **Intermediate.** Add a wildcard clause before a constructor clause and predict the redundant-clause warning before running the checker.

3. **Intermediate.** Specialize a two-column matrix whose first row begins with a variable and whose second pattern is a constructor.
4. **Advanced.** Change the column heuristic in an experimental copy. Compare tree sizes while keeping the full before/after execution corpus unchanged.
5. **Advanced.** Construct malformed Core that jumps to an outer join from a non-tail expression, and explain the continuation that would be discarded.

Further Reading

- Decision trees and column-selection heuristics (Maranget 2008).
- Usefulness, exhaustiveness, and redundancy (Maranget 2007).
- Join points as a compiler intermediate representation (Maurer et al. 2017).
- Source pattern and guard semantics (S. Marlow 2010).

For a standalone Rust treatment of the same matrix operations, see Patternkit. Fun keeps its implementation in the chapter so that the checking algorithm remains visible here.

17

The Prelude and Deriving

New files

- `src/derive.rs`

Changed files

- `src/primitives.rs`
- `src/desugar.rs`
- `src/ast.rs`
- `src/core_lint.rs`
- `src/eval.rs`
- `src/lexer.rs`
- `src/lib.rs`
- `src/main.rs`
- `src/rename.rs`

Motivation

A language starts to feel like a language when you can reach for `map`, `foldr`, or `maybe` without writing them yourself every time. We now have enough Fun to write its standard library in Fun. That's a good milestone!

We'll also teach the compiler to write the repetitive parts. Given a datatype, `deriving` generates an ordinary instance for equality, ordering, or display. The generated code goes through the same checker as code we'd write by hand. There's no secret equality operation in the interpreter waiting to rescue it.

This chapter brings the interpreted language together. Take a little time to play with it before we head into the back end. Small library functions are also excellent compiler examples. Their types and definitions often tell the whole story in just a few lines.

The Language, Now

A datatype may request deriving `Eq` or a nonempty parenthesized list drawn from `Eq`, `Ord`, and `Show`. This example constructs its record fields in a different order from their declaration:

```
data P a = P {x :: a, y :: Int} deriving (Eq, Ord, Show)

main = print (Just (P {y=2,x=Left 1} :: P (Either Int Bool)))
```

Derived equality compares constructors and then their fields. Ordering uses constructor declaration order, followed by lexicographic field order. Derived `Show` prints record fields in declaration order, regardless of construction order, and inserts parentheses around nested constructor applications where needed. Lists use comma-separated brackets. `[Char]` is still a list, not a string. Functions have no derived comparison or display instance.

The completed Prelude provides list folds and traversals, finite generation, string splitting and joining, numeric conversions, character operations, and monadic control. `[a..b]` is an inclusive `Int` range. `iterateN` produces a bounded sequence. Infinite lists are unavailable because constructor fields are strict. `foldl'` and `foldl` have the same strict behavior here.

Use `++` to append lists and `<>` to append strings. Numeric literals still have fixed types, so `sum` and `product` operate on `Int` lists. Their starting values are integer literals, and `Fun`'s `Num` class has no operation for turning a literal into an arbitrary numeric type.

Putting Small Functions Together

Now we can write something compact enough to read in one breath.

```
total = sum . map (\x → x * x) . filter even

main = print (total [1..10])
```

This prints 220. Read the composition from right to left. Keep the even numbers, square each one, then add them up. Each function does one familiar job, and `(.)` connects the result of one to the input of the next. We don't have to name a loop index or describe how to advance through the list.

That is a beautiful way to express the calculation. It doesn't mean the work has vanished. `Fun` is strict, and this program builds intermediate lists. Later optimizations could remove them, but the

useful separation is already here. We can understand what the program means before choosing how to make it faster.

Theory

Imagine writing equality for a tree by hand. Empty trees are equal. Two nodes are equal when their corresponding fields are equal. Different constructors aren't equal. The datatype declaration already tells us enough to write those clauses, so let's have the compiler write them.

That's deriving. We generate an ordinary instance, then check it just as we would check a hand-written one. The Haskell Report specifies the structural behavior (S. Marlow 2010). We adapt it to Fun's records, primitive strings, and smaller Show interface.

A parameter used in a field usually introduces a corresponding instance constraint. A phantom parameter does not. For `data Token a = Token Int`, equality needs no `Eq a` dictionary because no value of type `a` is examined. Type synonyms must be expanded before deciding which parameters occur in fields. The normal instance solver then checks whether the generated context actually suffices for every field operation (Jones 1999). Deriving is not an escape from those checks.

Printing constructors has a small but important wrinkle. `Just (Left 3)` needs parentheses, but `Left 3` on its own doesn't. A function of type `a → String` can't see where its result will be used. We need to pass that surrounding precedence in as well.

Fun's Show dictionary has an internal `_showsPrec` method for this purpose. Handwritten instances inherit a default, while derived instances implement it structurally. Top-level show starts at precedence zero. Positional constructor arguments use eleven, which asks nested constructor applications to parenthesize themselves. Record fields use ordinary show inside their labelled braces.

Strictness changes another familiar library definition. An ordinary top-level `undefined = error "undefined"` would fail during startup, even if unused. Instead, an unshadowed reference to the Prelude's `undefined` is a scoped abbreviation for applying a unit-taking helper. This preserves failure at the use site without introducing a thunk or a lazy binding. The evaluation-strategy distinction from Chapter 5 remains essential (Pierce 2002).

Implementation

`derive::derive` runs immediately before class collection and lexical renaming. Generated terms retain the datatype's source span, so a rejected field operation can point back to the declaration. The pass expands aliases when collecting used parameters, selects one of three generators, and appends ordinary instance declarations:

```
/// Generate only Eq, Ord and Show. Generated terms retain the declaration span.
```

```

pub fn derive(
    sess: &mut Session,
    program: &mut Program,
) → Result<(), Stop> {
    for d in &program.data {
        if d.deriving.is_empty() {
            continue;
        }
        let build = Builder { span: d.span };
        let mut used = BTreeSet::new();
        for c in &d.constructors {
            for ty in &c.fields {
                let expanded =
                    crate::kinds::expand(ty, &program.aliases, d.span)
                        .map_err(|e| {
                            sess.emit(e);
                            Stop
                        })?;
                variables(&expanded, &mut used);
            }
        }
        for class in &d.deriving {
            let methods = match class.as_str() {
                "Eq" ⇒ vec![build.eq(d)],
                "Ord" ⇒ vec![build.ord(d)],
                "Show" ⇒ build.show(d),
                _ ⇒ {
                    sess.emit(Diagnostic::error(
                        ErrorCode::Instance,
                        "only Eq, Ord and Show can be derived",
                        d.span,
                    ));
                    return Err(Stop);
                }
            }
        };
        let context = d
            .parameters
            .iter()
            .filter(|p| used.contains(*p))
            .map(|p| PredicateSyntax {
                class: class.clone(),
                variable: p.clone(),
            })
    }
}

```

```

        .collect();
    let head = d.parameters.iter().fold(
        TypeSyntax::Name(d.name.clone()),
        |a, p| {
            TypeSyntax::App(
                Box::new(a),
                Box::new(TypeSyntax::Name(p.clone()))),
            )
        },
    );
    program.instances.push(InstanceDecl {
        class: class.clone(),
        head,
        context,
        methods,
        span: d.span,
    });
}
}
Ok(())
}

```

The builders construct the existing surface expression and pattern types. Equality generates a clause for each pair of constructors. Unequal constructors return `False`. Matching constructors compare fields from left to right, stopping at the first unequal pair:

```

fn eq(&self, d: &DataDecl) → Binding {
    let mut rows = vec![];
    for (i, a) in d.constructors.iter().enumerate() {
        for (j, b) in d.constructors.iter().enumerate() {
            let mut rhs =
                self.var(if i == j { "True" } else { "False" });
            if i == j {
                for k in (0..a.fields.len()).rev() {
                    let test = self.call(
                        "=",
                        vec![
                            self.var(&format!("{}", k)),
                            self.var(&format!("{}", k)),
                        ],
                    );
                    rhs = self.expr(Kind::If(
                        Box::new(test),
                        Box::new(rhs),
                        Box::new(self.var("False")),
                    ));
                }
            }
            rows.push(self.row(
                vec![
                    self.constructor(a, "x"),
                    self.constructor(b, "y"),
                ],
                rhs,
            ));
        }
    }
    self.binding("=", rows)
}

```

Ordering follows the same constructor-pair structure. For a matching pair it compares each field, continuing only when the result is EQ. Show instead builds string expressions from constructor names, field labels, punctuation, and recursive `_showsPrec` calls. These expressions pass through ordinary renaming, inference, dictionary elaboration, match compilation, and Core lint. Requesting an unsupported class reports E0211. Requesting `Ord` without the required equality evidence also fails through the existing instance checks.

The library lives in `prelude.fun`, embedded ahead of the user program. Its private helper references are resolved within that scope, so a user binding cannot silently change the implementation

of a Prelude function. Inclusive ranges check the upper bound before incrementing. A range ending at the largest 63-bit integer must terminate rather than wrap around.

Character classification and simple case conversion use checked-in Unicode 17.0.0 tables in `unicode.fun`. `tools/unicode.py` records the upstream source and its digest, and the data license is retained. The generated tables are ordinary compact Fun strings. General-category predicates and simple case mappings therefore execute through the language rather than adding a large set of new primitives. `isDigit` deliberately recognizes ASCII decimal digits. Not every character with a Unicode numeric classification satisfies it.

A library also teaches us where the compiler is doing unnecessary work. Walking a list shouldn't copy its entire tail at every step. We share immutable constructor field vectors with `Rc`, and `Core` lint removes local bindings on scope exit instead of copying its whole environment. These are useful reminders to look at the cost of our Rust representation as well as the Fun program.

Running It

```
$ cd crates/ch17_prelude
$ cargo run -q -- run tests/programs/prelude/derive-show-record.fun --lint
Just (P {x = Left 1, y = 2})
$ cargo run -q -- dump --pass=types tests/programs/prelude/derive-show-
record.fun
main : IO ()
```

The output demonstrates both declaration-order record fields and parentheses around the record constructor beneath `Just`. `--pass=core` shows the generated instance dictionaries. `--pass=match` shows their compiled structural tests. The separate `derive-show-tree` and `derive-show-literals` fixtures cover nested applications and escaping.

Library edge cases make good experiments. Try a singleton range at the largest `Int`, an empty list passed to a fold, or a derived instance with a phantom parameter. The little examples are doing real work here. A boundary case can explain a library definition more clearly than a long ordinary input.

Exercises

1. **Warm-up.** Derive `Eq`, `Ord`, and `Show` for a two-constructor tree and compare the result with a handwritten instance.
2. **Intermediate.** Add an unused datatype parameter and inspect whether its dictionary appears in the derived instance context.
3. **Intermediate.** Trace the precedence passed while displaying `Just (Left (Just 3))`.
4. **Intermediate.** Test a singleton range at the maximum `Int` and explain why the order of the bound check and increment matters.

5. **Advanced.** Measure constructor-tail copying on a Unicode lookup before and after immutable field-vector sharing, without changing the Fun program.

Further Reading

- Standard derived instances and the Prelude (S. Marlow 2010).
- Instance contexts and entailment (Jones 1999).
- Classes and derived instances in the Haskell Report (S. Marlow 2010).
- Operational semantics and evaluation strategies (Pierce 2002).

Part III: Backend

18

A-Normal Form and Join Points

New files

- `src/erase.rs`
- `src/anf.rs`
- `src/anf_simplify.rs`
- `src/anf_machine.rs`
- `src/anf_lint.rs`

Changed files

- `src/lib.rs`
- `src/main.rs`

Motivation

Consider $f\ (g\ x)\ (h\ y)$. To run it, we need to evaluate the function and its arguments in the right order, keep the result of $g\ x$ while computing $h\ y$, and finally make the call. Our interpreter has been doing that bookkeeping for us. A code generator needs to see the steps.

A-normal form, usually shortened to ANF, writes those intermediate results down as named bindings. Function calls and other operations then take simple operands rather than nested computations. The name sounds rather grand for what amounts to introducing some useful local variables.

This also gives us a good place to bring back the CEK machine from Chapter 5. We'll run ANF with an explicit continuation stack, now with all of Fun's data, dictionaries, IO, and join points. We can compare its answers with Core while getting the program ready for native code.

The Language, Now

Let's follow a case expression whose result is used by another computation.

```
main = print (3 + (case Just 4 of { Just x → x; Nothing → 0 })))
```

Whichever alternative returns a value, the program must add three and print once. ANF makes that common continuation explicit as a join rather than copying it into each alternative. The new `--engine=anf` selects the iterative machine and is the default. `--engine=core` retains the reference interpreter.

`--no-simplify` exposes the ANF produced before the optional rewrites. Both forms must preserve stdout, the first stderr line, and process status. In particular, strict evaluation still runs an unused binding's initializer. A failing division cannot disappear merely because its result has no uses.

Theory

In $f (g \ x) (h \ y)$, the interesting work is in the calls. The names themselves just refer to values. ANF makes that distinction part of the syntax. An *atom* is a literal or a reference to an available value. A right-hand side performs one operation on atoms, and a *Let* gives the result a name. Nested evaluation has become an explicit sequence (Flanagan et al. 1993).

Conversion receives a context describing what to do with a result. At the end of a function it returns the atom. Inside an expression it binds the atom and continues. At the end of a case alternative it can jump to a shared continuation. This representation keeps joins local and second-class, following the motivation for join points discussed by Maurer and colleagues (Maurer et al. 2017). It does not turn every function into a function that explicitly receives a continuation.

Type erasure is justified by the role of System F's type abstractions and applications: they select and describe computations but do not themselves perform runtime work (Pierce 2002). Dictionary arguments are different. They contain actual method implementations and must remain as ordinary values. IO already has its unit-function representation before erasure.

Tail position is a property of the remaining work. If a function call supplies the current function's result directly, the machine needs no return frame for that activation. A non-tail call must remember its pending computation. Constant host-stack use therefore does not imply constant memory for every recursive program. Pending non-tail work can still grow the explicit continuation stack. This is the practical advantage of making the CEK continuation visible (Felleisen and Friedman 1986).

Implementation

`erase :: erase` preserves the order of global initialization while removing executable type information. A postorder worklist drops `TyLam` and `TyApp`, keeps dictionary terms, and rejects any nested `Match` that escaped the previous pass. Read-only constructor and location metadata remains available for primitives, printers, and diagnostics.

```

/// Erase executable types only; read-only metadata supports diagnostics and
printers.
pub fn erase(mut program: core::Program) → Program {
    let globals = program
        .groups
        .iter()
        .flat_map(|g| &g.binds)
        .map(|b| (b.name.name.clone(), expr(&b.rhs)))
        .collect();
    // Consuming Core must not recursively destroy its nested expressions.
    let mut todo = program
        .groups
        .drain(..)
        .flat_map(|g| g.binds.into_iter().map(|b| *b.rhs))
        .collect::<Vec<_>>();
    while let Some(e) = todo.pop() {
        use core::Expr as C;
        match e {
            C::Lam(_, b) | C::TyLam(_, b) | C::TyApp(b, _) ⇒ todo.push(*b),
            C::App(a, b) | C::Join(_, _, a, b) ⇒ {
                todo.push(*a);
                todo.push(*b);
            }
            C::Let(g, b) ⇒ {
                todo.push(*b);
                todo.extend(g.binds.into_iter().map(|b| *b.rhs));
            }
            C::Con(_, _, es) | C::Prim(_, _, es, _) | C::Jump(_, es, _) ⇒ {
                todo.extend(es)
            }
            C::Case(e, _, _, alts) ⇒ {
                todo.push(*e);
                todo.extend(alts.into_iter().map(|a| a.rhs));
            }
            C::Match(_) ⇒ {
                unreachable!("match compilation precedes erasure")
            }
            _ ⇒ {}
        }
    }
    Program {
        globals,
        metadata: program,
    }
}

```

The ANF Grammar

Here is the control language in symbolic form. We use a for an atom, r for a right-hand side, and e for a complete ANF body. Each row corresponds to a Rust variant below.

$e = \text{let } x = r \text{ in } e$	strict binding
$ \text{letrec } \{\overline{x = \lambda \bar{y}. e_1}\} \text{ in } e_2$	recursive functions
$ \text{join } j(\bar{x}) = e_1 \text{ in } e_2$	local continuation
$ \text{jump } j(\bar{a})$	jump to a join
$ \text{case } a \text{ of } \{\overline{q \bar{x} \rightarrow e}\}$	branch on an atom
$ \text{return } a$	return a value
$ \text{tailcall } a(\bar{a})$	replace this activation
$ \text{fail}_s$	pattern failure at span s

An overbar means a sequence. The name j identifies a join point, while q is an alternative's constructor tag, literal, or default marker. Constructor alternatives bind their fields to $\bar{x}$. Literal and default alternatives have no field binders.

The smaller pieces explain the “normal form” part of the name. An operand is always an atom, never another call waiting to happen.

$a = x$	local variable
$ g$	global reference
$ \ell$	literal
$r = a$	atom
$ a(\bar{a})$	function call
$ K(\bar{a})$	constructor
$ p(\bar{a})$	primitive operation
$ \lambda \bar{x}. e$	function value
$ \text{field } a \ i$	field at index i

Here K names a constructor and p a primitive. Primitive source spans are omitted from the grammar. Unlike Core's lambdas, these lambdas can take several parameters, and there are no type abstractions or type applications left.

The distinction between a call and a tail call is now visible in the syntax. $\text{let } x = f(a) \text{ in } e$ saves work to do after the call. $\text{tailcall } f(a)$ has no such body. A jump also has no following body, but stays inside the current function and transfers to a local join point.

