

Type-Safe Tree Transformations for Precisely-Typed Compilers

by

Pedro Oliveira

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2024

© Pedro Oliveira 2024

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Compilers translate programs from a source language to a target language. This can be done by translating into an intermediate tree representation, performing multiple partial transformations on the tree, and translating out. Compiler implementers must choose between encoding transformation invariants with multiple tree types at the cost of code boilerplate and duplication, or forgoing static correctness by using a broad and imprecise tree datatype.

This thesis proposes a new approach to writing precisely-typed compilers and tree transformations without code boilerplate or duplication. The approach requires encoding the tree nodes using two type indices, one for the phase of the root node and another for the phase of all children nodes. This tree datatype can represent partially transformed nodes, and distinguish between trees of different phases.

In order to make this approach possible it was necessary to modify the Scala type checker to better infer types in ‘match’-expressions. Precisely-typed tree transformations make use of the default case of a ‘match’-expression; this case must be elaborated to a type which is the type difference of the scrutinee expression and all previously matched patterns.

We demonstrate the viability of this approach by implementing a case study for a precisely-typed compiler for the instructional language Lacs. This case study modifies the existing implementation of Lacs to track which subset of tree nodes may be present before and after any given tree transformation. We show that this approach can increase static correctness without the burden of additional code boilerplate or duplication.

Acknowledgements

I am grateful for my parents for nurturing my interest in computers all these years. They always encouraged me to be a better, more curious, version of myself.

I would like to thank my sister for being an exceptional rubber duck when I rambled on about types not matching and programs not compiling.

I would especially like to thank my advisor, Ondřej Lhoták, for his direction in my studies. He introduced me to the world of Scala and the many interesting questions within it, only one of which this thesis attempts to answer. Thanks as well to Gregor Richards and Yizhou Zhang for reading this thesis and providing valuable feedback.

Finally, I would like to thank my amazing partner, Sakina, for her continued love and support. Always cheering my breakthroughs, and believing in me through setbacks. This thesis started due to her encouragement – one day I'll be able to explain to her what it does.

Table of Contents

1	Introduction	1
2	Type-Safe Tree Transformations	6
2.1	Monolithic Compiler	7
2.2	Composable Compiler	10
2.3	Uni-Typed Compiler	13
2.4	Phase-indexed compiler	17
2.5	Precise-typed compiler	22
3	Narrowing in Pattern Matching	29
3.1	Motivation	29
3.2	Overview	31
3.3	Type Decomposition	33
3.4	Type Difference	34
3.5	Complete and Incomplete Patterns	36
3.6	Disjoint Intersections	38
4	Extension: Folds and Unfolds	42
4.1	Encoding with Union Types	43
4.2	Encoding with Match Types	47
4.3	Discussion	50

5	Case Study: Lacs Compiler	52
5.1	Existing Implementation	52
5.1.1	Program Representation	52
5.1.2	Shallow Embedding	55
5.1.3	Tree Transformations	56
5.1.4	Garbage Collection	58
5.2	Modified Implementation	59
5.2.1	Program Representation	59
5.2.2	Tree Transformations	60
5.2.3	Type Signatures	61
5.3	Discussion	62
5.3.1	Selective Transformations	62
5.3.2	Compile-time Exhaustiveness	64
5.3.3	Preventing Common Mistakes	65
5.3.4	Static Type Conversions	67
5.3.5	Handling Variance	68
6	Related Work	70
6.1	Recursion Schemes	70
6.2	Data types à la carte	70
6.3	Trees that Grow	71
6.4	Scrap Your Boilerplate	72
6.5	Restrictable Variants	72
6.6	Nanopass Compilers	72
6.7	Flow Typing	73
6.8	Expression Problem	73
6.9	Exhaustiveness Checking	74
7	Conclusion	75
	References	76

Chapter 1

Introduction

The field of software development has grown in both importance and profitability over the years. As the industry grows larger, individual projects grow in size as well. Developers need to ensure projects are both correct and maintainable, so that the projects fulfill their pre-existing requirements and are malleable to changing future requirements. As software projects grow in size, the challenges to ensure correctness and maintainability grow as well.

Static type systems provide a way to guarantee some kinds of correctness. A type checker can determine whether or not a value belongs to a certain type without running the program. This allows programmers to convey more information to the type checker, ruling out entire sets of possible values. If the type checker discovers that a value may not have the desired type, an error is reported at compile time. This is a lightweight technique to rule out an entire class of errors, greatly reducing the number of tests that need to be written. More advanced type systems present the developer with a tradeoff between more static guarantees at the cost of additional complexity.

Type systems also help to create maintainable programs. When a developer annotates a term with a type, they create a computer checkable contract for future developers. Past developers inform future developers of the expected type of a term, helping understanding and thus maintainability. Future developers must either adhere to the contract by conforming to the existing type, or modify the contract by changing the type annotation. More precise types allow the type checker to enforce more precise contracts and give future developers more information to understand the program. The challenge in maintainability comes in designing types that disallow incorrect changes to the program, while allowing correct changes.

Compilers are large software projects that translate a program in a source language

to a target language. Traditional compilers parse and elaborate a program in the source language into an intermediate tree representation, which is progressively lowered and simplified before emitting the target language. This intermediate step may be further subdivided into successive transformations between intermediate tree representations. These transformations to the trees are at the heart of the compilation pipeline so it is essential that the transformations are both correct and maintainable.

In order to achieve correctness and maintainability, these tree transformations often require developers to create and maintain program invariants, such as restricting tree nodes to only appear before or after specific transformations. Unfortunately, these phase invariants are often enforced through runtime assertions or unit tests, instead of leveraging strong static types.

It is ironic that compiler writers rarely make full use of the strong type systems they themselves implement for others. Why not? Adding more precise types comes with additional costs. A compiler with N tree transformations would require $N + 1$ distinct types of trees. If the transformation only modifies a subset of each tree, the unmodified nodes would need duplicate definitions for each subsequent type. Each tree type would additionally require utility functions for printing, traversing and parsing them; any duplicate nodes would require further function duplication. This pattern would make the compiler too costly to develop and unwieldy to maintain.

To avoid this problem, compiler writers often use a small number of imprecise types. In this scheme, transformation functions accept and return trees of the same type. Imprecise types are unsatisfactory because the type system cannot check that a transformation indeed took place. If the developer makes an error, causing the function to return an incorrect tree, the type checker will accept it! Future developers must also rely on documentation, which may be stale, or implicit folklore, to determine which invariants must be maintained across different transformations. Imprecise types cannot distinguish between the type of tree before and after a transformation.

For a real world example, *Dotty*¹, the Scala compiler, contains over 100 phase transformations², any one of which may insert or remove a group of nodes from the tree. The compiler uses a few different datatypes to enforce some invariants, but most phase transformations operate over the same `Tree` datatype that allows any node to be present in the tree at any point in the transformation pipeline. Each phase may provide a custom post-condition to be checked at runtime, but there is no way to statically check the common case of unexpected nodes in a `Tree`.

¹<https://github.com/scala/scala3>

²<https://dotty.epfl.ch/docs/contributing/architecture/phases.html>

The pattern of using a few broad types without distinguishing between small tree transformations is also present in other compilers such as *swift*³ and *rustc*⁴. Compiler writers face a tradeoff when attempting to enforce more granular phase invariants through types. Each new datatype requires more boilerplate, reducing the very maintainability it is supposed to improve. Additional correctness guarantees from the more precise types are too costly in practice.

A solution to these woes is to model the intermediate trees using a single type constructor `Tree[+_]` indexed by a phase `P`, such that `Tree[P]` represents the tree which may contain nodes of phase `P`. A tree containing nodes disallowed by `P` cannot be typed by `Tree[P]`. Additionally, `Tree` is covariant in `P`, specified by the `+` at definition, making `Tree[P2]` a subtype of `Tree[P]` if `P2` is a subtype of `P`. Having a single type solves the issues with code duplication, while an explicit phase index retains the ability to distinguish trees across transformations. A transformation from a phase `P1` to a phase `P2` can be typed as a function `Tree[P1] => Tree[P2]`. A generic utility function, such as printing a tree, can be typed as `Tree[Any] => String` where `Tree[Any]` is a supertype of all trees. The `Tree[+_]` type makes a compiler precisely-typed: each tree transformation can specify which subset of nodes are allowed before and after it is run.

Simply using `Tree[+_]` solves the problem of distinguishing trees of different phases, as well as writing functions that operate on any tree, but it complicates tree traversals. Using a naive implementation, a transformation from `Tree[P1]` to `Tree[P2]` needs to manually recurse through every nodes' children before transforming the cursor node. This is especially problematic when creating polymorphic transformations, such as `Tree[P | P1] => Tree[P | P2]` where `P` is a type variable. Polymorphic transformations must explicitly inspect all nodes that may be present in any phase. This is far too much ceremony!

Ideally the developer should create an abstraction, such as a higher order function `transform`, to factor out the commonalities across transformations. However, it is impractical to type such a function for `Tree[+_]` as presented. While transforming a tree from phase `P1` to `P2`, the current node is neither of type `Tree[P1]` nor `Tree[P2]`. The children of the node have already been transformed to the next phase, but the current node is still in the previous phase. Such a node cannot be typed as either `Tree[P1]` or `Tree[P2]`, since it has only been partially transformed.

The solution is to redefine `Tree[+_]` such that it can express trees with children of a different phase than the root node. A new type `TreeK[+PSelf, +PChild]` is introduced, where `PSelf` is the phase index for the current root node and `PChild` is the phase index for

³<https://www.swift.org/documentation/swift-compiler/>

⁴<https://rustc-dev-guide.rust-lang.org/overview.html>

all nested child nodes. `Tree[+P]` is just a special case of `TreeK[+_, +_]` where both the root and child nodes are of the same phase. With this, the `transform` function can now be typed as `(Tree[P1], TreeK[P1, P2] => TreeK[P2, P2]) => Tree[P2]`. This technique makes it practical to implement precisely-typed tree transformations for compilers without the burden of extraneous boilerplate.

Precisely-typed compilers and tree transformations with minimal boilerplate are discussed in Chapter 2. Precisely-typed compilers involve creating an all-encompassing tree datatype which can be refined by different phase indices for each transformation phase. Each tree transformation can precisely track which subset of nodes are allowed in their input and output trees at compile time.

Unfortunately, these techniques are impractical to implement with the existing Scala type checker. A problem arises when writing a one-layer function `f` to be given to `transform`. In order to change the type of a node from `TreeK[P1, P2]` to `TreeK[P2, P2]` all nodes must be inspected and reconstructed, even the ones that are unchanged by `f`! This is because in a ‘match’-expression⁵, once a scrutinee term is matched against, the resulting pattern will have the same type as the scrutinee expression. Since this is not the same as the desired output type for the function, type checking fails. Ideally, developers should be able to add a default case that considers all node types not already explicitly covered by earlier cases in the pattern match, keep them unchanged, but elaborate them to the output type.