The ANF control language separates ordinary bindings, recursive closures, branches, joins, and terminal transfers. `Fail` preserves the source span of a pattern failure:

```
#[derive(Clone, Debug)]
/// Jumps transfer within a function; tail calls replace its activation.
pub enum Anf {
    Let(Var, Rhs, Body),
    LetRec(Vec<(Var, Lambda)>, Body),
    Join(Id, Vec<Var>, Body, Body),
    Jump(Id, Vec<Atom>),
    Case(Atom, Vec<(AltCon, Vec<Var>, Body)>),
    Ret(Atom),
    TailCall(Atom, Vec<Atom>),
    Fail(Span),
}
```

The converter's context has only three cases. `finish` interprets a context once the result has become an atom. Pending conversions live in a work vector, so a long expression does not become a long chain of Rust calls:

```
#[derive(Clone)]
enum Context {
    Return,
    Jump(Id),
    Bind(Var, Body),
}

fn finish(&mut self, a: Atom, k: Context) → Body {
    Rc::new(match k {
        Context::Return ⇒ Anf::Ret(a),
        Context::Jump(j) ⇒ Anf::Jump(j, vec![a]),
        Context::Bind(v, b) ⇒ Anf::Let(v, Rhs::Atom(a), b),
    })
}
```

For a non-tail case, conversion gives the remaining context a fresh join parameter and converts each alternative with a jump context. Existing Core joins survive. Adjacent lambdas merge into a multi-parameter lambda. This is syntactic arity, not arity inferred from a function type.

Application conversion must respect the order of curried calls. In `f a b`, the call `f a` may fail before `b` is evaluated. Combining both arguments into a single eager batch would change which error appears first. The converter batches only up to a known syntactic arity. Unknown calls consume one

argument at a time, and over-application preserves the boundary after the known prefix. The known-prefix-error regression checks this ordering explicitly.

`anf_simplify` propagates atoms, selects known-constructor branches, removes unused total right-hand sides, and reduces immediately applied saturated lambdas. Calls, effects, and potentially failing primitives are retained. The beta rule also checks whether the function escapes: passing a function to itself, as in the inherited `id id` example, means its binding cannot simply be deleted. Structural ANF lint independently checks names, scopes, join arities, and fresh binders before execution.

The machine is a loop over code, application, and return states. Its frames remember a non-tail binding continuation or arguments left after an oversaturated call:

```
enum State {
    Code(Body),
    Apply(Value, Vec<Value>),
    Return(Value),
}
enum Frame {
    Bind(Var, Body, Env, Joins),
    Apply(Vec<Value>),
}
```

Closures carry a lambda and a shared lexical environment. Recursive groups are built once in a machine-owned table. Their closures hold the group's index, which avoids reference-count cycles and rebuilding closures at every call. Applying too few arguments returns a partial application. Excess arguments are applied to the returned function. Tail calls replace the current state, and jumps enter a lexically captured join environment. Neither transfer recursively invokes the Rust interpreter. Primitive values cross a data-only adapter to the existing primitive implementation. Control flow does not call the Core evaluator.

Running It

```
$ cd crates/ch18_anf
$ cargo run -q -- run tests/programs/anf/case-continuation.fun --lint
7
$ cargo run -q -- dump --pass=types tests/programs/anf/case-continuation.fun
main : IO ()
```

Use `--pass=erase` to see type-free Core and `--pass=anf` to see named operations. Repeat the latter with `--no-simplify` to distinguish conversion from rewriting. Repeat execution with `--`

`engine=core` to compare the reference result. Dumps include the Prelude. The small snapshots in `tests/anf.rs` isolate individual transformations such as a non-tail case and a guard-failure join.

Alongside the handwritten examples, `testkit/src/gen.rs` generates small well-typed programs containing arithmetic, local bindings, branches, recursion, and lists. Both engines must agree with an independent wrapping-integer calculation. We save the source and seed of a failure so an unexpected answer becomes a program we can debug.

A million-call sum is a useful test of the machine’s control flow. We measure the number of pending continuations, rather than just checking that it finishes. A tail call should replace the current computation. A non-tail call may add pending work to the heap-allocated continuation stack. Neither needs another Rust call frame. Core uses its own explicit machine, so comparing it with ANF also checks that conversion hasn’t changed the answer.

Exercises

1. **Warm-up.** Locate the continuation after the example’s case in an unsimplified ANF dump.
2. **Intermediate.** Make two arguments fail with different messages and check which must run first for a known call and an unknown call.
3. **Intermediate.** Explain why removing an unused division can be wrong even though integer division has no IO effects.
4. **Advanced.** Instrument the machine on tail-recursive and non-tail-recursive sums and compare maximum continuation depth.
5. **Advanced.** Extend the deterministic generator with one new well-typed construct and an independent expected-result calculation.

Further Reading

- Direct representations and continuation-based compilation (Flanagan et al. 1993).
- Join points and shared continuations (Maurer et al. 2017).
- The CEK machine’s control, environment, and continuation (Felleisen and Friedman 1986).
- Continuation-based compiler construction (Appel 1992).
- Polymorphism and type erasure (Pierce 2002).

19

Closure Conversion

New files

- `src/closure.rs`
- `src/flat.rs`
- `src/flat_machine.rs`
- `src/flat_lint.rs`

Changed files

- `src/lib.rs`
- `src/main.rs`

Motivation

A function can outlive the call that created it. That is easy to write in Fun, but it leaves our back end with a question. Where do the variables from that old call live now?

Our interpreters answered by storing an environment with each function. We'll make the same idea explicit in the program we compile. Each lambda becomes a function in a program-wide table. Its closure carries the values that function needs from the surrounding scope.

We'll also work out how to call these functions. When we know the target, we can call it directly. When a function arrives as an argument or comes back from another call, we need to inspect its closure. Partial application adds one more interesting case. We'll model all of this in a Flat interpreter before we start laying out the objects in native memory.

The Language, Now

Here's a small function that remembers `x` and can be passed to `map`.

```
make x = \y → x+y

main = print (map (make 20) [1,2,3])
```

Every call of the returned function needs the supplied value 20. Here ANF uncurrying merges the adjacent lambdas into one global function, so its partial application retains that value in a PAP. The overloaded addition also needs a Num dictionary. Local lambdas separated by a binding or branch instead require explicit captured environments. Partial application, oversaturated application, recursive local functions, and class dictionaries used as values must all keep working. `--engine=flat` is now the default. Core and ANF remain selectable.

A closure environment is compiler-created storage. The source language gains no mutable field or reference operation. The only mutation introduced here initializes capture slots for a recursive group before user code can observe the closures.

Theory

Free-variable analysis identifies the values a lambda needs beyond its own parameters. Globals need no local capture slot because the program can address them directly. For each remaining free variable, conversion chooses a stable slot, stores the value when constructing the closure, and replaces lexical access with an environment load in the lifted function.

Closure conversion and lambda lifting are related but different choices. Closure conversion packages captured values into an environment. Lambda lifting can instead add them as explicit function parameters when call sites permit it (Appel 1992). This chapter keeps one environment mechanism for local functions, even non-escaping ones. Optional lambda lifting is deferred. A known local call can still bypass general apply while passing its environment explicitly.

A source function can be called a little at a time. Our lifted machine function expects a fixed number of arguments. How do we reconcile those two views?

If the function's syntactic arity is n and we have fewer arguments, we save them in a *partial application*, or PAP. If we have exactly n , we enter the function. If we have more, we call it with n and apply the leftover arguments to the function it returns. That's the eval/apply protocol in miniature (Marlow and Peyton Jones 2004). A single calling mechanism now handles all three cases.

Mutual recursion gives us a chicken-and-egg problem. If even captures odd and odd captures even, which closure can we build first? We'll allocate both with placeholder capture slots, then fill the slots once both addresses exist. No source code can call them until this initialization is finished. Requiring recursive bindings to be functions makes this possible without trying to evaluate an unfinished recursive value.

Implementation

The Flat Grammar

Flat keeps ANF's explicit control flow, but functions now live in a separate function table. There is no lambda form inside an executable body. Instead, we allocate a closure that names a lifted function and stores its captures.

$e = \text{let } x = r \text{ in } e$	strict binding
$\text{case } a \text{ of } \{\overline{q \bar{x} \rightarrow e}\}$	branch on an atom
$\text{join } j(\bar{x}) = e_1 \text{ in } e_2$	local continuation
$\text{jump } j(\bar{a})$	jump to a join
$\text{return } a$	return a value
$\text{tail } f[\rho](\bar{a})$	known tail call
$\text{tailapply } a(\bar{a})$	unknown tail call
fail_s	pattern failure at span s

We keep the ANF notation for atoms a , join names j , alternative tags q , and sequences marked with an overbar. The name f identifies a lifted function. Its optional environment operand ρ is either an atom or absent. A known call names its code directly. An unknown call must inspect a closure or PAP to find out what to do.

$a = x \mid g \mid \ell$	local, global, or literal
$\rho = a \mid \text{none}$	optional environment
$r = a$	atom
$\text{call } f[\rho](\bar{a})$	known call
$\text{apply } a(\bar{a})$	unknown call
$\text{con } K(\bar{a})$	allocate a constructor
$\text{closure } f[\bar{a}]$	allocate a closure
$\text{env } x[i]$	read a capture
$\text{field } a[i]$	read a constructor field
$p(\bar{a})$	primitive operation
$\text{patch } a[i] = a'$	fill a closure capture

Here K is a constructor, p a primitive, and i a slot index. Primitive source spans are omitted. `patch` is an internal operation used while tying recursive closures together, not mutable state available to a Fun program. The `LetRec` from ANF has disappeared into allocations followed by these patches.

Lifted Functions and Calls

Flat IR removes lambda expressions from executable bodies. Each lifted function records its parameters, optional environment parameter, body, and ordered capture list:

```
#[derive(Clone, Debug)]
/// Only lifted local lambdas receive an environment parameter.
pub struct Func {
    pub name: FuncId,
    pub source: Var,
    pub env: Option<Var>,
    pub params: Vec<Var>,
    pub body: Body,
    pub captures: Vec<Var>,
}
```

`closure::convert` collects lambda definitions, computes free variables, and analyzes uses. Capture order follows unique-name order, so dumps and slot indexes are deterministic. Top-level function entries have no environment parameter. Lifted local entries do, even when their environment is empty. Prefix `EnvField` bindings load captures at the start of each local function.

A call is known only when its argument count matches the syntactic arity and its head is a global function or a non-escaping local lambda. Merely having a function type is insufficient. The actual decision is small enough to inspect:

```
fn known(&self, a: &Atom, n: usize) → Option<(FuncId, Option<Atom>)> {
    let (Atom::Var(v) | Atom::Global(v)) = a else {
        return None;
    };
    let info = self.infos.get(v)?;
    (n == info.lambda.params.len() && (info.global || !info.escapes))
        .then(|| (info.id, (!info.global).then(|| a.clone())))
}
```

Escape analysis regards saturated use as a callee differently from returning, storing, or passing the function as an argument. A known local call still carries its closure as an environment operand. Other calls use `CallUnknown`. Tail position selects the corresponding tail-call form. Partial application therefore continues through `apply` even when the source spelling names a familiar function.

The right-hand-side language makes allocation and access visible:

```
#[derive(Clone, Debug)]
/// PatchClosure is a compiler-only operation used after all letrec
/// allocations.
pub enum Rhs {
    Atom(Atom),
    CallKnown(FuncId, Option<Atom>, Vec<Atom>),
    CallUnknown(Atom, Vec<Atom>),
    AllocCon(ConId, Vec<Atom>),
    AllocClosure(FuncId, Vec<Atom>),
    EnvField(Var, u32),
    Field(Atom, u32),
    Prim(Op, Vec<Atom>, Span),
    PatchClosure(Atom, u32, Atom),
}
```

For a LetRec, conversion emits every AllocClosure before any PatchClosure. The initial capture slots contain placeholder integer literals in the IR. After patching, each contains its actual captured value. Patch operations return unit into unused temporary binders. All patches precede the converted body, so no source expression can call a partially initialized closure.

Flat lint checks each lifted function independently. It rejects an implicit free variable, unknown function identity, incorrect known-call arity, missing or unexpected environment operand, out-of-bounds environment load, or invalid join transfer. Capture binder identities can recur in different functions, since each environment load rebinds the name in a separate function scope.

The Flat machine owns an indexed arena of closure records. A closure value is an arena index. Environment slots contain ordinary interpreter values, including other closure indexes. That permits recursive cycles without reference-count cycles. PAPs store a function value and supplied arguments. The arena lives for one execution and models object relationships. It does not yet model reclamation.

Execution is an iterative loop with separate code, known-call, apply, and return states. Known calls validate the entry's arity and environment. Apply flattens PAP arguments, handles short and exact calls, and keeps excess arguments in an explicit continuation. Joins remain lexical control transfers within a function. Primitive operands cross a data adapter, but evaluation never delegates a Flat body to the Core or ANF interpreter. Agreement is meaningful because the three engines implement control flow independently.

Running It

```
$ cd crates/ch19_closure
$ cargo run -q -- run tests/programs/closure/capture-return.fun --lint
[21,22,23]
$ cargo run -q -- dump --pass=types tests/programs/closure/capture-return.fun
main : IO ()
```

Dump this file with `--pass=flat` to find the partial application of `make`. The global entry has no environment. `Apply` saves its dictionary and integer arguments in a PAP. In contrast, `escape-recursive.fun` returns a local recursive closure capturing an outer value, and `known-local.fun` uses direct local calls. `pap-chain.fun` supplies arguments in stages. `over-application.fun` supplies arguments to a function whose result is itself a function. The small Flat snapshots in `tests/anf.rs` omit unrelated Prelude functions by selecting reachable code.

Try supplying arguments in stages, returning a local function, and making two local functions call each other while both capture an outer value. These all look natural in source, but exercise different parts of the closure machinery. Core, ANF, and Flat should agree on their results. The Flat machine also checks that a long tail-recursive sum keeps a bounded number of pending continuations.

Exercises

1. **Warm-up.** Trace how uncurrying and partial application preserve the example's integer argument and `Num` dictionary.
2. **Intermediate.** Change a local function from saturated-only use to returning it as a value. Explain the resulting call classification.
3. **Intermediate.** Trace a PAP receiving another argument, followed by an oversaturated call whose first result is a closure.
4. **Advanced.** Construct mutually recursive local functions that also capture an outer integer. Draw their environment graph after patching.
5. **Advanced.** Prototype lambda lifting for non-escaping functions and compare allocation counts while retaining differential agreement.

Further Reading

- Eval/apply, partial application, and calling conventions (Marlow and Peyton Jones 2004).
- Closure conversion in a complete compiler (Appel 1992).
- Strict functional implementation and value representation (Leroy 1990).
- Local joins versus first-class continuations (Maurer et al. 2017).

20

Data Representation and the Runtime

New files

- `rt/src/layout.rs`
- `rt/src/abi.rs`
- `rt/src/heap.rs`
- `rt/src/apply.rs`
- `rt/src/primitives.rs`
- `rt/build.rs`
- `rt/src/lib.rs`

Changed files

- `src/primitives.rs`
- `src/lib.rs`
- `src/main.rs`

Motivation

Our interpreters have enjoyed a useful luxury. Rust enums describe their values, and Rust helps manage their storage. The machine code we're about to generate won't have those enums. We need to decide exactly what an integer, a list node, or a closure looks like in memory.

Polymorphism makes the question especially interesting. One compiled `id` function should accept an integer, a string, or a tree. We'd like to pass all of them through the same calling convention. A uniform machine word gives us that common representation, with a little tagging to distinguish immediate values from pointers.

We'll build the runtime before connecting LLVM. This lets us inspect the memory layout and application protocol on their own. The interpreters will also reuse its primitive operations. We shouldn't need three slightly different ideas of how to print a floating-point number.

The Language, Now

The programs we write don't change here. Our new representation must still give us wrapping 63-bit integers, Unicode characters, UTF-8 strings, and strict constructor fields. Changing how a value fits in memory shouldn't change what a Fun program can observe.

This inherited program exercises floating display for NaN, infinity, and signed zero:

```
_old = (0.0/0.0, 1.0/0.0, -0.0)

main = print _old
```

The runtime is available as `fun-rt-20`, built both as an `rlib` for the compiler and a `staticlib` for external linking. The chapter compiler still runs Core, ANF, or Flat. Linking the runtime into a C test does not yet provide `fun build`. LLVM object generation is the next chapter's task.

Theory

A value is a 64-bit word. An immediate has low bit one. An aligned pointer has low bit zero. Encoding an integer shifts it left one bit and sets the tag, leaving 63 signed bits. Decoding uses an arithmetic shift. The same immediate encoding holds character scalar values and nullary constructor tags.

```
pub const fn integer(n: i64) → Word {
    ((n as u64) << 1) | 1
}
pub const fn signed(w: Word) → i64 {
    (w as i64) >> 1
}
```

Notice something odd about word 1. It can represent unit, `False`, the empty list, or `Nothing`. That isn't an ambiguity the runtime has to resolve. The type checker already established which interpretation the program can use. We don't inspect a list as though it were a boolean.

This is what makes the uniform representation useful. A polymorphic function can pass words around without needing a different memory layout for every instantiation. Tagging and boxing give strict functional implementations a practical way to make that work (Leroy 1990).

Heap objects begin with a header. Its low eight bits identify the object kind, the next 24 bits carry a constructor tag, and the high 32 bits count payload words. Constructor tags are local to a datatype

and follow declaration order. The fixed Prelude tags keep runtime-created lists and optional values compatible with compiler-created ones.

Boehm finds potential pointers by inspecting words in memory (Boehm and Weiser 1988). That saves us from describing every live pointer in the generated code. The tradeoff is that an unrelated bit pattern can look like an address and keep an otherwise dead object alive. The collector also can't freely move an object when it doesn't know which apparent pointers are real references to update.

A precise moving collector needs our help. We'll build that cooperation in Chapter 24, using an explicit account of roots. Henderson and the GC handbook develop this distinction (Henderson 2002; Jones et al. 2023).

Implementation

rt/src/layout.rs is the shared ABI definition, re-exported by the compiler as `runtime_layout` to avoid its existing offside-layout module. Header construction checks that the constructor tag fits:

```
/// Bits 0..8: kind, 8..32: tag, 32..64: payload word count.
pub const fn header(kind: u8, tag: u32, size: u32) → Word {
    assert!(tag ≤ 0x00ff_ffff);
    kind as Word | ((tag as Word) << 8) | ((size as Word) << 32)
}
```

The payload layouts are deliberately simple:

Kind	Payload after the header
CON	Constructor fields, one word each
CLOSURE	Slow code address, arity, then captures
PAP	Closure word, argument count, then supplied arguments
DOUBLE	One word containing binary64 bits
STRING	Byte length, then padded UTF-8 bytes
FORWARD	Reserved for the later copying collector

Reading the Heap

Start with the word held in a variable. Its low bit tells us whether the value fits in that word or points to an object. A pointer addresses the **header**, not the first field. All offsets below are bytes, and every row occupies eight bytes.

Value word (64 bits)

Int 42: [85] low bit 1

Heap value: [p] low bit 0

|

v

+-----+	
header	+0
payload[0]	+8
payload[1]	+16
...	
+-----+	

Header bits (most significant first)

63	32 31	8 7	0
+-----+			
size: 32 bits	tag: 24 bits	kind: 8 bits	
+-----+			

For example, `Just 42` needs one payload word. Its constructor tag is 1. A `Double` also needs one payload word, but that word contains floating-point bits rather than an encoded `Fun` value.

`Just 42`

+-----+	
CON, tag=1, size=1	+0
85 (tagged Int 42)	+8
+-----+	

`Double 3.0`

+-----+	
DOUBLE, tag=0, size=1	+0
0x4008000000000000	+8
+-----+	

A closure packages code with the values it captured. Imagine the local function $f\ x\ y = k + x + y$, with $k = 10$. Its arity is two and its environment has one capture. Applying f to just 20 creates a partial application, or PAP, which points back to the closure and remembers that argument.

```

PAP for f 20
+-----+
| PAP, tag=0, size=3          | +0
| closure word                | +8 ----+
| argument count: 1          | +16   |
| 41 (tagged Int 20)         | +24   |
+-----+                       |
      +-----+
      |
      | Closure for f
      | +-----+
      +->| CLOSURE, tag=0, size=3      | +0
          | slow entry address        | +8
          | arity: 2                  | +16
          | capture k: 21 (Int 10)    | +24
          +-----+

```

The arrow denotes a pointer to the whole closure object. Arity and argument count are raw counts, not tagged integers. Captures and saved arguments are Fun value words, so they may themselves point to other objects. The slow entry is a code address. Calling the PAP with 22 combines the saved 20 with 22, then enters the closure with both arguments. The closure's capture remains 10.

A string stores a byte count followed by UTF-8 bytes padded to whole words. For "hi", the payload has two words: the count and one word of byte storage. The padding is not part of the string, and no terminating zero is required.

```

String "hi"
+-----+
| STRING, tag=0, size=2       | +0
| byte length: 2             | +8
| 68 69 .. .. .. .. ..     | +16
+-----+
Hex bytes; .. denotes padding.

```

`rt_alloc` takes a total word count including the header. `rt_alloc_atomic` takes bytes and asks Boehm not to scan the contents. Doubles and strings use atomic allocation because their payload contains no value pointers. A string's stored length counts bytes. The source string-length primitive counts Unicode scalars. Alignment is eight bytes throughout.

Allocation must protect input fields while the collector runs. A Rust Vec is allocated by Rust's allocator, and Boehm does not automatically trace through that buffer. Roots owns a stable boxed word buffer and registers its interior range with the collector. Its destructor removes the range

before freeing the buffer. Two unregistered guard words keep adjacent registrations distinct. Object construction uses this protection before allocating and copying fields:

```
/// Allocate a scanned object, protecting fields even when supplied in a Rust
Vec.
pub fn object(kind: u8, tag: u32, fields: &[Word]) → Word {
    let roots = Roots::new(fields);
    let size = u32::try_from(fields.len()).expect("object too large");
    let p = rt_alloc(fields.len() + 1);
    // SAFETY: p has header + fields words; protected input and destination do
    not overlap.
    unsafe {
        p.write(layout::header(kind, tag, size));
        ptr::copy_nonoverlapping(
            roots.as_slice().as_ptr(),
            p.add(1),
            fields.len(),
        );
    }
    p as Word
}
```

The runtime initializes Boehm on the main thread. Foreign Rust worker threads register their stacks and unregister when they exit. Direct ABI tests force collection inside calls and while cyclic closure environments are live. Those cases would expose assumptions that work only when allocation happens not to collect.

A closure stores a C-convention slow entry and its syntactic arity. The entry receives the closure word and a pointer to exactly that arity's arguments. Closure construction simply builds the corresponding object:

```
/// Store a code address, syntactic arity, then captured values.
pub fn closure(code: Slow, arity: usize, captures: &[Word]) → Word {
    assert!(arity > 0);
    let mut fields = vec![code as usize as Word, arity as Word];
    fields.extend_from_slice(captures);
    heap::object(layout::CLOSURE, 0, &fields)
}
```

`rt_apply` roots its argument buffer, combines saved PAP arguments with new ones, and repeats the eval/apply protocol from Chapter 19 (Marlow and Peyton Jones 2004). A short call allocates a PAP, an exact call enters code, and an oversaturated call continues with the returned closure and remaining

arguments. `rt_apply1` through `rt_apply4` use the same implementation. `rt_closure_set` checks and patches capture slots only during unpublished recursive-group initialization.

The loop in `apply` does not guarantee native tail-call elimination through C slow entries: the entry itself may call `apply` again. Chapter 21's known fast calls need their own LLVM tail-call convention. The iterative ANF and Flat machines already avoid recursive host calls, but that is a property of those interpreters, not an automatic property of the C ABI.

`primitives.rs` separates a safe, typed value API from the word ABI in `abi.rs`. All three interpreters adapt their data values to that safe API. They retain their own control-flow engines and heap models. Native-facing exports decode words according to their checked source types, call the shared semantics, and encode the result. Unsafe accesses are confined to the runtime and documented with typed, live-object contracts. Unwinding across an extern "C" entry aborts. Language failures instead follow the explicit runtime error and exit paths.

`show_double` starts with Rust's shortest round-trip digits, then normalizes notation. Nonzero magnitudes from $1e-4$ inclusive to $1e16$ exclusive use positional notation. Other finite values use a signed exponent. Zero retains its sign, and special values spell NaN, Infinity, and -Infinity. We check boundary values through both the safe API and the word ABI, including whether finite values survive a print-and-read round trip with the same bits. `readInt` accepts optional surrounding whitespace and a decimal sign, then rejects values outside the signed 63-bit range. Floating round uses ties-to-even and saturates. NaN converts to zero.

Running It

The larger `double-specials` fixture also checks the positional/scientific notation boundary:

```
$ cd crates/ch20_runtime
$ cargo run -q -- run tests/programs/runtime/double-specials.fun --lint
-0.0
NaN
Infinity
-Infinity
0.0001
1.0e+16
$ cargo run -q -- dump --pass=types tests/programs/runtime/double-specials.fun
main : IO ()
```

From the repository root, `cargo test -p fun-rt-20` exercises primitive semantics and the stand-alone ABI harness. `python3 scripts/check_runtime_abi.py` builds the static runtime, asks rustc for the host's native static libraries, links the C smoke program with the system compiler, and checks that it prints 42. `just build` also builds the runtime explicitly before the workspace. Linux requires

`libgc-dev`. MacOS uses Homebrew’s `bdw-gc`, with `GC_LIB_DIR` available for another installation location.

A collector bug often hides until allocation happens at exactly the wrong moment. The runtime tests force collection while recursive closures are being built and while calls hold live arguments. They also exercise partial and oversaturated calls, Unicode strings, and output before errors. These are the places where a correct-looking object layout has to work as part of a running program.

Exercises

1. **Warm-up.** Compute the encoded words for `-1`, `0`, `True`, and `unit`. Explain why equal words need not represent equal source values.
2. **Intermediate.** Draw a two-field constructor and a one-argument PAP, including their headers and payload sizes.
3. **Intermediate.** Explain why a live pointer stored only in an ordinary Rust vector can disappear from Boehm’s view.
4. **Advanced.** Add a forced collection between recursive closure allocation and patching in an ABI test, keeping every unfinished object rooted.
5. **Advanced.** Identify what extra metadata a moving collector would need to update closure captures, PAP arguments, globals, and live local values.

Further Reading

- Conservative collection without compiler cooperation (Boehm and Weiser 1988).
- Root discovery with compiler cooperation (Henderson 2002).
- Representation choices in a strict functional runtime (Leroy 1990).
- Eval/apply and partial applications (Marlow and Peyton Jones 2004).
- Collection algorithms and implementation tradeoffs (Jones et al. 2023).
- Takenobu Tani’s GHC (STG, Cmm, asm) Illustrated (pdf / github), for GHC’s heap objects, stack layout, and calling convention next to ours.

21

LLVM Code Generation

New files

- `src/llvm/mod.rs`
- `src/llvm/control.rs`
- `src/llvm/values.rs`
- `src/llvm/primitives.rs`
- `src/native.rs`

Changed files

- `src/flat.rs`
- `build.rs`
- `src/flat_lint.rs`
- `src/lib.rs`
- `src/main.rs`

Motivation

We can now run Fun programs, but they still need our interpreter to do the work. In this chapter we'll produce an executable you can launch from the shell. That's a pretty satisfying destination for a project that began with a calculator!

Much of the preparation is already behind us. Pattern matching has become branches, closures have explicit environments, and Flat IR tells us where calls and tail transfers belong. LLVM will handle instruction selection, register allocation, and machine optimization.

We still have to tell it how our values fit in memory and how our functions call one another. If those decisions disagree with the runtime, LLVM can efficiently compile the wrong program. We'll

work through that boundary carefully, using the representations we already understand to keep the new work manageable.

The Language, Now

The language is unchanged. `fun-ch21 build FILE -o OUT` produces an executable. `run` still selects an interpreter. `-O0` through `-O3` control optimization, with `-O2` the default. `--dump=llvm` shows the module before optimization and `--dump=llvm-opt` shows it afterward.

The acceptance program brings together records, dictionaries, higher-order functions, IO, and a strict accumulator loop:

```
-- engines: native
data Shape = Circle Double | Rect Double Double
  deriving (Show)

data Point = Point { x :: Double, y :: Double }
  deriving (Eq, Show)

class Area a where
  area :: a → Double

instance Area Shape where
  area (Circle r) = 3.0 * r * r
  area (Rect w h) = w * h

norm :: Point → Double
norm p = sqrt (p.x * p.x + p.y * p.y)

sumTo :: Int → Int
sumTo n = go 0 n
  where
    go acc 0 = acc
    go acc k = go (acc + k) (k - 1)

main :: IO ()
main = do
  let p = Point { x = 3.0, y = 4.0 }
  print (norm p)
  mapM_ (print . area) [Circle 1.0, Rect 2.0 3.0]
  print (map (.x) [p, p { x = 10.0 }])
  print (p == p { y = 4.0 })
  print (sumTo 10000000)
```

The final call performs ten million iterations. Getting the number right is only part of the challenge. The native stack should stay bounded too, just as it does for a tail call in our explicit machines.

Theory

LLVM uses *static single assignment*, or SSA. Each instruction result has one definition, so a name always identifies the same computed value. Our unique Flat binders already fit this idea. A `Let` associates a binder with an LLVM value, while a `Join` becomes a basic block that several paths can enter.

The interesting question is how those paths pass values to the block. LLVM uses *phi nodes*. Each join parameter becomes a phi, and a `Jump` supplies its incoming value before branching to the block. The join points we’ve already built give this translation a natural shape (Maurer et al. 2017).

Imagine two paths that meet at a join. One supplies 10, the other 20. The phi takes the value associated with whichever path we arrived from. It doesn’t test the original condition again.

There is an easy codegen mistake here. Emitting one Flat operation can create several LLVM blocks. Division, for example, adds a zero check. The phi’s incoming edge must name the block we actually leave after that check, not the block where we started emitting the expression. Record the builder’s **current** block after emitting the jump arguments.

Flat construct	LLVM translation
Let	Instructions and a name-to-value entry
Case	Tag inspection and switch
Join	A block with phi parameters
Jump	Phi incoming values and br
Known tail call	<code>tail call tailcc</code> , immediately followed by <code>ret</code>
Unknown call	C-convention runtime apply

Tail position alone is insufficient for bounded native stack use. LLVM requires compatible calling conventions and a suitable return sequence. We give every fast entry `tailcc`, mark known tail calls, and immediately return the call’s result. Internal linkage also meets the visibility requirement for x86-64 position-independent code. These choices follow LLVM’s tail-call rules. The guarantee applies to known saturated calls, including mutual recursion. It does not extend across the runtime’s C eval/apply boundary.

Implementation

Establish the target first

`llvm/mod.rs` owns the module, builder, target machine, and symbol tables. Its LLVM values borrow a `Context` owned by the caller. This makes the lifetime boundary explicit and avoids a self-referential Rust structure. All function entries and global slots are declared before bodies, so forward and mutually recursive calls need no special case.

The chapter pins Inkwell 0.10.0 with `llvm22-1-prefer-dynamic` and LLVM 22. Target initialization runs once. The target machine uses the host triple, the generic CPU, and position-independent relocation. Its triple and data layout are installed on the module **before** emission. A 32-bit pointer target is rejected. For a smaller introduction to the bindings, see the examples in `Compiler Crates`.

Builder operations return errors. Flat lint checks the input, and LLVM's verifier checks every completed module, regardless of `--lint`. Optimization uses the new pass manager and verifies again:

```
/// Run the new pass manager only after constructing a complete, valid module.
pub fn optimize(&self, level: u8) → Result<()> {
    if level > 3 {
        return Err(Error("optimization level must be 0..3".into()));
    }
    self.module
        .run_passes(
            &format!("default<0{level}>"),
            &self.machine,
            PassBuilderOptions::create(),
        )
        .map_err(|e| Error(e.to_string()))?;
    self.verify()
}
```

The verifier can catch a missing terminator, an invalid phi, or a call with the wrong type. It's an excellent first line of defence. But perfectly valid LLVM can still be a wrong translation of Fun, perhaps evaluating arguments in the wrong order or mishandling integer overflow. We catch those mistakes by running the same programs through the interpreters and native code.

Traverse with a worklist

The emitter never recursively descends the Flat tree. Pending blocks live in a vector, and a loop consumes each straight-line `Let` spine. The terminal handler emits a return or transfer, or queues the bodies of joins and alternatives.

The environment is a table of unique SSA definitions. Lexical scope was checked by Flat lint, so emission does not clone a scope map at each branch. A join's phis are created before its callers are

visited. Incoming edges can be added as those callers are emitted. Each completed block has exactly one terminator.

The worklist approach also applies to lint, printing, and cleanup. That last one is easy to forget. Rust's generated destructors can recursively free a deep Rc tree after compilation has apparently succeeded. We drain those edges with a worklist too. Deep Flat trees are checked, printed, emitted, verified, and released on a small Rust stack. This checks our Flat traversals, rather than assuming a successful code-generation loop makes every other pass safe.

```
pub(super) fn body(
    &mut self,
    b: &Body,
    mut env: Env<'ctx>,
    mut joins: Joins<'ctx>,
) → Result<()> {
    // Unique binders index SSA values; lint already checked scopes.
    // Branches share this table without cloning lexical environments.
    let block = self.builder.get_insert_block().unwrap();
    let mut work = vec![Block {
        body: b.clone(),
        block,
    }];
    while let Some(Block { body, block }) = work.pop() {
        self.builder.position_at_end(block);
        let mut cursor = body.as_ref();
        while let Flat::Let(v, r, rest) = cursor {
            let value = self.rhs(r, &env)?;
            env.insert(v.clone(), value);
            cursor = rest;
        }
        self.terminal(cursor, &mut env, &mut joins, &mut work)?;
    }
    Ok(())
}
```

Preserve the word representation

Every Fun value is an LLVM i64, using Chapter 20's shared runtime_layout. Loads convert a word to a pointer and index an array of words. Inkwell's opaque pointers require the accessed element type explicitly:

```
pub(super) fn slot(
    &self,
    pointer: PointerValue<'ctx>,
    index: u64,
) → Result<PointerValue<'ctx>> {
    // SAFETY: GEP indexes i64 words using the checked ABI layout.
    Ok(unsafe {
        self.builder.build_gep(
            self.word,
            pointer,
            &[self.n(index)],
            "slot",
        )?
    })
}
```

The unsafe boundary constructs LLVM. It does not dereference a Rust pointer. The generated access still depends on checked constructor arity and the runtime layout. We use ordinary GEP rather than claiming additional inbounds guarantees. A constructor field is one word past the header. A closure capture also skips the slow-entry address and arity.