In order to elaborate precisely-typed tree transformations, it is necessary to introduce type narrowing in ‘match’-expressions so that a sufficiently precise type for the default case can be inferred by the compiler. This technique requires decomposing the type of the scrutinee expression and successively narrowing the remaining expression type under certain conditions. As such, any unchanged nodes can be elaborated with the type difference of the scrutinee type and all previously matched patterns. This allows any unchanged nodes to be elaborated with a subtype of the output tree. Chapter 3 introduces type narrowing in ‘match’-expressions, type decomposition, type difference, and complete patterns.

It is possible to extend type-safe tree transformations to allow for general fold and unfold operations; this is explained in Chapter 4. The fold operation allows a tree representation to be collapsed into a single value one layer at a time, while the unfold operation allows a seed value to be expanded into a full tree.

A case study of applying the precisely-typed compiler technique to the Lacs compiler is presented in Chapter 5. Lacs is an instructional compiler implemented in Scala used

⁵<https://docs.scala-lang.org/scala3/book/control-structures.html#match-expressions>

for teaching an undergraduate course at the University of Waterloo. The modified implementation makes use of various phase tags to track which tree nodes are allowed in specific compiler phases. This serves as a proof of concept for implementing type-safe tree transformations in larger compiler projects.

Chapter 6 discusses related work and Chapter 7 concludes the thesis.

Chapter 2

Type-Safe Tree Transformations

Real world production compilers often have hundreds of thousands of lines of code, translating from enormous source languages to very low-level and detailed target languages. Like any large software project, it is recommended to follow software development best practices, like minimizing code duplication and boilerplate, while maximizing type-safety. Achieving some of these goals often requires compromising on others.

To explore the different patterns involved while developing a compiler and the tradeoffs they afford, this chapter presents a series of small compilers that translate a small language **Source** to a core language **Target**. The **Source** language contains boolean literals, ‘if’-expressions, ‘let’-expressions, and variables; whereas the **Target** language only supports variables, lambda abstractions, and lambda applications. In order to be a more representative example, each compiler must provide additional niceties such as pretty printing capabilities and an interpreter for **Target**. The pretty-printing requirement serves as an example for the various analyses and helper functions present in production compilers. The interpreter requirement serves to highlight the type-safety of the final stage of compilation; in real compilers, this could be emitting machine code or executing the program. The final requirement is that the translation from **Source** to **Target** can be built up by composing smaller partial translations together. This requirement is not strictly necessary in this small example, but it is essential to manage complexity while translating larger languages.

Section 2.1 discusses a design that performs the entire **Source** to **Target** translation in a single step. Section 2.2 discusses a design that decomposes the translation into smaller partial translations while maintaining type-safety. Section 2.3 discusses a design which forgoes some type-safety to achieve better code reuse. Section 2.4 discusses a design which maintains type-safety but requires more code boilerplate. Section 2.5 discusses a design

which achieves minimal boilerplate while maximizing type-safety.

2.1 Monolithic Compiler

An initial design to translate `Source` to `Target` is a monolithic compiler. Such a compiler performs the entire translation from `Source` to `Target` at once. The `Source` and `Target` languages can be represented as distinct datatypes as shown in Listing 2.1. The `Source` enum contains constructors for each of the potential nodes in the input language: `True` and `False` (lines 3-4) represent boolean literals; `If` (line 5) represents ‘if’-expressions; `Var` (line 7) represents variable reads; and `Let` (line 8) represents ‘let’-expressions. The `Target` enum contains the constructors for the Lambda Calculus output language: `Var` (line 12) represents variable reads; `Lambda` (line 13) represents lambda abstractions; and `Apply` (line 14) represents lambda applications. Each datatype is entirely independent from the other and fully recursive with itself.

```
1 object Monolithic {
2   enum Source {
3     case True
4     case False
5     case If(cond: Source, con: Source, alt: Source)
6
7     case Var(name: String)
8     case Let(name: String, value: Source, body: Source)
9   }
10
11  enum Target {
12    case Var(name: String)
13    case Lambda(param: String, body: Target)
14    case Apply(fun: Target, arg: Target)
15  }
16
17  import Source as S
18  import Target as T
19
20  // ...
21 }
```

Listing 2.1: Source and Target definitions

Implementing the pretty-printer and interpreter for `Source` and `Target` is fairly straightforward (Listing 2.2). The printers `prettyS` and `prettyT` (lines 4 and 14) simply recurse

through the trees, translating each node into a `String`. The base cases (lines 5, 6, 9, and 15) translate a node into a `String`. The recursive cases (lines 7, 10, 16, and 18), call `prettyS` or `prettyT` on their child nodes before using the resulting values to construct a `String`.

The `interpret` function (line 22), contains an inner function `go` (line 23) that encapsulates the environment and performs the recursion. The function `go` pattern matches on the given expression, and applies the reduction rules for each node (lines 24-31), possibly by recursing on itself.

The constructors for the `Source` and `Target` datatypes represent distinct nodes, forcing the `prettyS` and `prettyT` functions to not contain any duplicate code. Additionally, because `Target` only contains nodes that can be reduced, the `interpret` function handles all possible inputs without resorting to runtime exceptions for unhandled input nodes. While it is important for the developer to call the recursive functions on every child node, the type checker will reject their program otherwise, pointing to the error at compile-time instead of requiring debugging the code at runtime. This is considered type-safe since the type checker will not accept a program that may cause a runtime type error. All of `prettyS`, `prettyT`, and `interpret` are type-safe, without any boilerplate, or duplicate code.

```

1 object Monolithic {
2   // ...
3   // Pretty Printing
4   def prettyS(source: Source): String = source match {
5     case S.True => "true"
6     case S.False => "false"
7     case S.If(cond, con, alt) =>
8       s"(if ${prettyS(cond)} ${prettyS(con)} ${prettyS(alt)})"
9     case S.Var(name) => name
10    case S.Let(name, value, body) =>
11      s"(let ($name ${prettyS(value)}) ${prettyS(body)})"
12  }
13
14  def prettyT(target: Target): String = target match {
15    case T.Var(name) => name
16    case T.Lambda(param, body) =>
17      s"(lambda ($param) ${prettyT(body)})"
18    case T.Apply(fun, arg) => s"(${prettyT(fun)} ${prettyT(arg)})"
19  }
20
21  // Interpreting
22  def interpret(target: Target): Target = {
23    def go(exp: Target, env: Map[String: Target]): Target = exp match {

```

```

24     case T.Var(name)                => env.getOrElse(name, exp)
25     case T.Lambda(param, body) => T.Lambda(param, go(body, env))
26     case T.Apply(fun, arg) =>
27         val fun2 = go(fun, env)
28         fun2 match {
29             case T.Lambda(param, body) => go(body, env + (param -> arg))
30             case -                      => T.Apply(fun2, go(arg, env))
31         }
32     }
33     go(target, Map.empty)
34 }
35
36 // ...
37 }

```

Listing 2.2: Pretty-printing and interpreting Source and Target

To compile `Source` to `Target`, it is only necessary to recurse through the `source` tree applying the appropriate translation rules to each node (Listing 2.3, lines 4-20). The first part of the function (line 5) applies rules for translating the subset of `Source` which represents boolean expressions, while the second part (line 16) applies rules for the subset of the language representing reading and defining variables. Once again, if the developer misses a recursive case, the type checker will help them in fixing their mistake at compile time.

In a small example like this, the `compile` function is understandable in its entirety, but this does not scale to larger compilers. Due to `Source` and `Target` being distinct types, it is impossible to decompose `compile` into multiple passes across separate functions. Suppose the developer wanted to write two hypothetical functions: `ConvertBool` and `ConvertLet`. The hypothetical `ConvertBool` function should apply all the translation rules from the subset of `Source` representing boolean expressions to `Target`, and `ConvertLet` should apply all the translation rules from the subset of `Source` representing reading and defining variables to `Target`. If these two functions existed, the developer could then compose them together to form the `compile` function, which would fully translate `Source` to `Target`. However, these functions are not typeable in this design of compiler. The function `ConvertBool` would need to accept a `Source` value, and return a mixture of `Source` and `Target` value. As `Source` and `Target` are distinct types, this is not possible. The same would be true for the function `ConvertLet`.

The benefit of this implementation is that `compile` is fully type-safe. The type checker ensures that every `Source` node is translated into `Target` nodes, otherwise it will report a compile-time error. It is still possible to mistranslate a node and introduce a bug, but

the type checker for the implementation language can rule out many sources of error. The main drawback of this approach is that, for larger languages, the `compile` function will become hard to maintain and understand.

```
1 object Monolithic {
2   // ...
3
4   def compile(source: Source): Target = source match {
5     // Booleans
6     case S.True  => T.Lambda("x", T.Lambda("y", T.Var("x")))
7     case S.False => T.Lambda("x", T.Lambda("y", T.Var("y")))
8     case S.If(cond, con, alt) =>
9       val ifBody =
10        T.Apply(T.Apply(T.Var("p"), T.Var("a")), T.Var("b"))
11        val if_ = T.Lambda("p", T.Lambda("a", T.Lambda("b", ifBody)))
12        T.Apply(
13          T.Apply(T.Apply(if_, compile(cond)), compile(con)),
14          compile(alt)
15        )
16    // Lets
17    case S.Var(name) => T.Var(name)
18    case S.Let(name, value, body) =>
19      T.Apply(T.Lambda(name, compile(body)), compile(value))
20  }
21 }
```

Listing 2.3: Monolithic compiler

2.2 Composable Compiler

The composable compiler aims to solve the biggest downside of the monolithic compiler: it introduces a `Partial` language to serve as a halfway point between the `Source` and `Target` languages. The composable compiler first translates `Source` into `Partial`, then translates `Partial` into `Target`. By introducing an intermediate language, this pattern enables the programmer to decompose the translation into separate partial steps, which can later be composed together to form the full translation.