Nullary constructors are immediates. Other constructors are allocated with `rt_alloc`, then initialized in slot order. A case on a type containing both forms tests the low bit before loading a header. Types containing only one form need no such branch. Integer and character literal cases switch directly on tagged words. String equality has already become primitive tests during match compilation.

String literals are aligned, read-only objects containing their header, byte length, and padded UTF-8 bytes. A retained data table records their root words. `llvm.used` keeps that table through optimization and linking. These static objects are never moved or freed. Heap strings and doubles use atomic allocation because they contain no pointers. Generated globals and live machine registers/stack slots remain visible to Boehm (Boehm and Weiser 1988).

Integer operations decode with arithmetic shift, compute in `i64`, and retag. We attach neither `nsw` nor `nuw`, since wrapping is defined behavior. A zero divisor branches to `rt_div_zero` before division. The smallest Fun integer divided by minus one fits in the wider intermediate `i64`. Retagging supplies the required 63-bit wrap. `div` and `mod` adjust truncating quotient and remainder when their signs require flooring.

Double operations unbox, use IEEE instructions without fast-math flags, and rebox. Ordered comparisons preserve NaN behavior. Transcendentals, saturating conversions, text, formatting, and IO

call the same runtime used by the interpreters. LLVM optimization must not replace those language-level choices.

Separate fast calls from closure application

Each lifted function has two entries. The fast entry receives an environment word followed by ordinary word arguments, with `tailcc`. A top-level function receives zero for its unused environment. Known calls set the convention on the **call instruction** as well as on the callee:

```
pub(super) fn known(
    &self,
    f: FuncId,
    env: Option<IntValue<'ctx>>,
    args: &[IntValue<'ctx>],
    tail: bool,
) → Result<IntValue<'ctx>> {
    let mut actual: Vec<BasicMetadataValueEnum> =
        vec![env.unwrap_or(self.n(0)).into()];
    actual
        .extend(args.iter().map(|v| BasicMetadataValueEnum::from(*v)));
    let call = self.builder.build_call(
        self.fast[f.0 as usize],
        &actual,
        "known",
    );
    call.set_call_convention(TAIL_CC);
    call.set_tail_call(tail);
    if self
        .builder
        .get_insert_block()
        .unwrap()
        .get_parent()
        .unwrap()
        .get_call_conventions()
        ≠ TAIL_CC
    {
        call.set_tail_call_kind(
            inkwell::values::LLVMTailCallKind::LLVMTailCallKindNoTail,
        );
    }
    Ok(call.try_as_basic_value().unwrap_basic().into_int_value())
}
```

The slow entry uses the C ABI, taking the closure word and an argument-array pointer. It unpacks exactly the function’s syntactic arity and enters fast code. Closures store this slow address. Unknown calls use `rt_apply1` through `rt_apply4`, or the general `rt_apply` for wider batches. General argument arrays are allocated in the function’s entry block, not repeatedly at a control-flow site. The runtime handles partial and oversaturated applications using the existing eval/apply protocol (Marlow and Peyton Jones 2004).

The C adapter is not a guaranteed tail transfer, nor is an unknown tail call through `rt_apply`. Higher-order recursion can therefore grow the native stack. Extending the guarantee requires an LLVM-emitted apply path with a compatible calling convention. Keeping this boundary explicit is more useful than treating an optional optimizer transformation as a language guarantee.

Initialize, emit, and link

`fun_init` installs global function closures first, then evaluates strict value initializers in dependency order. The generated C `main` initializes the runtime, calls `fun_init`, applies the source `main` to `unit`, and exits through `rt_exit` to flush output. This order preserves which top-level error occurs first and ensures merely constructing an IO action performs no effects.

`native.rs` writes an object with the target machine, then invokes the system `cc` directly. It links `libfun_rt_21.a`, Boehm, and the native library flags reported by `rustc`. `FUN_RT` selects an installed archive. Otherwise it is found beside the compiler. `just build` prepares the runtime and flags first. Linking uses a temporary directory beside the destination and installs the executable only on success. There is no shell interpolation and a failed link leaves an existing output intact.

Calls crossing between `tailcc` and the C runtime are explicitly `notail`. Otherwise LLVM can infer a tail transfer across the two conventions and get stack cleanup wrong for a wide call. This is a particularly unpleasant bug because the IR can pass verification. Marking this boundary doesn’t interfere with our guaranteed tail calls between known Fun functions.

Running It

Install LLVM 22 and Boehm as described in the repository README. Set `LLVM_SYS_221_PREFIX` to the LLVM installation prefix, then build from the root:

```

$ just build
$ target/debug/fun-ch21 build crates/ch21_llvm/tests/programs/native/
acceptance.fun -o build/acceptance --lint
$ build/acceptance
5.0
3.0
6.0
[3.0,10.0]
True
5000000050000000
$ target/debug/fun-ch21 dump --pass=llvm crates/ch21_llvm/tests/programs/
native/acceptance.fun > build/acceptance.ll
$ target/debug/fun-ch21 dump --pass=llvm-opt crates/ch21_llvm/tests/programs/
native/acceptance.fun > build/acceptance-opt.ll

```

Run the examples through both native code and the interpreters. Compare errors as well as successful answers, since evaluation-order bugs often reveal themselves in which error appears first. It's also worth trying all optimization levels. A calling-convention mistake can hide until LLVM decides to make a particular transformation.

Separate stress tests execute the native recursion programs with a 512 KiB stack. Another loop allocates at least 100 million words under a 64 MiB Boehm heap limit. Together these test two different obligations: tail transfers must release call frames, and dead heap objects must remain collectible. Passing LLVM verification alone establishes neither.

Exercises

1. **Warm-up.** Trace a field load from a tagged word through `inttoptr`, `GEP`, and `load`. State which earlier check justifies the field index.
2. **Intermediate.** Draw the blocks for a join with two callers. Add division in one caller and identify the correct phi predecessor afterward.
3. **Intermediate.** Explain why adding `nsw` to integer multiplication would change `Fun`'s semantics, even if most examples still pass.
4. **Advanced.** Compare unoptimized and optimized IR for `sumTo`. Check the calling convention at both ends of its recursive transfer.
5. **Advanced.** Design an unknown-call stress test that exposes the C apply boundary. Describe the ABI change needed to make it a guaranteed tail call.

Further Reading

- Compiler Crates: `Inkwell`, for building LLVM modules through Rust.

- LLVM Language Reference, especially calls, phi nodes, integer overflow flags, and pointer arithmetic.
- LLVM tail-call implementation, for the conditions behind a bounded-stack claim.
- Join points and shared continuations (Maurer et al. 2017).
- Fast application and the eval/apply boundary (Marlow and Peyton Jones 2004).
- Representation in a strict functional implementation (Leroy 1990).

22

JIT and the REPL

New files

- `src/jit.rs`
- `src/repl.rs`
- `src/bin/fun.rs`

Changed files

- `src/llvm/mod.rs`
- `src/llvm/control.rs`
- `rt/src/heap.rs`
- `src/session.rs`
- `src/rename.rs`
- `build.rs`
- `src/core.rs`
- `src/core_type.rs`
- `src/elaborate.rs`
- `src/erase.rs`
- `src/lib.rs`
- `src/main.rs`

Motivation

An executable can run after its compiler has exited. With a JIT, we keep the compiler around and run the generated code inside its process. This gives us something rather fun to build next, a REPL that compiles the definitions we type.

LLVM already knows how to produce machine instructions. The tricky part is keeping them alive for as long as the program needs them. Enter `saved = make 40`, then call `saved 2` on the next line. That closure might hold both a captured integer and an address in the previous input’s code. We must retain both.

We’ll keep the compiler pipeline and calling convention from Chapter 21. Around them we’ll build a session that remembers accepted definitions and leaves them intact when a new input fails. A typo on the next line shouldn’t destroy the work we’ve already done.

The Language, Now

There is no new Fun syntax. The REPL accepts expressions and declarations. Explicit braces put several declarations into one input. A file loaded with `:load` can use ordinary multiline layout.

```
make x = \y → x + y

saved = make 40
```

Entering `saved 2` prints 42. A pure expression is wrapped in the Prelude’s `print`, so its type must have a `Show` instance. An `I0` expression executes its action and discards the result. Inspect a function with `:t` rather than trying to print it.

Command	Meaning
<code>:t EXPR</code>	Infer the principal qualified type. <code>:type</code> is an alias.
<code>:k TYPE</code>	Infer the kind of a constructor or type application.
<code>:core EXPR, :anf EXPR, :llvm EXPR</code>	Inspect an intermediate program without executing it.
<code>:load FILE</code>	Replace definitions after successful checking and initialization.
<code>:reset</code>	Release retained code, roots, and definitions.
<code>:engine NAME</code>	Choose <code>jit</code> , <code>core</code> , <code>anf</code> , or <code>flat</code> . Preserve definitions.
<code>:quit</code>	Leave the session. <code>:q</code> is an alias.

Loading a file does not execute its `main`. Enter `main` explicitly to run it. Duplicate user definitions are errors, as in a source file. Reset or load a replacement to change them. A compiler error leaves accepted definitions intact. Put Prelude overrides in the first declaration block or a loaded file. Later inputs cannot change the names visible to earlier definitions. A fixity must accompany its operator definition in the same input.

Theory

A closure is a pair of code and environment, even when our representation stores several captured fields (Marlow and Peyton Jones 2004). Reachability of the environment is only half the lifetime problem. The code address must remain executable too. The Boehm collector understands heap reachability, but does not own LLVM's machine code or know when an execution engine can be discarded (Boehm and Weiser 1988).

We give a session ownership of all its execution engines. Each engine owns one input's module. A separate table holds global words in registered root slots. Earlier modules can therefore remain immutable, and later modules import their values through external global declarations. Reset destroys the engines before unregistering the slots. The enclosing LLVM Context outlives both.

Typing a bad definition shouldn't destroy the good ones we've already entered. We therefore prepare a candidate separately and retain it only after checking and initialization succeed. Inspection commands stop before initialization. Asking for the type of error "x" is safe. Running it is another matter. Keeping those paths separate lets us explore programs without accidentally executing them (Pierce 2002).

Implementation

A returning entry

A native executable starts with `main`, initializes globals, executes the IO action, and exits. A REPL cannot exit after every successful input. JIT lowering instead emits a C-callable `void fun_entry(void):` initialize new globals, apply the query action to unit, then return. Fast entries still use `tailcc`. Closure calls still use the same C eval/apply path. Calls across that ABI boundary are explicitly `notail`. A wide-call regression caught LLVM inferring a tail transfer from a `tailcc` function into C apply, which corrupted the callee-pop return frame on AArch64. Verification did not catch this target-level failure. Executing the same program on both backends did. The fix also applies to Chapter 21.

`Backend::for_jit` selects this entry and external global storage. It reuses Chapter 21's worklist emission, verifier, and pass manager. We run O1 for the interactive path. Native build retains its O0-O3 choice.

```

/// A returning entry and externally rooted globals replace the native process
startup.
pub fn for_jit(
    context: &'ctx Context,
    program: &'program flat::Program,
    inherited: BTreeSet<Id>,
) → Result<Self> {
    Self::create(context, program, 1, true, inherited)
}

```

Resolve symbols before execution

The system linker resolves a native object's runtime calls. MCJIT needs the same addresses inside the compiler. `runtime_symbols` takes the addresses of the linked runtime's C exports, keeping them in the binary. `map_symbols` checks every external function and installs its address with `add_global_mapping`. Earlier fast and slow entries are mapped to their retained engines in the same step. LLVM intrinsics remain LLVM's responsibility.

```

/// Resolve declarations before MCJIT finalizes the module. Intrinsic belong
to LLVM.
fn map_symbols(
    module: &inkwell::module::Module<'_,>,
    engine: &ExecutionEngine<'_,>,
    imports: &[(String, usize)],
) → Result<(), String> {
    let symbols = runtime_symbols();
    for function in module
        .get_functions()
        .filter(|f| f.count_basic_blocks() == 0)
    {
        let name =
            function.get_name().to_str().map_err(|e| e.to_string())?;
        if name.starts_with("llvm.") {
            continue;
        }
        let address = symbols
            .iter()
            .find(|(symbol, _)| *symbol == name)
            .map(|(_, address)| *address)
            .or_else(|| {
                imports
                    .iter()
                    .find(|(symbol, _)| symbol == name)
                    .map(|(_, address)| *address)
            })
            .ok_or_else(|| {
                format!("unmapped JIT runtime symbol: {name}")
            })?;
        engine.add_global_mapping(&function, address);
    }
    Ok(())
}

```

An unrecognized import is an error before execution. We do not depend on the host executable exporting symbols by name. String operations, formatting, application, errors, and IO all reach the Chapter 22 runtime rlib, whose ABI matches the static library used by native executables.

Keep code and roots together

The session's ownership is explicit:

```

/// Keep every engine alive until no REPL binding can reference its closures.
/// Fields drop in declaration order: release code before unregistering the
slots.
pub struct Jit<'ctx> {
    engines: Vec<ExecutionEngine<'ctx>>,
    roots: BTreeMap<GlobalKey, GlobalRoot>,
    queries: Vec<GlobalRoot>,
    code: BTreeMap<GlobalKey, (usize, usize, usize)>,
    types: BTreeMap<GlobalKey, Vec<String>>,
    context: &'ctx Context,
}

```

A global slot uses the runtime's `GlobalRoot` owner. Its boxed storage stays at a stable address and is registered with Boehm. `UnsafeCell` permits generated code to write the slot without conflicting with Rust's reference rules. Mapping an LLVM global to that address makes both generated loads and the collector see the same word. A Rust container alone would not provide this guarantee. Boehm does not scan Rust's ordinary heap transitively.

The source group order distinguishes Prelude globals from user globals that shadow their spelling. Session-local renamer uniques identify LLVM declarations within a compilation, but adding declarations may change their numbers. We need something else for persistent keys. Accepted slots are reused and their initializers are skipped. Before importing a value, an alpha-normalized type fingerprint checks that rechecking has not changed its type, and the saved syntactic arity checks the fast entry's argument count. Erasure preserves this flat metadata. The fingerprint traversal uses an explicit worklist. Query slots are fresh and retained with their engines, so no surviving relocation points into freed storage.

`Jit::run` finishes all mappings before retrieving a typed callable. Its unsafe call relies on a few facts we check beforehand. The backend emits exactly the C signature requested from Inkwell. All runtime imports are resolved, and the engine and roots remain alive throughout the call. The callable is used immediately. We never add code to that engine afterward. Inkwell documents that later additions can invalidate a previously retrieved function pointer.

```

/// Enter a finalized module without exposing a callable to the rest of the
/// compiler.
fn call_entry(engine: &ExecutionEngine<'_>) → Result<(), String> {
    // SAFETY: the verified backend emits exactly void fun_entry(void), C ABI.
    // All imports are explicitly mapped; code, constants, and root slots
    remain live.
    // This engine is never modified after lookup, and no pointer escapes this
    call.
    unsafe {
        let entry = engine
            .get_function::("fun_entry")
            .map_err(|e| e.to_string())?;
        entry.call();
    }
    Ok(())
}

```

Commit definitions after success

Repl::line dispatches commands in a loop. Declarations are added to a candidate source list, checked with a private \$repl entry, and initialized before the list replaces the accepted one. The private name cannot be written in Fun and does not collide with a loaded main. A load similarly constructs a replacement session before swapping it in.

Expression execution uses ordinary language operations:

```

/// Both display and IO sequencing are ordinary, type-checked Prelude
/// operations.
fn action_query(expression: &str, ty: &crate::types::Type) → String {
    if matches!(ty, crate::types::Type::Con(id, _) if id.name == "IO") {
        format!("{expression} >> pure ()")
    } else {
        format!("print ({expression})")
    }
}

```

The resulting query passes through inference and elaboration again. The wrapper does not bypass class solving or IO abstraction. Type, kind, and IR commands stop before execution. All new traversal and command control uses loops. No larger Rust stack or stack-growth library is introduced. Recursive frontend passes inherited from earlier chapters retain their existing limits.

We recheck accepted source and lower a candidate module on each input. Existing global values and fast/slow function entries are imported, preserving both initialization and code identity. Unused

regenerated local bodies are removed by optimization, but compilation work and retained machine code grow with the session. This makes ownership easy to inspect. Incremental checking and code reclamation would require additional dependency and lifetime machinery.

NOTE Runtime errors and `exitWith` retain native process-exit semantics, including in the JIT REPL. We never unwind or jump across generated frames. Compiler errors are recoverable. If creation of the execution engine fails, the REPL selects Flat. Verifier failures are reported rather than hidden by fallback. Explicit `run --engine=jit` reports setup failure.

Running It

Build with the LLVM 22 and Boehm setup from Chapter 21:

```
$ python3 scripts/build_runtime.py 22
$ cargo build -p ch22_jit
$ target/debug/fun repl --lint
Fun: JIT ready on first input. Type :help for commands.
fun> make x = \y → x + y
fun> saved = make 40
fun> saved 2
42
fun> :t saved
saved : Int → Int
fun> :k Maybe
Maybe : * → *
fun> :reset
Reset.
fun> :quit
$ target/debug/fun run crates/ch22_jit/tests/programs/native/acceptance.fun --
engine=jit --lint
5.0
3.0
6.0
[3.0,10.0]
True
500000050000000
$ target/debug/fun dump --pass=llvm crates/ch22_jit/tests/programs/jit/
returned-function.fun > build/jit-example.ll
```

`fun` and `fun-ch22` share one driver. The CLI keeps Flat as its default run engine. The REPL defaults to JIT. `--fuel` and the test-only `--print-value` remain interpreter facilities. Redirected REPL input omits prompts and the banner so transcripts can be compared byte for byte.