The representations for `Source` and `Target` remain unchanged from the previous design (Listing 2.4, lines 2-3). The `Partial` language is represented as a new datatype (line 5) containing elements of both `Source` and `Target`. Like `Source`, `Partial` contains variable reads and ‘let’-expressions (lines 6 and 7), and like `Target`, it contains lambda abstractions and applications (lines 9 and 10).

```

1 object Composable {
2   type Source = Monolithic.Source
3   type Target = Monolithic.Target
4
5   enum Partial {
6     case Var(name: String)
7     case Let(name: String, value: Partial, body: Partial)
8
9     case Lambda(param: String, body: Partial)
10    case Apply(fun: Partial, arg: Partial)
11  }
12
13  import Monolithic.{Source as S, Target as T}
14  import Partial as P
15
16  // ...
17 }

```

Listing 2.4: Partial language

The `compile` function (Listing 2.5, line 4) is composed of two other functions: `ConvertBool` (line 11), and `ConvertLet` (line 30). `ConvertBool` translates from `Source` to `Partial` by applying the translation rules from boolean expressions to lambda expressions (line 12) and by recursing on ‘let’-expression nodes that will be translated later (line 24). `ConvertLet` translates from `Partial` to `Target` by applying the translation rules for ‘let’-expressions (line 31) and by recursing on the already translated lambda expression nodes (line 36).

A downside of this design is that `Source`, `Partial`, and `Target` are completely separate at the type level, but overlap at the semantic level. Both functions need to recurse through unchanged nodes and apply a type-transforming but semantics-preserving transformation (lines 24 and 36). As an example, in order to translate from `Source.Let` to `Partial.Let`: `ConvertBool` must destruct the existing `Source.Let` node (line 27), recurse on its children, and construct a new `Partial.Let` node. Having distinct intermediate types also imposes a fixed order when composing partial transformations; `ConvertBool` must be applied to `Source`, and `ConvertLet` must be applied to `Target`; otherwise the types would be incorrect (line 6). Both of these issues stem from the same problem, there is no way to abstract over commonalities because the types are fully distinct.

```

1 object Composable {
2   // ...
3
4   def compile(source: Source): Target = {
5     ConvertLet(ConvertBool(source))
6     // ConvertBool(ConvertLet(source))

```

```

7     //           ~~~~~
8     // ERROR: Found 'Source', required 'Partial'
9   }
10
11  def ConvertBool(source: Source): Partial = source match {
12    // Booleans
13    case S.True  => P.Lambda("x", P.Lambda("y", P.Var("x")))
14    case S.False => P.Lambda("x", P.Lambda("y", P.Var("y")))
15    case S.If(cond, con, alt) =>
16      val ifBody =
17        P.Apply(P.Apply(P.Var("p"), P.Var("a")), P.Var("b"))
18      val if_ = P.Lambda("p", P.Lambda("a", P.Lambda("b", ifBody)))
19      P.Apply(
20        P.Apply(P.Apply(if_, ConvertBool(cond)), ConvertBool(con)),
21        ConvertBool(alt)
22      )
23
24    // Recurse
25    case S.Var(name) => P.Var(name)
26    case S.Let(name, value, body) =>
27      P.Let(name, ConvertBool(value), ConvertBool(body))
28  }
29
30  def ConvertLet(partial: Partial): Target = partial match {
31    // Lets
32    case P.Var(name) => T.Var(name)
33    case P.Let(name, value, body) =>
34      T.Apply(T.Lambda(name, ConvertLet(body)), ConvertLet(value))
35
36    // Recurse
37    case P.Lambda(param, body) => T.Lambda(param, ConvertLet(body))
38    case P.Apply(fun, arg) =>
39      T.Apply(ConvertLet(fun), ConvertLet(arg))
40  }
41
42  // ...
43 }

```

Listing 2.5: Composable compiler

The implementations of `interpret`, `prettyS`, and `prettyT` are unchanged from the previous design (Listing 2.6, lines 17, 4, and 5). However, the implementation of a pretty-printer for the `Partial` language (line 7) requires code duplication. Printing the nodes representing variables and ‘let’-expressions (lines 8-10) is duplicated almost verbatim from `prettyS` (Listing 2.2, line 4), while printing the Lambda Calculus nodes (lines 12-13) is

duplicated from `prettyT` (Listing 2.2, line 14). The entire function contains no new unique code. Any changes to the logic in `prettyS` or `prettyT` would need to be carefully mirrored in `prettyP`. If a larger compiler requires multiple partial languages, `Partial1` to `partialN`, where each language progressively replaces nodes from `Source` with nodes from `Target`; each function `prettyP_N` would duplicate logic from every other pretty printing function! This is clearly not scalable.

```

1 object Partial {
2   // ...
3   // Pretty Printing
4   def prettyS(source: Source): String = Monolithic.prettyS(source)
5   def prettyT(target: Target): String = Monolithic.prettyT(target)
6
7   def prettyP(partial: Partial): String = partial match {
8     case P.Var(name) => name
9     case P.Let(name, value, body) =>
10      s"(let ($name ${prettyP(value)}) ${prettyP(body)})"
11
12     case P.Lambda(param, body) => s"(lambda ($param) ${prettyP(body)})"
13     case P.Apply(fun, arg)      => s"(${prettyP(fun)} ${prettyP(arg)})"
14   }
15
16   // Interpreting
17   def interpret(target: Target): Target = Monolithic.interpret(target)
18 }

```

Listing 2.6: Pretty-printer for Source, Partial, and Target

The composable compiler retains some of the benefits of the monolithic compiler: every function is fully type-safe, and every case is fully handled. It enables composing different partial translations to form a complete translation, but it does so at the cost of quadratic code duplication.

2.3 Uni-Typed Compiler

The uni-typed compiler (Listing 2.7) is an alternative design that aims to reduce code duplication, while allowing partial transformations to be composed together. It does this by consolidating the `Source` and `Target` into a single language: `Lang` (line 2). `Lang` contains all the nodes in the `Source` language (lines 3-10), as well as all the nodes in the `Target` language (lines 9-14). Since `Lang` is represented as a single type, each node can contain any other `Lang` node, without distinguishing between `Source` and `Target`.

```

1 object UniTyped {
2   enum Lang {
3     // Source
4     case True
5     case False
6     case If(cond: Lang, con: Lang, alt: Lang)
7     case Let(name: String, value: Lang, body: Lang)
8
9     // Source & Target
10    case Var(name: String)
11
12    // Target
13    case Lambda(param: String, body: Lang)
14    case Apply(fun: Lang, arg: Lang)
15
16    // ...
17  }
18
19  type Source = Lang
20  type Target = Lang
21
22  // ...
23 }

```

Listing 2.7: Lang language

One of the benefits of representing the languages as a single type is that it is possible to abstract over common behaviour like tree transformations (Listing 2.8). The function `mapChildren` (line 5) accepts a function of type `Lang => Lang` and applies it non-recursively to each node’s children (lines 5-12). The function `transform` (line 15) accepts a function `fn` and ties the recursive knot by applying it to the result of calling `mapChildren` with itself (line 16). In essence, this line will recurse to the leaf nodes of the node, apply `fn` to them, and progressively rebuild the node tree while applying `fn` at each level. The `transform` function can be used to perform bottom-up tree-transformations on a `Lang` node, and `mapChildren` can provide the programmer with extra control for cases when a different traversal pattern is required. An unfortunate side effect of using a single type to represent the tree is that if the developer forgets to convert a child node in `mapChildren`, the type checker will not raise an error. Because both the input and output of the `fn` function is `Lang`, applying it or not makes no difference to the type checker.

```

1 object UniTyped {
2   enum Lang {
3     // ...
4

```

```

5     def mapChildren(fn: Lang => Lang): Lang = this match {
6         case True                => True
7         case False               => False
8         case If(cond, con, alt)  => If(fn(cond), fn(con), fn(alt))
9         case Let(name, value, body) => Let(name, fn(value), fn(body))
10        case Var(name)           => Var(name)
11        case Lambda(param, body) => Lambda(param, fn(body))
12        case Apply(fun, arg)     => Apply(fn(fun), fn(arg))
13    }
14
15    def transform(fn: Lang => Lang): Lang =
16        fn(this.mapChildren(child => child.transform(fn)))
17    }
18    // ...
19 }

```

Listing 2.8: transform function

The uni-typed compiler decomposes its `compile` function (Listing 2.9, line 4) into separate functions similar to the composable compiler. Unlike the composable compiler, however, this implementation of `ConvertBool` and `ConvertLet` (lines 14 and 31) makes use of the `transform` function. The `ConvertBool` function only applies the translation rules from boolean expressions to Lambda Calculus explicitly for a single layer of the tree (lines 14-25), and unchanged nodes are returned as is, without the need to deconstruct and reconstruct them (line 28). Notably, `ConvertBool` is not recursive; all the recursion is provided by `transform`. `ConvertLet` uses the same pattern to translate ‘let’-expressions to Lambda Calculus (lines 31-37). Unlike the composable compiler, this pattern has no code duplication, and affords the reuse of the code responsible for recursively transforming the tree. The uni-typed compiler also allows the developer to compose the transformations in different orders (lines 5 and 7). The flexibility provided by the uni-typed compiler has the cost of making the type checker much less helpful for catching mistakes. In addition to being able to re-order the different transformations, the developer can omit an entire transformation and still have the code type check, or potentially apply the same transformation more than once (lines 9 and 10).

```

1 object UniTyped {
2     // ...
3
4     def compile(source: Lang): Lang = {
5         val result = ConvertLet(ConvertBool(source))
6         // OK: Also valid
7         // val _ = ConvertBool(ConvertLet(source))
8         // type checks but is incorrect

```

```

9     // val _ = ConvertLet(source)
10    // val _ = ConvertBool(source)
11    result
12  }
13
14  def ConvertBool(ir: Lang): Lang = ir.transform {
15    case Lang.True  => Lang.Lambda("x", Lang.Lambda("y", Lang.Var("x")))
16    case Lang.False => Lang.Lambda("x", Lang.Lambda("y", Lang.Var("y")))
17    case Lang.If(cond, con, alt) =>
18      val ifBody = Lang.Apply(
19        Lang.Apply(Lang.Var("p"), Lang.Var("a")), Lang.Var("b"))
20      val if_ =
21        Lang.Lambda("p", Lang.Lambda("a", Lang.Lambda("b", ifBody)))
22      Lang.Apply(
23        Lang.Apply(Lang.Apply(if_, cond), con),
24        alt
25      )
26
27    // Unchanged
28    case other => other
29  }
30
31  def ConvertLet(ir: Lang): Lang = ir.transform {
32    case Lang.Var(name) => Lang.Var(name)
33    case Lang.Let(name, value, body) =>
34      Lang.Apply(Lang.Lambda(name, body), value)
35
36    // Unchanged
37    case other => other
38  }
39
40  // ...
41 }

```

Listing 2.9: Untyped compiler

The implementation of `interpret` (Listing 2.10) further exemplifies the lack of type-safety. The function follows the same core logic as the monolithic and composable compilers: the outer function `interpret` (line 4) is not recursive, the inner function `go` (line 6) is recursive, and the outer function calls the inner function to perform the computation (line 13). The major difference between this implementation and the previous ones is that `go` is no longer exhaustive. `go` explicitly handles the cases for `Var`, `Lambda`, and `Apply` (lines 7-9), and handles any other node by raising a runtime error (line 10). The additional fallback case is necessary because the type checker cannot guarantee that the `Lang` has

been fully translated from `Source` to `Target`. Representing both languages under a single datatype loses information.

Pretty-printing can be implemented as a single recursive function without any duplication. The implementation is omitted for brevity.

```
1 object UnTyped {
2   // ...
3
4   def interpret(ir: Lang): Lang = {
5
6     def go(exp: Lang, ctx: Ctx): Lang = exp match {
7       case Lang.Var(name)           => /* ... */
8       case Lang.Lambda(param, body) => /* ... */
9       case Lang.Apply(fun, arg)     => /* ... */
10      case other => throw Exception(s"Unexpected node: $other")
11    }
12
13    go(ir, Map.empty)
14  }
15 }
```

Listing 2.10: evaluate function

The uni-typed compiler serves to contrast with the composable compiler. Both aim to achieve the ability to decompose the large translation step from the monolithic design into separate smaller steps. The composable compiler achieves this by introducing extra types, at the cost of more boilerplate and code duplication. The uni-typed compiler achieves a slim code base with more code reuse, at the cost of losing type-safety.

2.4 Phase-indexed compiler

A phase-indexed compiler attempts to retain both the use of a single datatype to represent `Source` and `Target` from the uni-typed compiler, and the ability to distinguish between `Source` and `Target` from the composable compiler. Despite sounding like an oxymoron, it accomplishes this by using an advanced type system feature called *Generalized Algebraic Data Types* (GADTs) [6] present in Scala and other languages. GADTs allow the phase-indexed compiler to encode which phase each type of node belongs to.

The phase-indexed compiler first defines the groups of nodes that are meaningful to it (Listing 2.11, lines 2-5). The groups are: `PBool` for representing boolean expressions,

`PLetOnly` for representing ‘let’-expressions (but not variables), `PLambdaOnly` for representing lambda abstractions and applications (but not variables), and `PVar` for representing variables reads. Two type aliases, `PLet` and `PLambda` (lines 7 and 8), are also provided. Separating `PVar` into its own group allows the node to be used together with ‘let’ and lambda expressions without otherwise combining the other nodes present in these groups.

The datatype `Lang` (line 10) represents both the `Source` and `Target` languages. Unlike the `Lang` type for the uni-typed compiler (Section 2.3), this representation contains a type parameter `P` to distinguish which nodes may be present in the tree. Each of the nodes representing boolean expressions (lines 13-16) must extend `Lang` with `PBool` to indicate that they contain boolean-expression nodes. Non-terminal nodes such as `If` must extend `Lang` with `P | PBool` to represent that they contain nodes of phase `P` in addition to nodes of phase `PBool`. Nodes for ‘let’ and lambda expressions must similarly extend `Lang` with their respective phases `PLet` and `PLambda` (lines 19, 27, and 28). The node `Var` (line 23) is a special case as it may be present in either ‘let’-expressions or lambda expressions; as such, it is given a unique phase tag `PVar`, which maybe be used as part of `PLet` or `PLambda` (lines 7 and 8). For convenience, type aliases for `Source` and `Target` which instantiate `Lang` with the appropriate phase indices may be provided (lines 31 and 32).

Variance and union types (“|”) are important to allow the composition of phase indices when composing nodes together. The `Lang` type is covariant in its phase tag `P` (line 10); which means that since `PBool` is a subtype of `PBool | PLet`, then `Lang[PBool]` is a subtype of `Lang[PBool | PLet]`. In practice, creating a `Let` node with a `True` node as one of its children will instantiate `Let[P]` to `Let[PBool]`, and by definition, `Let[PBool]` extends `Lang[PLet | PBool]` (line 20). The phase-indexed compiler uses this relation to specify which groups of nodes may be present in the tree; in this case only nodes of `PLet` or `PBool` are allowed. The phase tags may contain groups as specific or as general as the compiler writer requires, since they can always be composed together using type unions. For a more detailed discussion on subtyping, see Chapter 3.

```

1 object PhaseIndexed {
2   sealed abstract class PBool
3   sealed abstract class PLetOnly
4   sealed abstract class PLambdaOnly
5   sealed abstract class PVar
6
7   type PLet      = PLetOnly    | PVar
8   type PLambda  = PLambdaOnly | PVar
9
10  enum Lang[+P] {
11
12    // PBool

```

```

13     case True extends Lang[PBool]
14     case False extends Lang[PBool]
15     case If[P](cond: Lang[P], con: Lang[P], alt: Lang[P])
16         extends Lang[P | PBool]
17
18     // PLet
19     case Let[P](name: String, value: Lang[P], body: Lang[P])
20         extends Lang[P | PLet]
21
22     // PVar
23     case Var(name: String) extends Lang[PVar]
24
25     // PLambda
26     case Lambda[P](param: String, body: Lang[P])
27         extends Lang[P | PLambda]
28     case Apply[P](fun: Lang[P], arg: Lang[P]) extends Lang[P | PLambda]
29 }
30
31 type Source = Lang[PBool | PLet]
32 type Target = Lang[PLambda]
33
34 // ...
35 }

```

Listing 2.11: PhaseIndexed language

Using a single type `Lang` allows the phase-indexed compiler to implement a `pretty` function without duplicating any code (Listing 2.12, line 4). The function `pretty` accepts a node of type `Lang[Any]`, where `Any` is supertype of all types in Scala, meaning the function can accept a node of any phase. Similar to the monolithic compiler, the function simply recurses on each node’s children first, then translates the result to a `String` (lines 5-14).

The phase indices allow the `interpret` function to specify the subset of nodes it can accept (line 17). By specifying that it requires a node with type `Lang[PLambda]`, the caller can only pass it a tree that contains the nodes `Var`, `Lambda`, and `Apply`. Attempting to call `interpret` with a tree that contains other nodes, either as a root or one of its children, would result in a compile-time type error. As a result of this, the inner function `go` only needs to handle the supported cases (lines 19-22) and is still type-safe. The type checker understands this, and does not require a fallback case to guarantee exhaustivity (line 24).

```

1 object PhaseIndexed {
2   // ...
3

```

```

4 def pretty(ir: Lang[Any]): String = ir match {
5   case Lang.True => "true"
6   case Lang.False => "false"
7   case Lang.If(cond, con, alt) =>
8     s"(if ${pretty(cond)} ${pretty(con)} ${pretty(alt)})"
9   case Lang.Var(name) => name
10  case Lang.Let(name, value, body) =>
11    s"(let ($name ${pretty(value)}) ${pretty(body)})"
12  case Lang.Lambda(param, body) =>
13    s"(lambda ($param) ${pretty(body)})"
14  case Lang.Apply(fun, arg) => s"(${pretty(fun)} ${pretty(arg)})"
15 }
16
17 def interpret(ir: Lang[PLambda]): Lang[PLambda] = {
18
19   def go(ir: Lang[PLambda], ctx: Ctx): Lang[PLambda] = ir match {
20     case Lang.Var(name) => /* ... */
21     case Lang.Lambda(param, body) => /* ... */
22     case Lang.Apply(fun, arg) => /* ... */
23     // Exhaustiveness guaranteed
24     // case other => throw Exception(s"Unexpected node: $other")
25   }
26
27   go(ir, Map.empty)
28 }
29 }

```

Listing 2.12: PhaseIndexed evaluation

The ability to distinguish between the different nodes in the tree allows the implementation of `compile` (Listing 2.13, line 4) to be composed from smaller partial transformation functions `ConvertLet` and `ConvertBool`. The `ConvertLet` function (line 14), accepts a node of type `Lang[P | PLet]` and returns a node of type `Lang[P | PLambda]` where `P` is a polymorphic type variable. A node of type `Lang[P | PLet]` may contain `Let` and `Var` nodes, in addition to the nodes which may be present in phase `P`, while a node of type `Lang[P | PLambda]` may contain `Lambda`, `Apply`, and `Var` nodes, in addition to nodes in phase `P`. The type of `ConvertLet` specifies that it will remove nodes of phase `PLet`, potentially add nodes of phase `PLambda`, and not change nodes of phase `P`. Crucially, it is generic in whether the tree contains nodes of phase `PBool`. The `ConvertBool` function (line 31) specifies a similar type, but swapping `PLet` for `PBool`. Because both `ConvertLet` and `ConvertBool` are polymorphic, the `compile` function may compose them in any order (lines 5 and 7). However, the type signature for `ConvertLet` does not specify that it removes any `PBool` nodes; if it is not composed with `ConvertBool`, the type checker will

statically know that it returns a node containing not only `PLambda` nodes, but `PBool` nodes as well (line 9). The inverse is true for `ConvertBool` (line 10).

The implementation of `ConvertLet` must recursively apply the translation rules from ‘let’-expressions to lambda expressions (lines 15-18). It must also recurse on the boolean expression nodes (lines 21-24) in order to translate their child nodes. Since `ConvertLet` allows its argument to be polymorphic, it must handle any potential node as an input, including lambda expressions (lines 25-28). The `ConvertBool` function follows a similar implementation pattern (line 31).

```

1 object PhaseIndexed {
2   // ...
3
4   def compile(source: Lang[PBool | PLet]): Lang[PLambda] = {
5     val result = ConvertLet(ConvertBool(source))
6     // OK: Also valid
7     // val _: Lang[PLambda] = ConvertBool(ConvertLet(source))
8     // ERROR:
9     // val _: Lang[PLambda] = ConvertLet(source)
10    // val _: Lang[PLambda] = ConvertBool(source)
11    result
12  }
13
14  def ConvertLet[P](ir: Lang[P | PLet]): Lang[P | PLambda] = ir match {
15    case Lang.Let(name, value, body) =>
16      Lang.Apply(
17        Lang.Lambda(name, ConvertLet[P](body)), ConvertLet(value))
18    case Lang.Var(name) => Lang.Var(name)
19
20    // Recurse
21    case Lang.True => Lang.True
22    case Lang.False => Lang.False
23    case Lang.If(cond, con, alt) =>
24      Lang.If(ConvertLet[P](cond), ConvertLet(con), ConvertLet(alt))
25    case Lang.Lambda(param, body) =>
26      Lang.Lambda(param, ConvertLet[P](body))
27    case Lang.Apply(fun, arg) =>
28      Lang.Apply(ConvertLet[P](fun), ConvertLet(arg))
29  }
30
31  def ConvertBool[P](ir: Lang[P | PBool]): Lang[P | PLambda] = // ...
32 }