The JIT should give the same answers as the interpreters and standalone executables. It also needs to survive long known-call recursion and allocation under a bounded heap. Known calls keep Chapter 21’s tail-call guarantee. Calls through C apply still have the limitation we discussed there.

Try defining a closure, entering more definitions, and then calling the old closure again. The lifetime tests push this further by forcing collections between modules. A value remaining in scope isn’t enough on its own. Both the captured data and the machine code it points to must still be there.

Exercises

1. **Warm-up.** Trace the owners of `saved = make 40` across two inputs. Identify which owner protects its environment and which protects its code address.
2. **Intermediate.** Remove one runtime mapping and inspect the failure. Explain why it must be detected before calling the generated entry.
3. **Intermediate.** Add a regression where a user definition shadows a Prelude name. Show why spelling alone cannot identify its persistent slot.
4. **Advanced.** Design an isolated worker protocol that recovers from runtime errors while retaining accepted definitions. Specify what must be replayed.
5. **Advanced.** Propose a module-reclamation scheme. Account for closures inside constructors, partial applications, global values, and captured environments.

Further Reading

- Inkwell ExecutionEngine documentation, particularly symbol mappings and the safety contract of `get_function`.
- Compiler Crates: Inkwell, for LLVM construction through the Rust bindings.
- Closures, partial applications, and slow entries (Marlow and Peyton Jones 2004).
- Conservative roots and cooperation with a runtime (Boehm and Weiser 1988).
- Typing and the boundary before evaluation (Pierce 2002).

23

Optimization

New files

- `src/rep.rs`
- `src/reach.rs`
- `src/dump.rs`

Changed files

- `src/erase.rs`
- `src/llvm/values.rs`
- `src/llvm/primitives.rs`
- `src/llvm/control.rs`
- `src/llvm/mod.rs`
- `src/anf.rs`
- `src/anf_simplify.rs`
- `src/anf_lint.rs`
- `src/closure.rs`
- `src/flat.rs`
- `src/flat_lint.rs`
- `src/primitives.rs`
- `rt/src/primitives.rs`
- `rt/src/abi.rs`
- `src/main.rs`
- `src/repl.rs`
- `src/jit.rs`
- `src/lib.rs`

- `rt/src/lib.rs`

Motivation

Getting the right answer is a good start for an optimizer. We'd also like it to avoid doing the same calculation twice or allocating a fresh box between every pair of arithmetic instructions. A faster-looking expression can turn out to be a rather expensive way to get the same result.

We'll make a few focused improvements and inspect what they actually change. Traces and machine steps help us catch duplicated or reordered work. A dependency graph tells us which globals we still need. Type information tells codegen when it can use native arithmetic instead of repeatedly boxing intermediate values.

These are small optimizations, but they expose the questions a larger optimizer must answer too. What do we know about this expression? What does the rewrite save? And what happens to evaluation order?

The Language, Now

The only new source function is `trace :: String → a → a`. It writes its first argument and a newline to `stderr`, then returns its second argument unchanged. Arguments remain strict. An error while evaluating the second argument happens before the trace is printed. `trace` is a diagnostic facility, not an alternative to IO for ordinary program effects.

```
-- stderr: exact
f x y = x + y

main = print (f (trace "left" 20) (trace "right" 22))
```

This prints `left` followed by `right` on `stderr`, and `42` on `stdout`. Moving a call across the other argument would now be visible. The runtime flushes `stdout` before writing a trace, preserving the ordering of preceding output.

Theory

Preserve Work

In a strict language, `let x = f y in x + x` evaluates `f y` once. Substituting `f y` for both occurrences of `x` would evaluate it twice. That changes cost and may repeat a trace. A failing expression is observable too. An unused division by zero cannot simply disappear.

Our simplifier can substitute atoms because reading an atom does no work. It can remove an unused total computation. It can move an immediately applied lambda body into its sole call site, preserving the already evaluated arguments. It cannot use that rule for a lambda called twice. These

restrictions make the small transformation understandable without a global cost analysis (Appel 1992).

We can ask the ANF machine to count its steps with simplification switched on and off. The results, errors, and traces must agree, and the simplified program must take no more steps. This gives us a useful way to catch a rewrite that accidentally duplicates work. It doesn't prove a wall-clock speedup, but it tests a claim we can actually measure without depending on a particular CPU.

Keep Reachable Globals

A global reference gives us an edge in a dependency graph. Starting from `main`, a worklist follows those edges and retains a predecessor for each newly reached global. This resembles Chapter 9's dependency analysis, but answers a different question. Which definitions can this execution need?

Strict initialization adds roots. An unused initializer may print a trace, fail, or diverge before `main`. We retain any initializer whose harmlessness the local classifier cannot establish. Calls and control-flow initializers remain roots. Allocating a closure does not execute its body, so an unused function can be removed. Allocation failure is outside the source cost model.

The surviving globals stay in their original dependency order. In the REPL, all visible bindings are roots because later input may refer to any of them. The witness dump gives one discovery edge for each retained binding, making pruning decisions inspectable.

Keep Type Evidence, Choose Storage Locally

An erased expression no longer carries a System F type, but the compiler can retain a small fact established by that type:

```
#[derive(Clone, Copy, Debug, PartialEq, Eq)]
pub enum Rep {
    Int,
    Double,
    Word,
}
```

`Int` and `Double` establish numeric types. `Word` covers everything else, including type variables. This is evidence, not a storage layout. An `Int` returned by a call still arrives as a tagged word. Code generation decides when to decode it into a register. Calls, returns, fields, captures, globals, and join parameters retain the uniform ABI from Chapter 20 (Leroy 1990).

Integer register operations need one extra step. An LLVM i64 has 64 bits, while Fun's signed integer has 63. Adding 2^{61} to itself must produce -2^{62} . Without normalization, a following comparison would see a positive number, and division by two would produce the wrong sign. After arithmetic

we shift left once, then arithmetically right once. This sign-extends bit 62 and restores Fun's range before any later use. Adding 2^{62} to itself instead wraps to zero, not a negative number.

Double arithmetic can keep intermediate results in floating-point registers. Passing the result to another function still requires a box because our calling convention passes words. So a primitive Double accumulation loop still boxes its accumulator once per iteration at the tail call. The improvement is that it doesn't box every arithmetic intermediate along the way.

We'll check that distinction by counting allocations. Overloaded source operators can still make dictionary calls, so they may not produce this direct primitive loop. Specializing those dictionaries is a separate opportunity.

Implementation

The shared runtime owns trace behavior. Both interpreters and the native ABI use it, so even an arbitrary closure can be the returned value:

```
/// The trace event happens after both strict operands have been evaluated.
pub fn trace(message: &str) → Result<(), Failure> {
    init_io();
    use std::io::Write;
    std::io::stdout().flush().map_err(|e| e.to_string())?;
    writeln!(std::io::stderr().lock(), "{message}")
        .map_err(|e| e.to_string())?;
    Ok(())
}
```

The simplifier states what its rewrites are allowed to do beside the code.

```
//! A rewrite may delete an unused total RHS, replace an atom by an equal
atom,
//! or move a lambda applied exactly once. It may never copy a call, a failing
//! primitive, or an effect, nor move one past another. Allocation failure is
//! outside the source cost model. The oracle checks transitions and trace
order.
```

`reach::analyze` performs a stable graph traversal. Its roots include observable initializers, and the first edge to a global becomes its witness:

```

/// Follow dependencies from explicit roots and observable initializers in
stable order.
pub fn analyze(program: &Program, roots: &[Id]) → Reach {
    let bodies = program
        .globals
        .iter()
        .map(|(id, body)| (id.clone(), body))
        .collect::<BTreeMap<_, _>>();
    let mut witness = BTreeMap::new();
    let mut todo = VecDeque::new();
    for id in roots.iter().cloned().chain(
        program
            .globals
            .iter()
            .filter(|(_, b)| !discardable(b))
            .map(|(v, _)| v.clone()),
    ) {
        if bodies.contains_key(&id) && !witness.contains_key(&id) {
            witness.insert(id.clone(), None);
            todo.push_back(id);
        }
    }
    while let Some(id) = todo.pop_front() {
        for next in references(bodies[&id]) {
            if bodies.contains_key(&next) && !witness.contains_key(&next) {
                witness.insert(next.clone(), Some(id.clone()));
                todo.push_back(next);
            }
        }
    }
    Reach {
        kept: program
            .globals
            .iter()
            .filter(|(v, _)| witness.contains_key(v))
            .map(|(v, _)| v.clone())
            .collect(),
        witness,
    }
}

```

Erasure classifies types while they are still available:

```

/// This is the single classification point. A type variable proves no numeric
representation.
pub(crate) fn representation(ty: &core::Type) → Rep {
    match ty {
        core::Type::Int ⇒ Rep::Int,
        core::Type::Double ⇒ Rep::Double,
        _ ⇒ Rep::Word,
    }
}

```

An explicit postorder worklist records binder and expression-result evidence. ANF copies it to fresh temporaries, including the result of each application in an oversaturated call. Closure conversion preserves it. Missing or contradictory numeric evidence is an internal lint error. The interpreters ignore this table. IR dumps show numeric annotations so the reader can inspect the facts themselves.

Within LLVM emission, a separate enum describes current storage:

```

/// Storage is local to codegen. All ABI boundaries still carry tagged words.
#[derive(Clone, Copy)]
pub(super) enum Loc<'ctx> {
    Word(IntValue<'ctx>),
    Int(IntValue<'ctx>),
    Double(inkwell::values::FloatValue<'ctx>),
}

```

atom_word handles ABI uses. Integer and Double primitive operands request native storage instead. A tagged input is decoded once for that use, while a result already in a native register can flow directly into the next primitive. Integer arithmetic calls the normalization helper before storing its native result:

```

fn normalize(&self, value: IntValue<'ctx>) → Result<IntValue<'ctx>> {
    let bits =
        self.builder
            .build_left_shift(value, self.n(1), "wrap.bits"?;
    Ok(self.builder.build_right_shift(
        bits,
        self.n(1),
        true,
        "wrap.int",
    )?)
}

```

To isolate this codegen change, we construct Flat loops that use primitives directly and count runtime calls in the optimized LLVM. The integer loop needs none. The Double loop needs one atomic allocation for the value crossing its call boundary. We separately compare overloaded source programs with the interpreter. That keeps us honest about which improvement we've implemented.

Running It

From the repository root:

```
$ cd crates/ch23_optimize
$ cargo run -q -- run tests/programs/optimize/trace-sharing.fun --engine=anf
--print-steps
once
42
steps: 92
$ cargo run -q -- run tests/programs/optimize/trace-sharing.fun --engine=anf
--print-steps --no-simplify
once
42
steps: 171
$ cargo run -q -- dump --pass=reach tests/programs/optimize/trace-sharing.fun
```

The last command lists the retained definitions and their witnesses, ending with the number kept and dropped. Use `dump --pass=anf` or `dump --pass=flat` to inspect numeric binder annotations. To examine the actual optimized source loop:

```
$ cargo run -q -- dump --pass=llvm-opt tests/programs/optimize/double-loop.fun
```

Follow the dictionary calls as well as the local floating-point instructions. Comparing only the number of instructions in the source function can hide work inside its callees. `cargo test --test optimization` runs the work oracle and trace corpus. `cargo test --test native` includes the allocation-count contract.

Exercises

1. **Warm-up.** Predict the trace order when the second argument to `trace` is another traced expression. Confirm it on all engines.
2. **Intermediate.** Extend the reachability dump to show the complete path from a root rather than one predecessor.
3. **Intermediate.** Add `Bool` evidence and keep comparison results untagged until they cross a word boundary.

4. **Advanced.** Carry native integers through join parameters. State the new storage invariant and extend lint before changing code generation.
5. **Advanced.** Extend the work oracle with a counter for dead stores. Construct a rewrite that preserves transition counts but performs an unnecessary store.

Further Reading

- Transformation and cost reasoning in a compiler (Appel 1992).
- Value representation in a strict functional implementation (Leroy 1990).

24

A Precise Garbage Collector

New files

- `rt/src/copying.rs`
- `src/liveness.rs`
- `src/llvm/roots.rs`

Changed files

- `src/llvm/control.rs`
- `src/llvm/values.rs`
- `rt/src/heap.rs`
- `rt/src/apply.rs`
- `src/llvm/mod.rs`
- `src/llvm/primitives.rs`
- `src/native.rs`
- `src/main.rs`
- `rt/src/lib.rs`
- `src/lib.rs`

Motivation

Boehm has let us allocate objects without telling the collector where every live pointer is. That was useful while we were building the language. Now our compiler knows which values may be used again, and our runtime knows which fields contain them. We have enough information to build a precise collector of our own.

We'll use a copying collector. Live objects move into fresh space, and the old space becomes available for reuse. The idea is wonderfully compact. The catch is that every reference to a moved object has to follow it.

Suppose a closure captures a list. Updating the closure's field after moving the list won't help a caller that still holds the old address in a register. We'll need cooperation between the generated code and the runtime to keep those references current (Henderson 2002). You can stay with Boehm and skip this chapter, but building the collector is a great way to see the whole memory story come together.

The Language, Now

There is no new Fun syntax. Select the collector when building an executable:

```
$ cd crates/ch24_gc
$ cargo run -q -- build tests/programs/gc/partial.fun \
  -o ../../build/partial --gc=copying --lint
$ FUN_GC_STRESS=1 ../../build/partial
100
αβγ
```

```
add x y z = x ◇ y ◇ z

main = do
  let f = add "α"
  let g = f "β"
  print (length (map show [1..100]))
  putStrLn (g "γ")
```

The ordinary `--gc=boehm` mode remains the default. Interpreters retain their Rust representations, and the JIT retains its Boehm roots. Copying collection is a native executable option. Its heap belongs to one thread, matching Fun's lack of concurrency. Selecting a collector happens before global initialization.

Theory

Copy the Reachable Graph

Split the heap into two regions. Allocation advances a pointer in the active region, called *from-space*. At collection, copy objects reachable from roots into *to-space*. Replace each evacuated object's old header with a forwarding marker and its first payload word with the new address. A second reference to that object follows the forwarding address, preserving sharing and cycles.

Here is the beautiful part of Cheney’s algorithm. The copied objects themselves form the work queue, so we don’t need a separate stack to walk the graph (Cheney 1970). Two pointers are enough to track our progress.



Everything before `scan` has been inspected. Between `scan` and `free` are objects we’ve copied but haven’t inspected yet. When we scan one, its fields may lead us to more objects, which we append at `free`. Eventually `scan` catches up. We’ve found the whole live graph!

Now swap the spaces and resume allocation after `free`. Sharing and cycles are handled by forwarding addresses, and the scan itself never recurses through Rust’s call stack.

Object kinds tell the collector what to inspect. Constructor payloads contain values. Closures begin with an untraced code address and arity, followed by captures. PAPs contain a closure, a raw argument count, and arguments. Doubles and strings contain no pointers. Nullary constructors and integers have their low bit set, so the tracer skips them. Static string addresses lie outside from-space and remain stationary.

Make Roots Writable

A *shadow stack* is an explicit stack of root ranges, maintained alongside the machine stack. Each generated function owns an array of tagged-word slots. Native integer and floating-point intermediates need no root. A tagged result that might be a pointer is stored in a slot, and subsequent uses reload it. Collection changes the slot, so a reload sees the new address.

Arguments need the same care. Boxing a later `Double` argument may move an earlier argument. Codegen first saves converted arguments in scratch roots, then reloads all of them for the call. Constructor allocation protects its fields across the allocator call before storing their relocated addresses in the new object. Rust-owned argument buffers use the same registration contract. Their API returns copied words or raw ABI pointers, never a shared Rust slice into cells that collection may update.

A returning function removes its frame. A known tail call removes the frame *before* entering its callee. There is no allocation between the final argument loads and the callee’s registration of its parameters. Thus tail recursion does not accumulate shadow frames. Unknown calls still have the eval/apply limitation from Chapter 21.

Retain Only Live Values

Keeping every value until return would be safe for addresses, but could retain a large dead structure inside a loop. Backward liveness determines which bindings may be read again. Each Flat node contributes its operands. Its successors contribute their live variables, excluding bindings defined on that edge. A jump's successor is its join body, with the join parameters removed.

The equations are monotone. Begin with empty sets and revisit predecessors when a successor changes. The least fixed point handles join back-edges as well as ordinary branches. Before emitting a computation, codegen clears slots for dead bindings and clears the scratch range. Liveness is conservative about control flow, but every nonzero traced slot contains a tagged value, never an unboxed numeric intermediate.