```

Listing 2.13: PhaseIndexed compiler

The phase-indexed compiler reduces some of the boilerplate from the composable compiler, especially when writing functions on all nodes like the function `pretty`; however, it introduces even more boilerplate when writing polymorphic functions like `ConvertLet` and `ConvertBool`. Ideally, `Lang` should support `mapChildren` and `transform` operations (Listing 2.14, lines 7 and 8) like the uni-typed compiler, but it is impossible to create a useful type for these functions. If they accept a function of type `Lang[P] => Lang[P]`, then they cannot change the phase of the tree by adding or removing nodes. If they accept a function `fn` of type `Lang[P1] => Lang[P2]`, for some phases `P1` and `P2`, then `fn` itself must recursively implement a phase transformation, in essence making `transform` an identity function. The root issue is that `Lang` cannot represent a tree that is partially transformed, with some nodes from one phase, and other nodes from another.

```

1 object PhaseIndexed {
2   // ...
3
4   enum Lang[+P] {
5     // ...
6
7     // def mapChildren(fn: Lang[?] => Lang[?]): Lang[?] = ???
8     // def transform(fn: Lang[?] => Lang[?]): Lang[?] = ???
9   }
10
11 // ...
12 }

```

Listing 2.14: PhaseIndexed language

2.5 Precise-typed compiler

The precise-typed compiler is a design that retains the type safety of the composable and phase-indexed compilers, while including the abstractions and code reuse from the uni-typed compiler. Like the phase-indexed compiler, this design includes tags for the various phases of the node tree (Listing 2.15, lines 3-9). The major difference in representing the languages is that it uses a type, `LangK[+PSelf, +PChild]` (line 12), with two type variables. The first type variable, `PSelf`, represents the phase of the root node of the tree, while the second type variable, `PChild`, represents the phases of all child nodes of the tree and subtrees. The ability to distinguish between the phase of the root and child nodes adds a large amount of flexibility compared to the phase-indexed compiler, while retaining all of its type-safety. For the common use case in which both the root and child nodes are

of the same phase, a type alias `Lang[+P]` is provided (line 11). All of the type variables presented must be covariant for the same reasons as the phase-indexed compiler.

Each node in the precise-typed compiler should extend the `LangK` type differently depending on whether it is a leaf or branch node. Both kinds of nodes should extend `LangK` with their own phase tag for `PSelf`; the difference arises on which phase tag to extend `PChild` with. Leaf nodes have no child nodes, and should extend `LangK` with `Nothing` for `PChild`, while branch nodes have children, and should extend `LangK` with a polymorphic type `P` for `PChild`. The `True` node is an example of a leaf node (line 15); its own phase is `PBool` and it contains no children. The `If[+P]` node is an example of a branch node (line 17); its own phase is also `PBool`, and its children have phase `P`. The remaining leaf nodes are defined in the same way in lines 16 and 25, and the remaining branch nodes are defined in lines 17, 21, 25, 28, and 30. The `Source` and `Target` languages can be represented as type aliases of `Lang` with the appropriate phase tags (lines 35 and 36).

```

1 object PreciseTyped {
2
3   sealed abstract class PBool
4   sealed abstract class PVar
5   sealed abstract class PLetOnly
6   sealed abstract class PLambdaOnly
7
8   type PLet      = PLetOnly      | PVar
9   type PLambda  = PLambdaOnly   | PVar
10
11  type Lang[+P] = LangK[P, P]
12  enum LangK[+PSelf, +PChild] {
13
14    // PBool
15    case True extends LangK[PBool, Nothing]
16    case False extends LangK[PBool, Nothing]
17    case If[+P](cond: Lang[P], con: Lang[P], alt: Lang[P])
18      extends LangK[PBool, P]
19
20    // PLet
21    case Let[+P](name: String, value: Lang[P], body: Lang[P])
22      extends LangK[PLet, P]
23
24    // PVar
25    case Var(name: String) extends LangK[PVar, Nothing]
26
27    // PLambda
28    case Lambda[+P](name: String, body: Lang[P])
29      extends LangK[PLambda, P]

```

```

30     case Apply[+P](fun: Lang[P], arg: Lang[P]) extends LangK[PLambda, P]
31
32     // ...
33 }
34
35 type Source = Lang[PBool | PLet]
36 type Target = Lang[PLambda]
37
38 // ...
39 }

```

Listing 2.15: `PreciseTyped` language

The ability to express that the root and child nodes of a tree may be part of different phases allows the precise-typed compiler to express the `mapChildren` function correctly (Listing 2.16, line 5). `mapChildren` simply applies the given function `fn` to each node’s children, giving the overall function the type:

$$(\text{LangK}[\text{PSelf}, P1], \text{Lang}[P1] \Rightarrow \text{Lang}[P2]) \Rightarrow \text{LangK}[\text{PSelf}, P2]$$

The first argument, `LangK[PSelf, P1]`, is the input node of phase `PSelf` with children of phase `P1`. The second argument, `Lang[P1] => Lang[P2]`, is a function from `Lang[P1]` to `Lang[P2]`. The return type, `LangK[PSelf, P2]`, is the output node of phase `PSelf` with children of phase `P2`.

The function `transform` is implemented as an extension method (line 24) because it only applies to trees with the same phase for root and children. The function has the same implementation as the `transform` function for the uni-typed compiler; it first maps the children of the node by applying them to itself (line 27), then it applies the result to the given function `fn` (line 28). However, the function has a different type:

$$(\text{Lang}[P1], \text{LangK}[P1, P2] \Rightarrow \text{LangK}[P2, P2]) \Rightarrow \text{Lang}[P2]$$

The given function `fn` accepts a node of phase `P1` with children of phase `P2`, and returns a node of phase `P2`. Once `transform` is given a node, `node`, of type `LangK[P1, P1]`, it needs to transform the children of `node` from `Lang[P1]` to `Lang[P2]`, before passing the result to `fn`. It does this by mapping each child, `child`, of type `Lang[P1]`, with `transform` and `fn` (line 27). The recursion reaches its base case when `child` is a leaf node, which our invariants demand have type `LangK[P1, Nothing]`. Since `LangK` is covariant in its second type argument `PChild`, `LangK[P1, Nothing]` is a subtype of `LangK[P1, P2]`, making it a valid argument for `fn`.

```

1 object PreciseTyped {
2   enum LangK[+PSelf, +PChild] {

```

```

3 // ...
4
5 def mapChildren[PChild2](
6   fn: Lang[PChild] => Lang[PChild2]
7 ): LangK[PSelf, PChild2] = this match {
8   // PBool
9   case True           => True
10  case False          => False
11  case If(cond, con, alt) => If(fn(cond), fn(con), fn(alt))
12  // PLet
13  case Let(name, value, body) => Let(name, fn(value), fn(body))
14  // PLet & PLambda
15  case Var(name) => Var(name)
16  // PLambda
17  case Lambda(name, body) => Lambda(name, fn(body))
18  case Apply(fun, arg)    => Apply(fn(fun), fn(arg))
19 }
20
21 }
22
23 extension [P1](node: Lang[P1]) {
24   def transform[P2](
25     fn: LangK[P1, P2] => LangK[P2, P2]
26   ): Lang[P2] = {
27     val mappedChildren = node.mapChildren(child => child.transform(fn))
28     fn(mappedChildren)
29   }
30 }
31
32 // ...
33 }

```

Listing 2.16: PreciseTyped transform

The implementation of `compile` (Listing 2.17, line 3) is once again decomposed into `ConvertLet` and `ConvertBool`. Similar to the phase-indexed compiler, this design allows for `compile` to be composed together in a non-fixed order (lines 4 and 6), while statically requiring both passes to be applied (line 8 and 9).

The implementation of `ConvertBool` makes use of the new `transform` function (line 13) in a similar way to that of the uni-typed compiler. `ConvertBool` applies the translation rules from boolean expressions to lambda expressions for each layer of the tree (lines 16-24). The children of the `If` node have already been converted to lambda expressions by `transform` (line 18), so there is no need to explicitly recurse into them. An additional difference is that the remaining case `other` (line 26) is inferred to have type

Lang[P | PLambda], allowing transform to change the phase of the tree. Inferring this type requires a change to the Scala compiler discussed in Chapter 3. Informally, this type of other can be deduced as follows: the transform function requires a function fn of type LangK[P | PBool, P | PLambda] => LangK[P | PLambda, P | PLambda]; the only cases of LangK that extend PBool are True, False, and If; once these have been matched, the remaining case other cannot possibly extend LangK[PBool, P | PLambda]; therefore other must extend LangK[P, P | PLambda] only, which is a valid subtype of the expected node type LangK[P | PLambda, P | PLambda]. The implementation of ConvertLet follows the same pattern (line 29).

```

1 object PreciseTyped {
2   // ...
3   def compile(source: Lang[PBool | PLet]): Lang[PLambda] = {
4     val result = ConvertLet(ConvertBool(source))
5     // OK: Also valid
6     // val _ = ConvertBool(ConvertLet(source))
7     // ERROR:
8     // val _: Lang[PLambda] = ConvertLet(source)
9     // val _: Lang[PLambda] = ConvertBool(source)
10    result
11  }
12
13  def ConvertBool[P](ir: Lang[P | PBool]): Lang[P | PLambda] =
14    import LangK._
15    ir.transform {
16      case True => Lambda("x", Lambda("y", Var("x")))
17      case False => Lambda("x", Lambda("y", Var("y")))
18      // If(cond: Lang[P | PLambda],
19      //     con: Lang[P | PLambda],
20      //     alt: Lang[P | PLambda])
21      case If(cond, con, alt) =>
22        val ifBody = Apply(Apply(Var("p"), Var("a")), Var("b"))
23        val if_ = Lambda("p", Lambda("a", Lambda("b", ifBody)))
24        Apply(Apply(Apply(if_, cond), con), alt)
25
26      case other => other // : Lang[P | PLambda]
27    }
28
29  def ConvertLet[P](ir: Lang[P | PLet]): Lang[P | PLambda] =
30    import LangK._
31    ir.transform {
32      case Let(name, value, body) => Apply(Lambda(name, body), value)
33      case Var(name) => Var(name)
34      case other => other // : Lang[P | PLambda]

```

```

35     }
36     // ...
37 }

```

Listing 2.17: `PreciseTyped` compiler

The precise-typed compiler allows the `pretty` function (Listing 2.18, line 3) to be implemented as a single function. It accepts a node `ir` of any phase `Any`, and recurses through each case, applying `Pretty` to each child node (lines 4-13). The implementation of `interpret` (line 16) uses an inner recursive function `go` (line 18) to perform the reduction rules. The function `go` is type safe, as specifying the only input phase as `PLambda` guarantees exhaustivity when pattern matching (line 23).

```

1 object PreciseTyped {
2     // ...
3     def pretty(ir: Lang[Any]): String = ir match {
4         case LangK.True => "true"
5         case LangK.False => "false"
6         case LangK.If(cond, con, alt) =>
7             s"(if ${pretty(cond)} ${pretty(con)} ${pretty(alt)})"
8         case LangK.Var(name) => name
9         case LangK.Let(name, value, body) =>
10            s"(let ($name ${pretty(value)}) ${pretty(body)})"
11        case LangK.Lambda(param, body) =>
12            s"(lambda ($param) ${pretty(body)})"
13        case LangK.Apply(fun, arg) => s"(${pretty(fun)} ${pretty(arg)})"
14    }
15
16    def interpret(ir: Lang[PLambda]): Lang[PLambda] = {
17
18        def go(exp: Lang[PLambda], ctx: Ctx): Lang[PLambda] = exp match {
19            case LangK.Var(name) => /* ... */
20            case LangK.Lambda(param, body) => /* ... */
21            case LangK.Apply(fun, arg) => /* ... */
22            // Exhaustiveness guaranteed
23            // case other => throw Exception(s"Unexpected node: $other")
24        }
25
26        go(ir, Map.empty)
27    }
28 }

```

Listing 2.18: `PreciseTyped` pretty printer and interpreter

The design of the precise-typed compiler allows the developer to increase type-safety without increasing code duplication. The design encodes the initial language `Source`, the

final language **Target**, and the intermediate steps of compilation within a single language **LangK**. The ability to distinguish between the phase of the current root node and the phase of its children allows this design to abstract over tree transformations without compromising on type-safety and exhaustivity. The use of a single datatype allows helper functions such as pretty-printing to work on any combination of phases in the language.

Chapter 3

Narrowing in Pattern Matching

In order to correctly elaborate the precisely-typed tree transformations introduced in Section 2.5, it is necessary to infer more precise types within ‘match’-expressions. This chapter introduces the concept of type narrowing in pattern matching. Section 3.1 motivates the need for more precise type inference. Section 3.2 presents an overview of the algorithm used to narrow types in ‘match’-expressions. Section 3.3 presents type decomposition as a technique to decompose a type into a combination of its subtypes. Section 3.4 presents the concept of narrowing a decomposed type. Section 3.5 presents the concept of complete and incomplete patterns as a necessary condition to uncover type narrowing relations within ‘match’-expressions. Section 3.6 presents a subtyping rule for disjoint type intersections.

3.1 Motivation

Types often serve as an approximation of what values a term may have at runtime [4]. The type checker must determine that the term may have any value within a given type, but definitely not one outside the given type. It is possible, however, that the type checker infers a type more broad than it needs to. In these situations, the programmer is left to resort to runtime assertions or type casts to convince the type checker of what they already know.

```
1 sealed abstract class P
2 case object C1 extends P
3 case object C2 extends P
4
5 def sToString(s: P): String = s match {
```

```

6   case C1 => "C1"
7   case c2 => c2ToString(c2)
8 }
9
10 def c2ToString(c2: C2.type): String = "C2"

```

Listing 3.1: Example program with a ‘match’-expression

Consider the program in Listing 3.1. The type `P` (line 1) is declared to be inhabited by two values: `C1` and `C2` (lines 2 and 3). The function `sToString` (line 5) converts the input of type `P` to a `String` by pattern matching on the input `s` (line 5). If the value is `C1`, then it returns ‘‘C1’’ (line 6); otherwise it falls through to the default case (line 7) where it calls `c2ToString`. The function `c2ToString` simply converts a value of type `C2.type` to the `String` ‘‘C2’’ (line 10).

Unfortunately this program fails to type check. Most languages (including Scala) over-approximate the type of `c2` (line 7) to be `P`. In ‘match’-expressions, the type of every ‘case’-pattern, `C1` and `c2`, is the same as the scrutinee expression `s`. This causes a type mismatch when calling `c2ToString(c2)`, as `c2ToString` expects an argument of type `C2.type`. Intuitively this program is correct and would run without getting stuck. Since each ‘case’-branch is run in top-down order, if the `c2` branch is run (line 7), it is because `s` failed to match on `C1` (line 6). If `s` failed to match on `C1`, the only possible value it can be is `C2` and passing it to `c2ToString` is safe. It is therefore safe for the type of `c2` to be inferred as `C2.type` and to allow the call to `c2ToString` to type check. This is an example of type narrowing [2] over ‘match’-expressions, which we call match narrowing.

It is possible to provide an explicitly typed pattern to guide the type checker:

```

1 def sToString(s: P): String = s match {
2   case C1           => "C1"
3   case c2: C2.type => c2ToString(c2)
4 }

```

By adding the typed pattern in line 3, the type checker assigns the type of `c2` to be `C2.type`, and the program type checks. In Scala this is compiled to a potentially fallible runtime downcast; an exhaustiveness checker is run at a later phase to determine if the cast always succeeds. Additionally this option can quickly become unwieldy if the pattern `c2` has a more complicated type than `C2.type`. If `P` had the values `C1` to `Cn`, the type ascription would need to be a union of types `C2.type` to `Cn.type`. This pattern comes up when writing tree transformations in a precisely-typed compiler when program trees can have numerous nodes (as shown in Section 2.5). Providing explicit type patterns does not scale to larger types; enabling the compiler to infer such precise types is desirable.

3.2 Overview

To implement type narrowing over ‘match’-expressions, a basic understanding of how the Scala compiler currently elaborates ‘match’-expressions is required. Listing 3.2 presents a simplification of the algorithm currently used. The function `typedCases` (line 1) receives a list of untyped ‘case’-branches `cases` (line 2), the type of the scrutinee expression `selType` (line 3), and the prototype for the resulting expression `pt` (line 4). The Scala type checker uses a version of colored local type inference which elaborates an untyped expression and a prototype to a typed expression [27]. `typedCases` then returns the list of typed ‘case’-branches (line 4) to be used in elaborating the complete ‘match’-expression (line 13). Within the body of `typedCases`, the function simply maps over the ‘case’-branches and calls `typedCase` to elaborate them (line 7). Notably it passes the unchanged `selType` to be used as the type of the ‘case’-pattern; it is this behaviour that imposes that all ‘case’-patterns have the same type as the scrutinee expression.

```
1 def typedCases(  
2   cases: List[untpd.CaseDef],  
3   selType: Type,  
4   pt: Type): List[CaseDef] =  
5 {  
6   cases.map { cas ->  
7     typedCase(cas, selType, pt)  
8   }  
9 }  
10  
11 def typedMatch(tree: untpd.Match, pt: Type): Match =  
12   // ...  
13   val cases = typedCases(tree.cases, selType, pt)  
14   // ...  
15  
16 def typedCase(cas: untpd.CaseDef, selType: Type, pt: Type): CaseDef =  
17   /* ... */
```

Listing 3.2: Existing algorithm to elaborate ‘case’-branches, simplified

The key insight for match narrowing to work is that more information about the scrutinee expression is discovered as a ‘match’-expression is elaborated. Given an expression such as Listing 3.3, the scrutinee expression is initially of type `P` (line 1). If `s` matches `C1` (line 2), then `c1` will have type `C1.type` within that branch. If `s` reaches the `c2` branch (line 3), it means it cannot be of type `C1.type`; thus the value `c2` must belong to type `P` but not type `C1.type`. If types can be thought of as sets of possible values [4], it would be useful to have some subtraction operation ‘-’ on types. Unfortunately the domain of Scala

types does not have such an operation, so it is necessary to approximate it.

```
1 s: P match {
2   case c1: C1.type => // c1: C1.type
3   case c2           => // c2: P - C1.type
4 }
```

Listing 3.3: Sample ‘match’-expression

The modified algorithm to narrow types over ‘match’-expressions makes use of an approximation of type subtraction (Listing 3.4). The function `decompose` (line 6) transforms a given type `P` into a union of its variants: `C1 | ... | Cn`. The function `narrowType` (line 11) can then approximate type subtraction by removing specific variants from the union: `(C1 | ... | Cn) - C1 = (C2 | ... | Cn)`. Additional functions `isComplete` (line 10) and `And` (line 8) are required to ensure the correct behaviour. `isComplete` determines if a ‘case’-pattern will match on all values of its type; this not always the case due to ‘if’-guards and sub-patterns. It is only possible to narrow the remaining type if `isComplete(cas)` is true (Section 3.5). `And` returns a type intersection of both its arguments; this is required to ensure that the narrowed type is always a subtype of the original wide type. Notably, each ‘case’-branch is elaborated with the narrowed type (line 9), allowing the type information discovered in earlier branches to propagate to later ones. These concepts will be further explained in the following sections.

```
1 def typedCases(
2   cases: List[untpd.CaseDef],
3   selType: Type,
4   pt: Type): List[CaseDef] =
5 {
6   var remainingType = decompose(selType)
7   cases.map { cas ->
8     val narrowedType = And(selType, remainingType)
9     val cas = typedCase(cas, narrowedType, pt)
10    if isComplete(narrowedType, cas) then
11      remainingType = narrowType(whole= remainingType, part= cas.pat.tp)
12    cas
13  }
14 }
```

Listing 3.4: Modified algorithm to elaborate ‘case’-branches

3.3 Type Decomposition

Type decomposition refers to the ability to decompose a class hierarchy into a union of its subclasses. Since this requires knowledge of all subclasses, this can only be done for sealed hierarchies. A sealed hierarchy is a class hierarchy [8] that is only extended by a finite set of subclasses known at compile time; in Scala this is guaranteed by restricting all subclasses to be defined in the same file as its sealed superclass. Decomposing a type makes type narrowing more useful because it creates type unions and allows for an approximation of type subtraction.

Given a type `P` and the decomposition operator `decompose`, it is necessary that `decompose(P)` is a subtype of `P`, but also that `P` is a subtype of `decompose(P)`. This is required to ensure that type decomposition does not lose any type information. Given the following hierarchy:

```
1 sealed abstract class P
2 case object C1 extends P
3 case object C2 extends P
```

The type `P` can be decomposed into `(C1.type | C2.type)`, but this is not enough to preserve type equality with `P`. The subtyping rules [1, 29] for type unions state that $(A | B) <: C$ if $A <: C$ or $B <: C$. Since both `C1.type` and `C2.type` are subtypes of `P`, `(C1.type | C2.type)` is a subtype of `P`. However this relation is not guaranteed in the reverse: `P` is not a subtype of `(C1.type | C2.type)`. To achieve this, it is necessary to add a type intersection to the resulting type. The subtyping rules for type intersections state that $A <: (B \& C)$ if $A <: B$ or $A <: C$. Since `P` is a subtype of itself via reflexivity, `P` can be decomposed into `P & (C1.type | C2.type)` while maintaining the bidirectional subtyping relation. This pattern allows the decomposed type to be a subtype of the original type and allows for more precision when narrowing the type.