Implementation

`liveness.rs` builds its control-flow graph with an explicit worklist. The solver is another worklist, so source nesting does not become Rust recursion:

```
let mut queue = nodes.keys().copied().collect::<VecDeque<_>>();
let mut queued = nodes.keys().copied().collect::<BTreeSet<_>>();
while let Some(id) = queue.pop_front() {
    queued.remove(&id);
    let n = &nodes[&id];
    let mut live = n.uses.clone();
    for (next, defined) in &n.edges {
        live.extend(result.before[next].difference(defined).cloned());
    }
    if result.before[&id] != live {
        result.before.insert(id, live);
        for p in predecessors.get(&id).into_iter().flatten() {
            if queued.insert(*p) {
                queue.push_back(*p);
            }
        }
    }
}
```

`llvm/roots.rs` assigns root slots and changes pointer-bearing locations from SSA words into reloadable slots. These addresses remain stable for the activation:

```

pub(super) fn root_binding(
    &self,
    v: &Id,
    loc: Loc<'ctx>,
) → Result<Loc<'ctx>> {
    let Some(frame) = &self.frame else {
        return Ok(loc);
    };
    if !matches!(loc, Loc::Word(_) | Loc::Root(_)) {
        return Ok(loc);
    }
    let p = self.slot(frame.array, frame.slots[v])?;
    let w = self.loc_word(loc)?;
    self.builder.build_store(p, w)?;
    Ok(Loc::Root(p))
}

```

The runtime's evacuation step either preserves a stationary word, follows a forwarding address, or copies one object. The header's size includes only payload words, so the allocation also reserves its header:

```

fn evacuate(
    from: &mut [Word],
    used: usize,
    to: &mut [Word],
    free: &mut usize,
    w: Word,
) → Word {
    let base = from.as_ptr() as usize;
    let address = w as usize;
    // Immediates, null placeholders, and static string/code addresses never
    move.
    if layout::is_immediate(w)
        || address < base
        || address ≥ base + used * 8
    {
        return w;
    }
    assert_eq!((address - base) % 8, 0);
    let start = (address - base) / 8;
    let header = from[start];
    if layout::kind(header) == layout::FORWARD {
        return from[start + 1];
    }
    let words = 1 + layout::size(header);
    assert!(words ≥ 2 && start + words ≤ used);
    let dest = *free;
    to[dest..dest + words].copy_from_slice(&from[start..start + words]);
    *free += words;
    let pointer = to.as_mut_ptr().wrapping_add(dest) as Word;
    from[start] = layout::header(layout::FORWARD, 0, 1);
    from[start + 1] = pointer;
    pointer
}

```

The collector scans roots, then consumes the queue in to-space. The heap grows when necessary, allocating the destination before moving any object. Existing addresses are never invalidated by resizing active from-space.

```

fn collect(&mut self, reserve: usize) {
    let capacity = if self.used + reserve > self.from.len() {
        self.from
            .len()
    }
}

```

```

        .saturating_mul(2)
        .max(self.used.checked_add(reserve).expect("heap overflow"))
    } else {
        self.from.len()
    };
    self.spare.resize(capacity, 0);
    let mut free = 0;
    for (&address, &len) in &self.roots {
        for i in 0..len {
            let slot = (address as *mut Word).wrapping_add(i);
            // SAFETY: registered ranges remain writable throughout
collection.
            unsafe {
                slot.write(evacuate(
                    &mut self.from,
                    self.used,
                    &mut self.spare,
                    &mut free,
                    slot.read(),
                ));
            }
        }
    }
    let mut scan = 0;
    while scan < free {
        let header = self.spare[scan];
        let size = layout::size(header);
        let first = match layout::kind(header) {
            layout::CON => 0,
            layout::CLOSURE => 2, // code and arity are raw, not values
            layout::PAP => 0,
            layout::DOUBLE | layout::STRING => size,
            _ => panic!("invalid object in to-space"),
        };
        for field in first..size {
            if layout::kind(header) == layout::PAP && field == 1 {
                continue;
            }
            let slot = scan + 1 + field;
            let value = self.spare[slot];
            self.spare[slot] = evacuate(
                &mut self.from,
                self.used,

```

```

        &mut self.spare,
        &mut free,
        value,
    );
}
scan += size + 1;
}
std::mem::swap(&mut self.from, &mut self.spare);
self.used = free;
self.collections += 1;
self.copied_words += free;
}

```

We can make stale-pointer bugs much easier to reproduce by setting `FUN_GC_STRESS=1`, which collects before every allocation. A closure or PAP must survive even if collection happens between the steps that construct it. Shared references must still point to one object, cycles must remain cycles, and raw Double bits must never be mistaken for a field to trace.

Running It

```

$ cd crates/ch24_gc
$ cargo run -q -- dump --pass=llvm tests/programs/gc/partial.fun --gc=copying
; ... root arrays, rt_roots_push, reloads, rt_roots_pop ...
$ cargo test -p fun-rt-24 --test copying
$ cargo test --test copying

```

The dump shows the root protocol before LLVM optimization. Stress collection is intentionally expensive. Repeatedly copying a growing live list takes quadratic work. Use it to expose stale addresses, then measure normal collection separately. Neither backend promises recovery from operating-system allocation failure.

Native copying executables link a separate copying-only runtime archive. That build excludes the Boehm imports and link flag. The ordinary runtime `rlib` still supports the JIT's Boehm heap. `just build` prepares both archives and records each one's native link flags. A test checks the executable's dynamic dependencies as well as its output under collection stress.

Heap Growth

Correct tracing is not enough. If almost all of a semispace remains live, each collection can copy a large graph while recovering only a few words. After a capacity-triggered collection we grow the space when live data and the pending allocation would occupy more than half of it. Stress collection still runs on every allocation, but does not itself force growth. A regression test holds seven eighths

of a space live and bounds the total words copied during churn. The bound measures collector work without relying on a machine-specific timer.

Exercises

1. **Warm-up.** Trace two roots that point to the same object. Explain why only one copy is made and where each root ends up.
2. **Intermediate.** Draw a closure/PAP graph and mark every payload word the collector must skip. Explain why low-bit testing alone is insufficient for raw metadata and Double bits.
3. **Intermediate.** Give a join back-edge that needs more than one liveness iteration. Show the sets as they grow.
4. **Advanced.** Reuse root slots whose live ranges do not overlap. Preserve the existing stress tests and compare generated frame sizes.
5. **Advanced.** Design a generational extension. Identify the write barrier required by recursive closure patching before implementing it.

Further Reading

Cheney supplies the nonrecursive copying algorithm (Cheney 1970). Henderson explains explicit roots in an otherwise uncooperative compilation environment (Henderson 2002). The handbook puts copying, generations, and concurrent collection in context (Jones et al. 2023). Compare the root contract with Boehm and Weiser’s conservative approach (Boehm and Weiser 1988).

25

Future Work

We started with a calculator. Now we have a language with inferred types, algebraic data, records, type classes, and a compiler that produces native code. We've even built an optimizer and a garbage collector. That's quite a journey!

The best part is that the compiler is now yours to experiment with. You know where its decisions are made, and you have smaller chapter versions to return to when you want to try a different approach. Here are a few directions that might be fun to explore next.

Where Next?

Explicit Delay and Laziness

Sometimes we really do want to describe a computation and leave it for later. I'd explore that through explicit delay, using call-by-push-value to distinguish values from computations (Levy 2001). A programmer could then see where a computation is suspended and where it is forced.

Sharing the result is another choice. Call-by-need needs a way to remember that a suspension has already been evaluated, as well as a way to handle recursive forcing. Our closure representation is a useful starting point, though adding a thunk object alone won't do all of that work. An STG-like intermediate language would give these operations a place to live.

Modules and Separate Compilation

As your programs grow, you'll probably get tired of putting everything in one file. Modules would let us group definitions, expose an interface, and hide the rest. Separate compilation would let us rebuild one part without compiling the whole program again.

We already have unique names within a compilation. The next step is deciding which identities and type information must survive between compilations. Classes make this especially interesting. An imported function may depend on instances, so we need a clear rule for which instances it sees. A small module system with explicit interfaces would be a good first experiment.

Rows, Effects, and Classes

Our records have names and fixed fields. What if a function could accept any record with an `x` field? Row polymorphism is one way to express that. Effect rows open another line of exploration, letting types describe which effects a computation may perform.

These ideas would take us beyond the constraint language we've built, but the basic habit still applies. Find a small program you'd like to express, write down its intended type, and work out what information inference needs.

You could also explore multi-parameter classes, functional dependencies, or type families. Our Haskell 98 restrictions keep instance search manageable (S. Marlow 2010). Relaxing one at a time gives you a chance to understand what it was buying us before the next complication arrives.

Complete Tail Calls

Known calls already use LLVM's tail calling convention. Unknown calls pass through the C runtime, where a chain of calls can still accumulate stack frames. It would be satisfying to close that gap.

One route is to emit apply dispatch in LLVM too. Exact calls, PAP entry, and oversaturation could then share a tail-call protocol. Oversaturation is the interesting case. We must keep the leftover arguments somewhere while the first call returns a function to receive them. The eval/apply model gives us a place to start (Marlow and Peyton Jones 2004).

Specialization and Unboxed Values

We can keep arithmetic intermediates in native registers, but calls still pass uniform words. When the compiler knows the concrete types at a call, it could create a specialized version with unboxed arguments. Knowing a dictionary could also turn a method selection into a direct call.

There is plenty of room to experiment here. There is also plenty of room to make the executable enormous. Specialized copies cost code space, so a useful experiment should measure both the work saved and the code added. Start with one small numeric function and inspect the result before building a general specializer.

Optimization and Debugging

Inlining can expose a constant, a constructor, or a dictionary method that the next pass can simplify. It can also duplicate work. The traces and machine-step counts from Chapter 23 give us a way to investigate that tradeoff. An optimization should earn its place by helping programs we can actually run.

A source debugger would be another rewarding project. Imagine stepping through Fun code while the native executable runs underneath. DWARF can connect machine instructions to source locations, but we'd also need to describe scopes and variable locations. Start without optimization. Explaining where a variable went after the optimizer removed it is a problem we can enjoy later.

Pick a Starting Point

You don't have to rebuild the whole compiler to try one of these ideas. Each has a natural place to begin.

Direction	Where to Begin
Delay	Core evaluation and closure entry
Modules	Renamed identities and compilation-unit metadata
Rows or effects	Types, constraints, and elaboration
Class extensions	Instance checking and constraint solving
Unknown-call TCO	Flat calls, slow entries, and runtime apply
Specialization	Type evidence before erasure and calling conventions
Debugging	Source spans through Flat and LLVM emission

The IR Reference and Runtime ABI are there when you need to look up the details.

Congratulations on making it through! Building a compiler brings together a remarkable range of ideas, and you've followed them from source text all the way to a running program. I hope some of the machinery that once looked intimidating now looks like something you could sit down and build.

Functional programming has advanced a great deal since Haskell 2010, and we've only scratched the surface. There are beautiful ideas left to discover in its languages, type systems, and implementations. Pick one that makes you curious and try it. This is only the beginning of your compiler journey.

Appendices

A

Surface Syntax

This grammar collects the final Fun language in one place. It describes source syntax after layout has inserted braces and semicolons. Name resolution, kinds, and typing impose the restrictions listed below. See Appendix B for the corresponding judgments.

Reading the Grammar

Quoted strings are terminals. `|` separates alternatives, `[ X ]` means optional, and `{ X }` means zero or more repetitions. Parentheses group grammar expressions. Quoted brackets and braces are literal tokens. `varid`, `conid`, and literal tokens are defined in the lexical section. An unbraced source file receives an outer layout block, so both styles have the same `program` production.

Declarations

```

program      ::= '{' [ topdecl { ';' topdecl } ] [ ';' ] '}'
topdecl      ::= dataddecl | typeddecl | classdecl | instdecl
              | fixity | sig | bind

dataddecl    ::= 'data' tycon { tyvar } '=' constr
              { '|' constr } [ deriving ]
constr        ::= con { atype }
              | con '{' fielddecl { ',' fielddecl } '}'
fielddecl     ::= varid { ',' varid } '::' type
deriving      ::= 'deriving' conid
              | 'deriving' '(' conid { ',' conid } ')'
typeddecl     ::= 'type' conid { tyvar } '=' type

classdecl     ::= 'class' [ context '⇒' ] conid tyvar
              [ 'where' '{' [ member { ';' member } ] [ ';' ] '}' ]
member        ::= sig | bind
instdecl      ::= 'instance' [ context '⇒' ] conid insthead
              [ 'where' '{' [ bind { ';' bind } ] [ ';' ] '}' ]
insthead      ::= conid | '(' conid { tyvar } ')'
              | '[' tyvar ']' | '(' ' )'
              | '(' tyvar ',' tyvar [ ',' tyvar ] ')'
fixity        ::= ('infixl' | 'infixr' | 'infix') digit op { ',' op }
sig           ::= var { ',' var } '::' qualtype

bind          ::= funlhs rhs | pat rhs
funlhs        ::= var { apat } | apat varop apat
rhs           ::= '=' expr [ 'where' decls ]
              | guard '=' expr { guard '=' expr } [ 'where' decls ]
guard         ::= '|' expr
decls         ::= '{' [ localdecl { ';' localdecl } ] [ ';' ] '}'
localdecl     ::= sig | bind

```

A record declaration has exactly one constructor. Its labels are distinct, but other record types may reuse them. Instance-head variables are distinct and the context mentions only those variables. Fixity declarations occur only at the top level, with precedence 0 through 9. The only deriving classes are Eq, Ord, and Show. Pattern bindings are nonrecursive. A recursive binding must become a function after desugaring.

Expressions

```

expr      ::= infixexp [ '::' qualtype ]
infixexp  ::= lexp { op lexp } | '-' infixexp
lexp      ::= '\' apat { apat } '→' expr
           | 'let' decls 'in' expr
           | 'if' expr 'then' expr 'else' expr
           | 'case' expr 'of' '{' alt { ';' alt } [ ';' ] '}'
           | 'do' '{' stmt { ';' stmt } [ ';' ] '}'
           | fexp
fexp      ::= aexp { aexp }
aexp      ::= atom { projection | update }
atom      ::= var | con | literal
           | '(' expr ')'
           | '(' expr ',' expr [ ',' expr ] ')'
           | '[' [ expr { ',' expr } ] ']'
           | '[' expr '..' expr ']'
           | '(' infixexp op ')' | '(' op infixexp ')'
           | '(' PROJ_SECTION varid ')'
           | conid '{' fbind { ',' fbind } '}'
projection ::= PROJ varid
update     ::= '{' fbind { ',' fbind } '}'
fbind      ::= varid '=' expr
alt        ::= pat ( '→' expr | guard '→' expr
                  { guard '→' expr } ) [ 'where' decls ]
stmt       ::= expr | pat '←' expr | 'let' decls

var        ::= varid | '(' varsym ')'
con        ::= conid | '(' ':' ')' | '[' ']' | '(' ' )'
           | '(' ',' ')' | '(' ' ',' ',' ' ')'
op         ::= varsym | ``' varid ``' | ':'
varop      ::= varsym | ``' varid ``'

```

A block's last `do` statement must be an expression. A bare constructor followed by a record brace is construction, not update. Construction requires every field exactly once. Updates preserve the record's type. Tuples have arity two or three. `(- e)` is negation rather than a right section. `(&&)` and `(||)` are accepted as values, but these operators cannot be redefined.

Projection and update bind tighter than application. Thus `f r.x` means `f (r.x)`. Application associates left. Arrow types associate right. An operator chain is resolved using the declarations in scope, with `infixl 9` as the default. Unary minus follows Haskell's precedence-six rule. Conflicting associativities at equal precedence require parentheses.

The Prelude supplies these fixities:

Associativity	Prece- dence	Operators
Right	9	.
Left	7	*, /, `div`, `mod`, `quot`, `rem`
Left	6	+, -
Right	6	$\diamond$
Right	5	:, ++
Nonassocia- tive	4	=, $\neq$, <, $\leq$, >, $\geq$
Left	4	<\$>, <*>
Right	3	&&
Right	2	\\
Left	1	$\gg$, >>
Right	0	\$

Patterns and Types

```

pat      ::= lpat [ ':' pat ]
lpat     ::= conid apat { apat } | '-' intlitt | apat
apat     ::= varid [ '@' apat ] | '_' | con
           | conid '{' [ fpat { ',' fpat } ] '}'
           | literal | '(' pat ')'
           | '(' pat ',' pat [ ',' pat ] ')'
           | '[' [ pat { ',' pat } ] ']'
fpat     ::= varid '=' pat

qualtype ::= [ context '⇒' ] type
context  ::= pred | '(' pred { ',' pred } ')'
pred     ::= conid tyvar
type     ::= btype [ '→' type ]
btype    ::= atype { atype }
atype    ::= tycon | tyvar | '(' type ')'
           | '(' type ',' type [ ',' type ] ')'
           | '[' type ']'
tycon    ::= conid | '[' ']' | '(' ' )'
           | '(' ' ',' ' )' | '(' ' ',' ',' ' )' | '(' '→' ' )'
tyvar    ::= varid

```

Pattern variables are distinct within a pattern vector. Omitted record fields mean wildcards. Constructors in patterns must be saturated. Type synonyms must be fully applied and acyclic. There is no explicit `forall`, kind annotation, lazy pattern, qualified name, or user-defined infix constructor.

Lexical Structure and Layout

Identifiers are ASCII: `varid` matches `[a-z_][A-Za-z0-9_]*` and `conid` matches `[A-Z][A-Za-z0-9_]*`. `_` alone is a wildcard. Reserved words are:

```

case class data deriving do else if in infix infixl infixr
instance let of then type where

```

An operator is a maximal run of the following symbol characters, excluding the reserved operators below. User-defined operators cannot begin with `..`.

```

symbol characters: ! # $ % & * + . / < = > ? @ \ ^ | ~ : -
reserved operators: .. : :: = \ | ← → @ ⇒

```

Integer literals are decimal or `0x` hexadecimal, with underscore separators. Double literals contain a decimal point or an exponent, as in `1.5`, `1e10`, and `2.5e-3`. Signs are syntax outside the literal token. Character literals contain one Unicode scalar in single quotes. Strings contain Unicode text in double quotes. Both support `\n`, `\t`, `\\`, `\'`, `\"`, `\0`, and `\u{...}` escapes. `literal` ranges over integer, double, character, and string tokens.

Line comments start with `--` followed by a nonsymbol character. `→` is an operator. Block comments use `{-` and `-}` and may nest.