Type decomposition is partially implemented via the `decompose` function (Listing 3.5, line 1). This function only decomposes a type to a union of its children, leaving the final type intersection to be applied at use site for flexibility (Listing 3.4, line 8). `decompose` must get the children of a given type `tp` by calling the `childrenOf` function (line 2). Once the children are returned, it is simply a matter of creating a type union of all the children (line 3). If `tp` does not have a known set of children, `childrenOf` returns it in a singleton `List` and `decompose` returns the type unchanged.

The complexity resides in the `childrenOf` function (line 5). The function is recursive, and if the given type `tp` is sealed, it will return all leaf subclasses in the hierarchy. First it must determine if `tp` can be decomposed; it does this by calling a helper method

`isDecomposableToChildren` (line 6). This method is already present in the Scala compiler for the purposes of pattern match exhaustiveness checking. In essence, it checks if the given class is sealed and all its subclasses are named. If `tp` is not decomposable, the function returns a singleton `List` containing the original type (line 13). This happens when `tp` does not correspond to a class, such as when `tp` is a type variable, a union type, or an intersection type. If `tp` is decomposable, the function must collect all the direct child classes (line 7). If `children` only contains the initial type `tp`, then the recursion terminates (line 9), otherwise the function recurses on each child (line 11)

```

1 def decompose(tp: Type): Type =
2   val children = childrenOf(tp)
3   children.reduce((x, y) => Or(x, y))
4
5 def childrenOf(tp: Type): List[Type] =
6   if tp.isDecomposableToChildren then
7     val children = tp.children
8     if children.length == 1 && children.head == tp then
9       children
10    else
11      children.flatMap(child => childrenOf(child))
12 else
13   List(tp)

```

Listing 3.5: Simplified algorithm for type decomposition

3.4 Type Difference

Type difference is the operation to transform a general type into a more specific subtype. This serves as an approximation for type subtraction ‘-’ and is done by specifying part of the general type that the value cannot inhabit. Given types `A` and `B`, `A - B` is a subtype of `A`, but `A - B` is not a subtype of `B` as long as `A` is not `Bottom` and `B` is not `Top`.

The type difference operation can be expressed as: `NarrowType(Whole, Part) = Specific`; where `Whole`, `Part`, and `Specific` are types. Since Scala does not have a notion of type subtraction, this can only be expressed by removing alternatives from type unions ‘|’. In other words, `NarrowType(A | B, A) = B`; the general type `A | B` is narrowed by the part `A`, resulting in the specific type `B`. The intuition is: if type `A | B` is known not to be `A`, then it must be `B`. The complete rules are:

```

1 NarrowType(A, C) = Bottom if A <: C
2 NarrowType(A | B, C) = B if A <: C

```

```

3 NarrowType(A | B, C) = A      if B <: C
4 NarrowType(A | B, C) = NarrowType(A, C) | NarrowType(B, C)
5 NarrowType(A & B, C) = NarrowType(A, C) & NarrowType(B, C)
6 NarrowType(A,      _) = A

```

If `A` is narrowed by its supertype `C`, then the result is `Bottom`, the subtype of all types. If `A | B` is narrowed by a `C`, where `A` is a subtype of `C`, then the result is `B`. Narrowing a union or intersection (`&`) type distributes `NarrowType` over its members. Finally, if the type `A` is neither a union or intersection, and not a subtype of `C`, then the result is simply `A`.

```

1 def narrowType(whole: Type, part: Type): Type = whole match {
2   case _ if whole <: part                => defn.NothingType
3   case OrType(tp1, tp2) if tp1 <: part => narrowType(tp2, part)
4   case OrType(tp1, tp2) if tp2 <: part => narrowType(tp1, part)
5   // Recurse on both sides
6   case OrType(tp1, tp2) =>
7     Or(narrowType(tp1, part), narrowType(tp2, part))
8   case AndType(tp1, tp2) =>
9     And(narrowType(tp1, part), narrowType(tp2, part))
10  // Terminate
11  case _ => whole
12 }

```

Listing 3.6: Simplified algorithm for type narrowing

The function `narrowedType` (Listing 3.6) is a simplified version of the type difference algorithm implemented in the compiler; it narrows the type `whole` by the type `part` (line 1). If `whole` is a subtype of `part`, the function returns the `Nothing` type (line 2), which is the Scala representation of `Bottom`. If `whole` is an `OrType` and one of its alternatives is a subtype of `part`, then the result is the narrowing of the other alternative by `part` (lines 3 and 4). If `whole` is an `OrType` or an `AndType`, the function recurses on both its members (lines 6-9). The helper functions `Or` and `And` (lines 7 and 9) normalize the type expression to a consistent format within the compiler. Finally, if `whole` is any other type, it cannot be narrowed by `part` and is returned unchanged (line 11).

The existing pattern exhaustiveness checker contains elements of type decomposition and type difference through an abstraction called *space* [19]. The exhaustiveness checker projects both types and patterns into *spaces* in order to perform a subspace check, but *spaces* cannot be mapped back to types. In order to support match narrowing, the type checker was modified to implement type decomposition and type difference, reusing some of the logic and intuition from the exhaustiveness checker. The type checker implementation never converts types into *spaces* and therefore can use the resulting types to further elaborate the program.

The combination of type decomposition and type difference allows for an approximation of type subtraction in Scala. Recall the example program in Listing 3.1. Once the type P is decomposed into $P \& (C1.type \mid C2.type)$, narrowing it by $C1.type$ would yield the type $P \& C2.type$. This type would be a subtype of $C2.type$, allowing the call to `c2ToString(c2)` in line 7 to type check. It is now necessary to determine when it is valid to apply type difference to the remainder type.

3.5 Complete and Incomplete Patterns

Complete patterns are a necessary condition for the type of a scrutinee to be narrowed. Consider the following example:

```
1 enum Option { case None; case Some(value: List) }
2 enum List { case Nil; case Cons(x: Int, xs: List) }
3 (scrutinee : Option) match {
4   case Some(Cons(x, xs)) => /* ... */
5   case None              => /* ... */
6 }
```

Listing 3.7: An incomplete pattern

The type `Option` (line 1) is inhabited by `None` and `Some`. The type `List` (line 2) is inhabited by `Nil` and `Cons`. The term `scrutinee` (line 3) is inspected at runtime to determine if it is a `Some` or `None` value (lines 4 and 5). However, even though `Some` and `None` are the only possible values of `Option`, the match expression is not exhaustive. This is because the `Some(Cons(x, xs))` (line 4) is an incomplete pattern; the value `Some(Nil)` is not covered, which would cause a runtime crash. An incomplete pattern is a pattern that does not match all values of its subtype, either due to containing nested sub-patterns or ‘if’-guards. In this example the pattern `Some(Cons(x, xs))` has type `Some`, but does not match the value `Some(Nil)` even though it also has type `Some`. A pattern is complete if it is a variable pattern, or a data constructor where all sub-patterns are variable patterns. Because `Some(Cons(x, xs))` is incomplete, the type `Some` cannot be considered to be fully covered by the match expression. If the pattern were instead `Some(list)`, then it would be considered complete, as it would match on all values that are subtypes of `Some`.

Consider the following modification to Listing 3.7:

```
1 (scrutinee : Option) match {
2   case Some(list) => /* ... */
3   case rest      => /* ... */
```

4 }

Listing 3.8: A complete pattern

The only way for the `rest` branch (line 3) to run is if `scrutinee` does not match on the pattern `Some(list)` (line 2). Additionally, since the pattern `Some(list)` is complete, if `scrutinee` fails to match on it, then it cannot be a subtype of `Some`; therefore `rest` must be of type `Nil.type`! If a branch `N` has a complete pattern, then branch `N+1` can only run if the selector expression does not match on branch `N`; this means the runtime value of the expression is discovered to not be a subtype of the pattern of branch `N` and thus the type of the pattern of branch `N+1` can be narrowed by the type of the pattern of branch `N`.

```
1 def isComplete(narrowType: Type, cas: Tree): Boolean =
2   if !cas.guard.isEmpty then
3     return false
4   cas.pat match
5     // id
6     case id: Ident => true
7     // prefix . name
8     case Select(prefix, name) => true
9     // con(pats...)
10    case UnApply(con, _, pats) =>
11      (con.symbol.name != nme.unapplySeq)
12      && pats.forall(p => alwaysMatches(narrowedType, p))
13    case _ => alwaysMatches(narrowType, cas.pat)
14
15 def alwaysMatches(narrowType: Type, pat: Tree): Boolean = pat match
16   // id
17   case id: Ident => untpd.isVarPattern(id)
18   // id @ pat
19   case Bind(id, pat) => alwaysMatches(pat)
20   // pat : tpt
21   case Typed(pat, tpt) => (narrowType <: tpt.tpe) && alwaysMatches(pat)
22   case _ => false
```

Listing 3.9: Simplified algorithm for determining complete patterns

Two functions are needed to determine if a pattern is complete, `alwaysMatches` (Listing 3.9, line 15) and `isComplete` (line 1). The function `isComplete` determines if a given pattern is complete for a given type. In order to do so, `isComplete` may call `alwaysMatches` to determine if a sub-pattern always matches the given type. The distinction is that a constructor pattern may be complete, but it does not always match on expressions of a given type. To enforce this, `isComplete` is only called on the root of the pattern tree, while `alwaysMatches` may be called recursively on its subtrees.

The function `alwaysMatches` (line 15) determines if a pattern `pat` will always match an expression of a given type `narrowType`. If the pattern is an identifier (line 17), it will always match if it is a variable (as opposed to a data constructor with no arguments). If the pattern is a bind pattern like `id @ pat'`, it will always match if the sub-pattern `pat'` always matches (line 19). If the pattern is a typed pattern like `pat' : tpt`, it will always match if `narrowType` is a subtype of `tpt.tpe` and if `pat'` always matches (line 21). Otherwise the pattern `pat` does not always match an expression of the type `narrowType` (line 22).