Before layout, adjacency distinguishes projection dots. A dot immediately after a variable, closing delimiter, or literal, and immediately before a `varid`, is `PROJ`. Immediately after `(` and before a `varid`, it is `PROJ_SECTION`. Otherwise it is an ordinary operator. Consequently `f.g` projects a field, while `f . g` composes functions.

Layout inserts braces after `let`, `where`, `of`, and `do`, and around an unbraced file. Equal indentation separates declarations or statements. Dedentation closes blocks. Explicit braces take precedence, and `in` closes an implicit `let` block even on the same line. Chapter 8 shows the inserted tokens directly. The Haskell Report gives the underlying lexical and layout conventions (S. Marlow 2010).

B

Typing Rules

These rules give a declarative account of Fun’s source type system. They collect the judgments used throughout the book, including patterns, binding groups, classes, records, and IO. They describe which programs are well typed. The implementation finds those derivations using unification, levels, and constraint solving. Core has its own explicitly polymorphic typing judgment, introduced in Chapter 10.

How to Read a Rule

Read a fraction from bottom to top as a question: to establish the conclusion, establish each premise above the line. Read it from top to bottom as a proof: these premises justify that conclusion. A rule with no premises is an axiom. The name in parentheses identifies the step, not an extra condition (Pierce 2002).

Our main judgment is $P; \Gamma \vdash e : \tau$. It says that expression e has type τ when the variables in Γ have their listed schemes and the class predicates in P hold. The semicolon separates environments. It has no meaning in Fun source. For example:

$$\frac{x : \text{Int} \in \Gamma}{P; \Gamma \vdash x : \text{Int}} \text{ (Var-Int)}$$

The typing environment is a finite map. Extending it with $x : \tau$ shadows an outer binding of x . A bar means a sequence, so $\bar{\alpha}$ stands for zero or more type variables. Rules with a subscript i apply to every member of the indicated sequence. Side conditions written below a rule are part of it.

Notation	Meaning
τ	A monotype: variable, constructor, application, or arrow
σ	A scheme $\forall \bar{\alpha}. Q \Rightarrow \tau$
P, Q	Sets of class predicates, such as $Eq\ \alpha$
Γ	Term variables and their schemes
Δ	Type variables and constructors with their kinds
$S\tau$	Capture-avoiding substitution in a type
$fv(X)$	Free type variables of X
$P \vdash Q$	The givens P entail every predicate in Q
Γ_p	Monomorphic variables introduced by a pattern

Types are identified after expansion of fully applied synonyms. Schemes are identified up to renaming of bound variables. The global signature records data constructors, record fields, classes, and instances. All rules below assume that signature has passed the declaration checks.

Kinds and Type Formation

Kinds are $\$^*\$$ and $\kappa_1 \rightarrow \kappa_2$. The star means a type whose expressions can have values. Kind arrows classify type constructors, not term functions.

$$\begin{array}{c}
\frac{\alpha : \kappa \in \Delta}{\Delta \vdash \alpha : \kappa} \text{ (K-Var)} \quad \frac{T : \kappa \in \Delta}{\Delta \vdash T : \kappa} \text{ (K-Con)} \\
\\
\frac{\Delta \vdash \tau_1 : \kappa_1 \rightarrow \kappa_2 \quad \Delta \vdash \tau_2 : \kappa_1}{\Delta \vdash \tau_1 \tau_2 : \kappa_2} \text{ (K-App)} \\
\\
\frac{\Delta \vdash \tau_1 : * \quad \Delta \vdash \tau_2 : *}{\Delta \vdash \tau_1 \rightarrow \tau_2 : *} \text{ (K-Arrow)}
\end{array}$$

Every term type and constructor field has kind $*$. In a predicate C τ, τ must have the kind of the class parameter. Thus `Functor` accepts a constructor of kind $* \rightarrow *$, while `Eq` accepts a type of kind $*$. Kinds are inferred jointly within each type-declaration component. Remaining unconstrained kind variables become $*$. Synonym cycles and partial synonym applications are errors, even if later unification could otherwise hide them.

Variables, Literals, Functions

$$\frac{x : \forall \bar{\alpha}. Q \Rightarrow \tau \in \Gamma \quad P \vdash SQ}{P; \Gamma \vdash x : S\tau} \text{ (Var)}$$

Here S replaces the quantified variables with monotypes of matching kinds. Constructors use the same rule with their declared schemes. Literals have fixed types, without a class constraint:

$$\begin{array}{c}
 \frac{}{P; \Gamma \vdash n : \text{Int}} \text{(Int)} \quad \frac{}{P; \Gamma \vdash d : \text{Double}} \text{(Double)} \\
 \\
 \frac{}{P; \Gamma \vdash c : \text{Char}} \text{(Char)} \quad \frac{}{P; \Gamma \vdash s : \text{String}} \text{(String)} \\
 \\
 \frac{P; \Gamma, x : \tau_1 \vdash e : \tau_2}{P; \Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2} \text{(Abs)} \\
 \\
 \frac{P; \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad P; \Gamma \vdash e_2 : \tau_1}{P; \Gamma \vdash e_1 e_2 : \tau_2} \text{(App)}
 \end{array}$$

A lambda parameter is monomorphic throughout its body. Instantiating `id` twice may choose different types. Using a lambda-bound `f` twice must respect the same type. Pattern parameters use the pattern rules below before applying (Abs).

Let, Generalization, and Signatures

$$\frac{P, Q; \Gamma \vdash e_1 : \tau_1 \quad P; \Gamma, x : \sigma \vdash e_2 : \tau_2}{P; \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2} \text{(Let)}$$

Here $\sigma = \forall \bar{\alpha}. Q \Rightarrow \tau_1$. The generalized variables are free in τ_1 or Q , but not in Γ or P . Every variable mentioned by Q must also occur in τ_1 . The solver first reduces and simplifies Q , discharges concrete predicates, and resolves every field constraint. Predicates on variables outside this binding's generalization scope remain in P . A residual predicate with no possible generalization or instance is an error. There is no defaulting and no value restriction.

For an unannotated recursive component, assign a fresh monotype τ_i to each x_i , then check all bodies under those same assumptions:

$$\frac{P, Q; \Gamma, \bar{x} : \bar{\tau} \vdash e_i : \tau_i \quad P; \Gamma, \bar{x} : \bar{\sigma} \vdash e : \tau}{P; \Gamma \vdash \text{let } \bar{x} = \bar{e}_i \text{ in } e : \tau} \text{(Let-Rec)}$$

Generalize the solved types together relative to the outer environment, using the same constraint and ambiguity conditions as (Let). All recursive right-hand sides must be syntactic functions after desugaring. Recursion within an unannotated component is monomorphic.

An annotation is a promise checked with fresh rigid variables:

$$\frac{P, SQ; \Gamma \vdash e : S\tau}{P; \Gamma \vdash e : \forall \bar{\alpha}. Q \Rightarrow \tau} \text{(Signature)}$$

Here S replaces each quantified variable with a fresh skolem of the same kind. These skolems cannot be unified with arbitrary types or escape into the outer environment. The body's required predicates

must follow from P, SQ and the instance environment. An annotated binding is available recursively at its specified scheme, which permits polymorphic recursion. Dependency analysis removes edges to annotated bindings before forming unannotated components. Expression annotations are checked by this rule, then instantiated for use as expressions. They do not introduce first-class polymorphic values.

Patterns, Guards, and Cases

The judgment $\vdash p : \tau \Rightarrow \Gamma_p$ says that a pattern matches a value of type τ and introduces the variables in Γ_p .

$$\begin{array}{c}
 \frac{}{\vdash x : \tau \Rightarrow x : \tau} \text{ (P-Var)} \quad \frac{}{\vdash _ : \tau \Rightarrow \Gamma_0} \text{ (P-Wild)} \\
 \\
 \frac{\vdash p : \tau \Rightarrow \Gamma_p}{\vdash x@p : \tau \Rightarrow \Gamma_p, x : \tau} \text{ (P-As)} \\
 \\
 \frac{K : \forall \bar{\alpha}. \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow T \bar{\alpha} \quad \vdash p_i : S\tau_i \Rightarrow \Gamma_i}{\vdash K \bar{p} : T S\bar{\alpha} \Rightarrow \bar{\Gamma}_i} \text{ (P-Con)}
 \end{array}$$

Here Γ_0 is the empty environment. The pattern environments must have disjoint domains. Literal patterns have the literal's fixed type and introduce no bindings. A record pattern is its constructor pattern in declaration order, with wildcards for omitted fields. List, tuple, and unit patterns use their ordinary constructors.

$$\begin{array}{c}
 \frac{P; \Gamma \vdash e : \tau_s \quad \vdash p_i : \tau_s \Rightarrow \Gamma_i \quad P; \Gamma, \Gamma_i \vdash e_i : \tau}{P; \Gamma \vdash \text{case } e \text{ of } \bar{p}_i \rightarrow e_i : \tau} \text{ (Case)} \\
 \\
 \frac{P; \Gamma, \Gamma_p \vdash g_j : \text{Bool} \quad P; \Gamma, \Gamma_p \vdash e_j : \tau}{P; \Gamma, \Gamma_p \vdash \bar{g}_j \rightarrow \bar{e}_j : \tau} \text{ (Guards)}
 \end{array}$$

Every guarded branch has the case's result type. A row's where bindings are checked by the binding-group rules in the pattern environment, and scope over both its guards and results. Function equations use the same row judgment after introducing one argument per pattern column. All equations agree on arity. Nonexhaustive and unreachable rows produce warnings, not additional typing rules. Failure to match remains a possible runtime error at any result type.

A nonrecursive pattern binding first checks its right-hand side and pattern at the same type. It then generalizes the introduced binders relative to the outer environment, with the conditions of (Let). The pattern's binders are not in scope on that binding's right-hand side.

Nominal Records

Write $field(T, l) = \tau_l$ for the field type in a record declaration parameterized by $\bar{\alpha}$, and let S substitute its actual type arguments. Each rule requires the record constructor to have been identified.

$$\frac{P; \Gamma \vdash e_i : S\tau_{l_i}}{P; \Gamma \vdash K \{ \overline{l_i = e_i} \} : T S\bar{\alpha}} \text{ (Record)}$$

Every declared field occurs exactly once, with no unknown label. The source order determines evaluation order even though field slots use declaration order.

$$\frac{P; \Gamma \vdash e : T S\bar{\alpha} \quad field(T, l) = \tau_l}{P; \Gamma \vdash e.l : S\tau_l} \text{ (Project)}$$

$$\frac{P; \Gamma \vdash e : T S\bar{\alpha} \quad P; \Gamma \vdash v_i : S\tau_{l_i}}{P; \Gamma \vdash e \{ \overline{l_i = v_i} \} : T S\bar{\alpha}} \text{ (Update)}$$

Update labels are distinct and belong to T . The same substitution appears in the input, field types, and result, so an update cannot change type parameters.

Inference generates $Has(\tau, l, \tau_r)$ before T is known. If only one record declares l , it chooses that record with fresh parameters. Otherwise it waits for τ 's head constructor. The solver repeats this process before generalization. Any unresolved field is ambiguous (E0245), even if some whole-program interpretation could choose a record. Field constraints never enter a scheme. $(.l)$ is checked as $\backslash r \rightarrow r.l$, so the same restriction applies.

Class Entailment and Declarations

$$\frac{\pi \in P}{P \vdash \pi} \text{ (Given)} \quad \frac{P \vdash C \tau \quad C \text{ has superclass } D}{P \vdash D \tau} \text{ (Super)}$$

$$\frac{instance \ Q \Rightarrow C \tau \quad P \vdash SQ}{P \vdash C S\tau} \text{ (Instance)}$$

The last rule uses a declared instance and a kind-correct substitution that matches its head to the wanted predicate. Entailment of a set requires entailment of every member. Superclass projection and instance application also construct the dictionary evidence used during elaboration.

A class has one parameter. Its method signatures may mention that parameter but may not add constraints of their own. Other method type variables are implicitly quantified. A default method is checked at its declared type with the class predicate given. Superclass relationships must be acyclic.

An instance head is a constructor applied to distinct variables. Its context constrains only those variables. No two instance heads may overlap. Check each method against the class method signature specialized to that head, using the instance context as givens. Required superclass dictionaries must

also be obtainable. A missing method requires a default. Deriving generates instances that undergo these same checks.

Data declarations introduce constructor schemes, with their parameters universally quantified and their fields as successive arrow arguments. Constructor result types name the declared datatype. The language has no existential variables, GADT equations, or constructor-specific class contexts.

Sugar, IO, and Whole Programs

These forms have no separate inference mechanism. Their expansions determine their types:

Surface form	Typing account
<code>if b then x else y</code>	<code>b</code> : <code>Bool</code> , and both branches have one type
<code>a && b</code> , <code>a b</code>	Both operands are <code>Bool</code> , with conditional evaluation
<code>a + b</code> , other operators	Application of the operator's scheme
<code>-e</code>	Application of negate
<code>(e op)</code> , <code>(op e)</code>	A lambda applying the operator to a fresh parameter
<code>[a..b]</code>	<code>enumFromTo a b</code> , with both bounds <code>Int</code>
Lists, tuples, unit	Applications of their data constructors
<code>where</code>	A local binding group

For a monad constructor $m : * \rightarrow *$, a bind step requires an action of type $m \alpha$ and a continuation whose result has type $m \beta$:

$$\frac{P \vdash \text{Monad } m \quad P; \Gamma \vdash e : m \alpha \quad \vdash p : \alpha \Rightarrow \Gamma_p \quad P; \Gamma, \Gamma_p \vdash s : m \beta}{P; \Gamma \vdash \text{do } \{p \leftarrow e; s\} : m \beta} \text{ (Do-Bind)}$$

An expression statement uses `>>`, a `let` statement uses the binding rules, and a final expression supplies the block's result. Refutable bind patterns expand to a case with an error fallback. There is no `MonadFail` constraint.

`IO` is abstract to the source checker and has kind $* \rightarrow *$. It is an instance of `Monad`, but a source program cannot equate `IO a` with `() → a`. That representation is exposed only during elaboration. Effectful primitives are visible only while compiling the Prelude.

A complete program kind-checks its declarations, validates its class environment, checks binding components in dependency order, and solves all remaining constraints. Its entry point must have type `IO ()`. No unresolved class or field constraint may survive. Top-level initializers run before that action, in dependency order. These evaluation requirements complement the typing rules: well typed does not mean terminating or free of explicit runtime errors.

C

Intermediate Representations

The final compiler has three executable intermediate languages. Each removes a specific obligation from the next pass. These definitions are included from Chapter 24, so the reference and implementation stay together.

Representation	Makes explicit	Still leaves implicit
System F Core	Types, dictionaries, constructors, control flow	Evaluation temporaries and closure environments
ANF	Evaluation order, intermediate results, joins, tail calls	Captured variables and application dispatch
Flat	Lifted code, closure fields, known and unknown calls	Physical storage and machine instructions

Typed Core

A value identifier carries its type. A type application instantiates a type abstraction. Dictionary arguments are ordinary value arguments, so erasing `TyLam` and `TyApp` does not erase class evidence.

```

/// A Core identifier carries its independently checkable type.
#[derive(Clone, Debug, PartialEq, Eq)]
pub struct Id {
    pub name: crate::names::Id,
    pub ty: Type,
}
/// Core type applications retain kinds in the program's binder table.
#[derive(Clone, Debug, PartialEq, Eq)]
pub enum Type {
    Double,
    Int,
    Bool,
    Char,
    String,
    Var(u32),
    Con(crate::data::TyConId, Vec<Type>),
    App(Box<Type>, Box<Type>),
    Arrow(Box<Type>, Box<Type>),
    Forall(u32, Box<Type>),
}
#[derive(Clone, Debug)]
/// System F Core makes type abstraction and instantiation explicit.
pub enum Expr {
    Var(Id),
    Global(Id),
    TyLam(u32, Box<Expr>),
    TyApp(Box<Expr>, Type),
    Lit(Lit),
    Lam(Id, Box<Expr>),
    App(Box<Expr>, Box<Expr>),
    Let(BindGroup, Box<Expr>),
    Con(crate::data::ConId, Vec<Type>, Vec<Expr>),
    Match(Match),
    Case(Box<Expr>, Id, Type, Vec<Alt>),
    Join(crate::names::Id, Vec<Id>, Box<Expr>, Box<Expr>),
    Jump(crate::names::Id, Vec<Expr>, Type),
    PatternFail(Span, Type),
    Prim(Op, Vec<Type>, Vec<Expr>, Span),
}

```

Match belongs to elaborated Core. Match compilation removes it, leaving flat Case alternatives and lexical Join/Jump. Constructors and primitives are saturated. A jump is fully applied and appears

only in tail position within the scope of its join. The lint pass checks these properties independently of the source type checker.

```

/// Flat alternatives inspect one constructor or literal, never nested
patterns.
#[derive(Clone, Debug, PartialEq, Eq)]
pub enum AltCon {
    Con(crate::data::ConId),
    Lit(Lit),
    Default,
}
/// Constructor slots become typed binders scoped over the alternative body.
#[derive(Clone, Debug)]
pub struct Alt {
    pub con: AltCon,
    pub binders: Vec<Id>,
    pub rhs: Expr,
}
#[derive(Clone, Debug)]
/// A recursive binding is guaranteed to have a lambda right-hand side.
pub struct Bind {
    pub name: Id,
    pub rhs: Box<Expr>,
}
#[derive(Clone, Debug)]
/// An ordered component; recursive members are in scope in every right-hand
side.
pub struct BindGroup {
    pub recursive: bool,
    pub binds: Vec<Bind>,
}

```

A recursive group contains functions. A nonrecursive group's right-hand sides are evaluated in dependency order. Case binders name the already evaluated scrutinee. Pattern failure and failing primitives retain source locations.

A-Normal Form

Atoms cannot perform an effect or fail when read from a well-scoped environment. A `Let` names one strict computation. The body type is `Rc<Anf>`, allowing the machine and transformation worklists to share immutable syntax.

```

#[derive(Clone, Debug, PartialEq, Eq)]
/// Atoms cannot perform work or fail when read from a well-scoped
environment.
pub enum Atom {
    Var(Var),
    Global(Id),
    Lit(Lit),
}
#[derive(Clone, Debug)]
/// Lambda parameters are merged only across syntactically adjacent lambdas.
pub struct Lambda {
    pub params: Vec<Var>,
    pub body: Body,
}
#[derive(Clone, Debug)]
/// One strict computation whose operands have already been evaluated.
pub enum Rhs {
    Atom(Atom),
    Call(Atom, Vec<Atom>),
    Con(ConId, Vec<Atom>),
    Prim(Op, Vec<Atom>, Span),
    Lambda(Lambda),
    Field(Atom, u32),
}
#[derive(Clone, Debug)]
/// Jumps transfer within a function; tail calls replace its activation.
pub enum Anf {
    Let(Var, Rhs, Body),
    LetRec(Vec<(Var, Lambda)>, Body),
    Join(Id, Vec<Var>, Body, Body),
    Jump(Id, Vec<Atom>),
    Case(Atom, Vec<(AltCon, Vec<Var>, Body)>),
    Ret(Atom),
    TailCall(Atom, Vec<Atom>),
    Fail(Span),
}

```

`Ret` returns to the current continuation. `TailCall` replaces the current activation. `Jump` stays inside it. A non-tail source case sends its alternatives to one join representing the rest of the computation. The iterative ANF machine uses this distinction to run deep tail recursion without recursive Rust calls.

Flat IR

Every lambda has become a lifted function. Its environment is explicit when it captures values. Function indices refer to the program's function table.

```
#[derive(Clone, Debug)]
/// Only lifted local lambdas receive an environment parameter.
pub struct Func {
    pub name: FuncId,
    pub source: Var,
    pub env: Option<Var>,
    pub params: Vec<Var>,
    pub body: Body,
    pub captures: Vec<Var>,
}
#[derive(Clone, Debug)]
/// Function globals are installed before ordered strict value initializers
run.
pub struct GlobalInit {
    pub name: Var,
    pub function: Option<FuncId>,
    pub body: Body,
}
#[derive(Clone, Debug)]
pub struct Program {
    pub funcs: Vec<Func>,
    pub globals: Vec<GlobalInit>,
    pub metadata: core::Program,
    pub reps: crate::rep::Table,
}
#[derive(Clone, Debug)]
/// PatchClosure is a compiler-only operation used after all letrec
allocations.
pub enum Rhs {
    Atom(Atom),
    CallKnown(FuncId, Option<Atom>, Vec<Atom>),
    CallUnknown(Atom, Vec<Atom>),
    AllocCon(ConId, Vec<Atom>),
    AllocClosure(FuncId, Vec<Atom>),
    EnvField(Var, u32),
    Field(Atom, u32),
    Prim(Op, Vec<Atom>, Span),
    PatchClosure(Atom, u32, Atom),
}
```

```
#[derive(Clone, Debug)]
/// Control transfers are explicit and joins remain local to each function.
pub enum Flat {
    Let(Var, Rhs, Body),
    Case(Atom, Vec<(AltCon, Vec<Var>, Body)>),
    Join(Var, Vec<Var>, Body, Body),
    Jump(Var, Vec<Atom>),
    Ret(Atom),
    TailCallKnown(FuncId, Option<Atom>, Vec<Atom>),
    TailCallUnknown(Atom, Vec<Atom>),
    Fail(Span),
}
```

Known calls name a lifted function with the correct number of arguments. Unknown calls use `eval/apply`, which supports exact, partial, and oversaturated calls. `PatchClosure` initializes a recursive capture after allocating every closure in the group. It is an internal operation, never a source mutation primitive.

Numeric Evidence and Storage

Chapter 23 preserves a small classification derived from checked Core types:

```
#[derive(Clone, Copy, Debug, PartialEq, Eq)]
pub enum Rep {
    Int,
    Double,
    Word,
}
```

This is type evidence, not a calling convention. Call results and parameters remain words even when their evidence says `Int` or `Double`. LLVM's local locations distinguish words, native numbers, and reloadable roots:

```
/// Storage is local to codegen. All ABI boundaries still carry tagged words.
#[derive(Clone, Copy)]
pub(super) enum Loc<'ctx> {
    Root(PointerValue<'ctx>),
    Word(IntValue<'ctx>),
    Int(IntValue<'ctx>),
    Double(inkwell::values::FloatValue<'ctx>),
}
```

The copying backend reloads a Root after a possible collection. Arithmetic can keep a native number in a register until a boundary requires tagging or boxing. The runtime ABI specifies those boundaries.

D

Runtime ABI

This appendix describes the final native compiler and `fun-rt-24`. Supported hosts use 64-bit words and pointers: Apple Silicon macOS and x86-64/AArch64 Linux. The ABI is internal to this compiler edition. There is no user-written FFI.

Words and Objects

Every source value crossing a call boundary is an LLVM `i64` and a Rust `u64`. A low bit of one denotes an immediate. A low bit of zero denotes an aligned object address, except for temporary zero placeholders during initialization.

Value	Encoding
Int	$(n \ll 1) \mid 1$, signed 63-bit two's complement
Char	Unicode scalar shifted left one bit, low bit set
Nullary constructor	$(tag \ll 1) \mid 1$
Heap value	Address of its header, aligned to eight bytes

The expected type determines the interpretation. Thus `False`, `[]`, `Nothing`, and `()` may all have word value 1 without being interchangeable.

```
header: [ payload size: 32 bits | constructor tag: 24 bits | kind: 8 bits ]
object: [ header ][ payload word 0 ][ payload word 1 ] ...
```

Size counts payload words. Constructor tags are local to their data type and follow declaration order. Fixed Prelude tags are `False`=0, `True`=1, `[]`=0, `()`=1, `Nothing`=0, `Just`=1, `()`=0, and `LT`=0, `EQ`=1, `GT`=2.

Kind	ID	Payload	Traced fields
CON	0	Constructor fields	All
CLOSURE	1	Slow code address, raw arity, captures	Captures
PAP	2	Closure word, raw argument count, arguments	Closure and arguments
DOUBLE	3	Binary64 bits	None
STRING	4	Raw byte length, UTF-8 bytes padded to words	None
FORWARD	5	Destination address during collection	Collector-only marker

There is no heap allocation for a nullary constructor. Every movable object therefore has room for a forwarding address. Strings count bytes in storage and Unicode scalars in `length`-style primitives.

Entries and Application

Entry	Convention and signature
Known function	<code>tailcc i64 f(i64 env, i64 a1, ..., i64 an)</code>
Slow adapter	<code>C: u64 slow(u64 closure, const u64 *args)</code>
General apply	<code>C: u64 rt_apply(u64 closure, usize n, const u64 *args)</code>
Small apply	<code>C: u64 rt_applyN(u64 closure, u64 a1, ..., u64 aN), N=1..4</code>
PAP construction	<code>C: u64 rt_make_pap(u64 closure, usize n, const u64 *args)</code>
Recursive capture patch	<code>C: void rt_closure_set(u64 closure, usize index, u64 value)</code>

The slow adapter receives exactly the function’s syntactic arity. The general apply routine flattens PAP arguments, invokes exact calls, or applies remaining arguments to a returned function. Only known tail calls have a guaranteed native tail transfer. Crossings between C and `tailcc` retain their ABI adapter frame.

All arguments are strict and already evaluated. Raw argument arrays must remain readable for the call. A collector may relocate their *values*, so persistent buffers belong to a root owner. A slow entry loads its arguments and establishes its generated root frame before any allocation.

Allocation and Roots

Export	Contract
<code>void rt_init()</code>	Initialize Boehm and register the calling runtime thread
<code>u64 *rt_alloc(usize words)</code>	Reserve scanned storage, including header
<code>u8 *rt_alloc_atomic(usize bytes)</code>	Reserve pointer-free storage
<code>void rt_use_copying()</code>	Select the single-threaded moving heap before initialization
<code>void rt_roots_push(u64 *slots, usize n)</code>	Register a writable, initialized generated root range
<code>void rt_roots_pop()</code>	Remove the current generated root range
<code>void rt_global_root(u64 *slot)</code>	Register a static global for the process lifetime

Allocation can collect before returning. The caller initializes the entire new object before any further allocation. Native startup registers globals before installing functions or evaluating values. Static string objects are immutable and stationary. The Boehm backend also retains their address table.

Rust's `Roots` and `GlobalRoot` owners register stable buffers until drop. The moving collector updates their cells. Borrowed string slices and unrooted copies of pointer words must not survive an allocation. Primitive exports decode strings into owned Rust values before allocating their result.

Primitives, Errors, and Exit

Every `prim_*` operation has a C export named `rt_prim_*`. Each argument and the return value is a word, including a unit result. Integer and Double arithmetic may instead be emitted inline. Nontrivial operations share their safe Rust implementation with the interpreters.

Primitive family	Arguments	Result
intAdd/Sub/MuL/Quot/Rem/Div/Mod	Two Int words	Int word
intNeg	Int	Int
intEq/Lt/Le	Two Int words	Bool immediate
dblAdd/Sub/MuL/Div/Pow	Two boxed Doubles	Boxed Double
dblNeg/Sqrt/Exp/Log/Sin/Cos	Boxed Double	Boxed Double
dblEq/Lt/Le	Two boxed Doubles	Bool immediate
intToDbL, dbLFloor/Round/Truncate	Source numeric value	Converted numeric value
chrOrd, chrChr, chrEq/Lt	Character/integer operands	Integer/character/Bool
strAppend/Length/Eq/Compare	One or two Strings	String/Int/Bool/Int
strUnpack, strPack	String or character list	Character list or String
showInt/Double/Char/String	One value of the named type	String
readInt	String	Maybe Int
putStr, getLine	String or unit	Unit or String
trace	String message, arbitrary value	The same value
error, exit	String or Int	Does not return

Integer results wrap modulo 2^{63} . Floating conversion saturates to the Int range, with NaN becoming zero. Zero division, invalid characters, EOF on getLine, and failed matches are runtime errors.

Export	Behavior
rt_error(u64 string)	Flush stdout, write fun: MESSAGE, exit 1
rt_pattern_fail(const char *file, u32 line, u32 col)	Report source location, exit 1
rt_div_zero()	Report fun: divide by zero, exit 1
rt_exit(i32 code)	Flush stdout and terminate

Generated C main initializes the runtime, evaluates globals in dependency order, applies source main to unit word 1, and exits successfully. A runtime panic aborts rather than unwinding through generated code.

E

Diagnostic Index

This table is generated from the final compiler's error registry. Codes are stable identifiers. Messages and secondary labels add context about the source program. The primary label points to the location that triggered the diagnostic.

Code	Meaning
E0001	Invalid token
E0002	Invalid syntax
E0003	Invalid layout
E0004	Invalid projection dot
E0101	Unbound variable
E0102	Duplicate definition
E0103	Unbound constructor or type
E0104	Duplicate field
E0110	Reserved operator redefined
E0111	Primitive outside Prelude
E0120	Recursive non-function binding
E0130	Fixity without definition
E0201	Type mismatch
E0202	Infinite type
E0203	Signature too general
E0210	No instance
E0211	Invalid instance methods or duplicate instance
E0230	Ambiguous type
E0240	Record must have one constructor
E0241	Missing field
E0242	Duplicate field binding
E0243	Unknown field
E0244	Type does not declare field
E0245	Ambiguous field access
E0250	Main must have type IO ()
E0301	Kind mismatch
E0302	Partially applied synonym
E0303	Cyclic synonym
W0002	Non-exhaustive match

Lexing, layout, and parsing use E00xx. Renaming uses E01xx, typing uses E02xx, and kind checking uses E03xx. A pass may report several diagnostics, but the driver does not run later passes after an error.

W0001 reports a possibly missing match and includes witness patterns when available. W0002 identifies an unreachable clause. Warnings do not reject a program. A missing match that execution actually reaches raises a runtime error.

Runtime errors have messages rather than compiler codes. They begin with `fun:` and exit with status 1. `exitWith` uses the requested status. Compiler dumps and internal lint failures are developer tools, not an alternative diagnostic API.

For an ambiguous record field, annotate the receiver's type. For an ambiguous class constraint, make the constrained type visible in the result or select it explicitly. Fun does not default unresolved numeric constraints.

Bibliography

- Appel, Andrew W. 1992. *Compiling with Continuations*. Cambridge University Press. <https://doi.org/10.1017/CBO9780511609619>.
- Boehm, Hans-Juergen, and Mark Weiser. 1988. “Garbage Collection in an Uncooperative Environment.” *Software: Practice and Experience* 18 (9): 807–20. <https://doi.org/10.1002/spe.4380180902>.
- Cheney, C. J. 1970. “A Nonrecursive List Compacting Algorithm.” *Communications of the ACM* 13 (11): 677–78. <https://doi.org/10.1145/362790.362798>.
- Damas, Luis, and Robin Milner. 1982. “Principal Type-Schemes for Functional Programs.” In “Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL ’82.” Special issue, *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL ’82*, POPL ’82, 207–12. <https://doi.org/10.1145/582153.582176>.
- Felleisen, Matthias, and Daniel P. Friedman. 1986. *Control Operators, The SECD-Machine, And the Lambda-Calculus*. Technical Report No. TR197. <https://hdl.handle.net/2022/33996>.
- Flanagan, Cormac, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. “The Essence of Compiling with Continuations.” In “Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation.” Special issue, *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI93, 237–47. <https://doi.org/10.1145/155090.155113>.
- Girard, Jean-Yves. 1972. “Interprétation Fonctionnelle Et Élimination Des Coupures De L'arithmétique D'ordre Supérieur.” Doctoral dissertation. <https://girard.perso.math.cnrs.fr/These.pdf>.
- Henderson, Fergus. 2002. “Accurate Garbage Collection in an Uncooperative Environment.” In “Proceedings of the 3rd International Symposium on Memory Management.” Special issue, *Proceedings of the 3rd International Symposium on Memory Management*, ISMM02, 150–56. <https://doi.org/10.1145/512429.512449>.
- Jones, Mark P. 1993. “A System of Constructor Classes: Overloading and Implicit Higher-Order Polymorphism.” In “Proceedings of the Conference on Functional Programming Languages and

- Computer Architecture.” Special issue, *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA93, 52–61. <https://doi.org/10.1145/165180.165190>.
- Jones, Mark P. 1999. “Typing Haskell in Haskell.” In “Haskell Workshop.” Special issue, *Haskell Workshop*, <https://web.cecs.pdx.edu/~mpj/thih/thih-sep1-1999/index.html>.
- Jones, Richard, Antony Hosking, and Eliot Moss. 2023. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. 2nd ed. Chapman and Hall/CRC. <https://doi.org/10.1201/9781003276142>.
- Kiselyov, Oleg. 2013. “How Ocaml Type Checker Works—or What Polymorphism and Garbage Collection Have in Common.” <https://okmij.org/ftp/ML/generalization.html>.
- Leroy, Xavier. 1990. *The ZINC Experiment: An Economical Implementation of the ML Language*. Technical Report No. RT-117. <https://caml.inria.fr/pub/papers/xleroy-zinc.pdf>.
- Levy, Paul Blain. 2001. “Call-by-Push-Value.” Doctoral dissertation. <https://pblevy.github.io/papers/thesisqmwphd.pdf>.
- Maranget, Luc. 2007. “Warnings for Pattern Matching.” *Journal of Functional Programming* 17 (3): 387–421. <https://doi.org/10.1017/s0956796807006223>.
- Maranget, Luc. 2008. “Compiling Pattern Matching to Good Decision Trees.” In “Proceedings of the 2008 ACM SIGPLAN Workshop on ML.” Special issue, *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ICFP08, 35–46. <https://doi.org/10.1145/1411304.1411311>.
- S. Marlow. 2010. *Haskell 2010 Language Report*. <https://www.haskell.org/onlinereport/haskell2010/>.
- Marlow, Simon, and Simon Peyton Jones. 2004. “Making a Fast Curry: Push/enter Vs. Eval/apply for Higher-Order Languages.” In “Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming.” Special issue, *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, ICFP04, 4–15. <https://doi.org/10.1145/1016850.1016856>.
- Maurer, Luke, Paul Downen, Zena M. Ariola, and Simon Peyton Jones. 2017. “Compiling Without Continuations.” In “Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation.” Special issue, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’17, 482–94. <https://doi.org/10.1145/3062341.3062380>.
- Mitchell, Neil, and Shayne Fletcher. n.d. “Record Dot Syntax.” <https://ghc-proposals.readthedocs.io/en/latest/proposals/0282-record-dot-syntax.html>.
- Pierce, Benjamin C. 2002. *Types and Programming Languages*. MIT Press. <https://www.cis.upenn.edu/~bcpierce/tapl/>.
- Rémy, Didier. 1992. *Extension of ML Type System with a Sorted Equational Theory on Types*. Technical Report No. RR-1766. <https://gallium.inria.fr/~remy/ftp/eq-theory-on-types.pdf>.

Reynolds, John C. 1974. "Towards a Theory of Type Structure." In *Programming Symposium*. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-06859-7_148.