The function `isComplete` presents a simplified algorithm to determine if a pattern is complete (line 1). It accepts the current narrowed type `narrowType` and the entire 'case'-branch `cas` (line 1). If a 'case'-branch has a non-empty 'if'-guard, then the pattern is not complete (line 2). Otherwise, it matches on the 'case'-pattern `cas.pat` (line 4). If the pattern is a variable pattern or a single data constructor, then the pattern is complete (line 6). If the pattern is a qualified data constructor, then it is also complete (line 8). If the pattern is a data constructor with sub-patterns, then it is only complete if all the sub-patterns will always match (line 10). A special case must be added if the data constructor is `Seq`, which allows for a variable number of sub-patterns and cannot be determined to be complete (line 11). Any other pattern is only complete if it will always match an expression of the given `narrowType` (line 13); this work is delegated to the function `alwaysMatches`.

Combined with type decomposition and type difference, complete patterns allow the remainder type of 'match'-expressions to be progressively narrowed. This enables more precise typing and programming patterns.

3.6 Disjoint Intersections

The features presented in the previous sections work for simple cases of match narrowing, but would fail to type check more complex examples with *GADTs* presented in Section 2.5. The final piece of the puzzle required for match narrowing is to introduce a new subtyping rule for disjoint type intersections. Consider the following example:

```

1 sealed abstract class Shape [+P]
2
3 sealed abstract class PPoint
4 sealed abstract class PCircle
5 sealed abstract class PRectangle
6
7 case object Point extends Shape [PPoint]
8 case class Circle(r: Int) extends Shape [PCircle]
```

```

9 case class Rectangle(h: Int, q: Int) extends Shape[PRectangle]
10
11 def pointToCircle[P](shape: Shape[P | PPoint]): Shape[P | PCircle] =
12   shape match {
13     case Point => Circle(0)
14     case other => other
15   }

```

Shape (line 1) is a *GADT* [6] indexed by P, the kind of shape. Point, Circle and Rectangle (lines 7-9) are the different possible shapes. Each extends Shape with its corresponding index types PPoint, PCircle and PRectangle (defined in lines 3-5). The function pointToCircle (line 11) converts a Shape which may be a Point to a Shape which may be a Circle. It makes use of the match narrowing type other ‘case’-branch (line 14). The resulting other expression (line 14) needs to be a subtype of the return type Shape[P | PCircle]. However, match narrowing as presented so far fails to type this correctly.

To understand why, it is necessary to step through the deduction rules [30]. The type of the scrutinee shape (line 12) is decomposed into the type:

```
Shape[P | PPoint] & (Point | Circle | Rectangle)
```

The pattern Point (line 13) is complete, so it is possible to take a difference from the scrutinee type, resulting in the type for other:

```
Shape[P | PPoint] & (Circle | Rectangle)
```

Now the type checker must determine if the type of other is a valid subtype of function’s return type Shape[P | PCircle]; it needs to show that:

```
Shape[P | PPoint] & (Circle | Rectangle) <: Shape[P | PCircle]
```

It distributes the type intersection ‘&’ in the type union ‘|’:

```
(Shape[P | PPoint] & Circle) | (Shape[P | PPoint] & Rectangle)
  <: Shape[P | PCircle]
```

The subtyping rules state that $A \mid B <: C$ if $A <: C$ and $B <: C$, so the type checker must determine:

- (1) `Shape[P | PPoint] & Circle <: Shape[P | PCircle]`
- (2) `Shape[P | PPoint] & Rectangle <: Shape[P | PCircle]`

The subtype relation (1) can be shown by the rule $A \& B <: C$ if $A <: C$ or $B <: C$. In this scenario it is only necessary to show that:

```
Circle <: Shape[P | PCircle]
```

Since `Circle` is defined to be a subtype of `Shape[PCircle]`, it is sufficient to show that:

```
Shape[PCircle] <: Shape[P | PCircle]
```

`Shape` is covariant in its type argument, so the type checker must show that:

```
PCircle <: P | PCircle
```

It does this by using subtype rules for unions ($A <: B \mid C$ if $A <: B$ or $A <: C$) and reflexivity ($A <: A$).

The subtype relation (2) can be shown by approximating `Rectangle` to its supertype `Shape[PRectangle]`:

```
Shape[P | PPoint] & Shape[PRectangle] <: Shape[P | PCircle]
```

The intersection can be distributed into the constructor `Shape`:

```
Shape[(P | PPoint) & PRectangle] <: Shape[P | PCircle]
```

Since `Shape` is covariant, the compiler only needs to show that:

```
(P | PPoint) & PRectangle <: P | PCircle
```

It does this by distributing the intersection to the type union:

```
(P & PRectangle) | (PPoint & PRectangle) <: P | PCircle
```

Now it needs to show that both sides of the union are subtypes of `P | PCircle`:

```
(3) P & PRectangle <: P | PCircle
```

```
(4) PPoint & PRectangle <: P | PCircle
```

(3) is true because $P \& PRectangle <: P \mid PCircle$ if $P <: P \mid PCircle$, and that is true because $P <: P$.

Finally, in order to prove the larger subtype relation, the type checker needs to prove (4), $PPoint \& PRectangle <: P \mid PCircle$. Up until this point, all the subtype rules are implemented in the original Scala compiler. However to determine $PPoint \& PRectangle <: P \mid PCircle$, the compiler needs to determine either that $PPoint <: P \mid PCircle$ or that $PRectangle <: P \mid PCircle$, but neither is possible as the types are unrelated. Failing this subtype check causes the entire function to fail to type check.

Fortunately, there is a way to solve this constraint. The Scala compiler contains a function that determines if two types are known to be disjoint from each other [7]. If types

A and B are proven to be disjoint, there exists no value x such that $x <: A$ and $x <: B$. This intuition can be leveraged to complete the subtype check presented above. The type $A \& B$ is a subtype of C if A and B are provably disjoint. This is valid because $A \& B$ is uninhabited, making it equivalent to the bottom type `Nothing`.

Returning to the subtype check (4), `PPoint & PRectangle <: P | PCircle` is true if `PPoint` and `PRectangle` are disjoint. Since `PPoint` and `PRectangle` are not related, and both are sealed classes with all subclasses known at compile time, the type checker can prove that they are indeed disjoint. This enables the compiler to prove the overall relation `Shape[P | PPoint] & (Circle | Rectangle) <: Shape[P | PCircle]` to be true, and for the function `pointToCircle` to type check.

The changes to the Scala compiler are available in full at: <https://github.com/pjmc-oliveira/scala-match-narrowing>.

Chapter 4

Extension: Folds and Unfolds

The precise-typed `transform` function introduced in Section 2.5 borrows behaviour from traditional `map` [12] and `fold` [11] operations. Like `map`, `transform` applies a function to each element of the tree, and like `fold` it is capable of reducing the structure of the tree at each step. However, unlike `fold`, `transform` is not capable of expressing transformations from a tree to another type, nor from other types to trees.

The traditional `fold` function collapses a tree-like structure to a single value. For a type like list of integers, `fold` would have the type signature:

```
(list: List[Int], fn: (Int, A) => A, base: A) => A
```

The function `fn` combines an `Int` with the result of the fold of the remainder of the list; `base` is the base case for an empty list.

The inverse operation is `unfold`, which expands a seed value to a tree-like structure. For a list of integers, it would have the type signature:

```
(seed: A, fn: A => Option[(Int, A)]) => List[Int]
```

The `seed` serves as the initial value; `unfold` applies `fn` to `seed` which either yields an `Int` and a new seed to continue the recursion, or `None` to stop the unfold.

Together, `fold` and `unfold` allow developers to express a greater variety of tree transformations without resorting to writing recursive functions by hand. This furthers the goal of reducing boilerplate while maintaining type-safety.

This chapter introduces an extension to precise-typed transformations to allow for both folding and unfolding of trees. Section 4.1 shows an encoding of folds and unfolds using type unions (1). Section 4.2 shows an alternative encoding of folds and unfolds using ‘match’-types. Section 4.3 evaluates the tradeoffs of each encoding.

4.1 Encoding with Union Types

In order to represent a fold over a precisely-typed tree, it is necessary to represent individual layers of the tree as they are being folded over. The ability to represent an individual layer allows a `fold` function to have the type: $(\text{Lang}[P], \text{Layer}[P, A] \Rightarrow A) \Rightarrow A$, where `Layer[P, A]` is a node of phase `P` with children of type `A` (where `A` might not be related to `LangK`). It is also desirable to retain the ability to express a `transform` function of type: $(\text{Lang}[P], \text{Partial}[P, P2] \Rightarrow \text{Partial}[P2, P2]) \Rightarrow \text{Lang}[P2]$, where `Partial[P, P2]` is a partially transformed tree with root of phase `P` and children of phase `P2`. Additionally, `transform` can be implemented in terms of `fold`. These requirements can be satisfied with a simple modification of `LangK` presented in Listing 4.1. `LangK[PSelf, PChild, A]` represents a node whose children are either of type `Lang[PChild]` or type `A`. An extra type parameter `A` is added to the definition of `LangK` to represent the type of the result of the fold (line 1). The type alias `Lang` is updated for the common case where the tree only contains other tree nodes by setting `A` to `Nothing` (line 6). A new type alias `Child[+P, +A]` is created to represent the children of each node (line 7). The type variable `P` represents the phase of the node as before, while the type variable `A` represents the result of the fold. The definition of `Child` expresses that it may be an instance of `Lang[P]` or an instance of `A`. In other words, a value of type `LangK[P, PChild, A]` is a node whose children are either of type `Lang[PChild]` or of type `A`. The signature of `mapChildren` is changed to reflect that the given function `fn` must modify each child of type `Child[PChild, A]` to `Child[PChild2, B]` (line 2). Each subclass of `LangK` must now define all its child nodes to be of type `Child` (instead of `Lang` as shown in Section 2.5).

```
1 sealed abstract class LangK[+PSelf, +PChild, +A] {
2   def mapChildren[PChild2, B](
3     fn: Child[PChild, A] => Child[PChild2, B]
4   ): LangK[PSelf, PChild2, B]
5 }
6 type Lang[+P] = LangK[P, P, Nothing]
7 type Child[+P, +A] = Lang[P] | A
```

Listing 4.1: Representation of `LangK` with type unions

An example node `If` is given in Listing 4.2. The invariants regarding the phase of the node are the same as in Section 2.5: every node must extend `LangK` with a phase type for `PSelf`, and a phase type for `PChild` only if it contains child nodes; additionally, an extra type variable `A` is added in line 2. The child nodes: `cond`, `con`, and `alt`, must now be of type `Child[P, A]` (lines 3-5). The only difference to the implementation of `mapChildren` (line

7) is its type signature; the implementation remains the same in that it simply applies `fn` to each child without recursing. The other nodes presented in Section 2.5 can be similarly represented.

```

1 sealed abstract class PBool
2 case class If[+P, +A](
3     cond: Child[P, A],
4     con: Child[P, A],
5     alt: Child[P, A]) extends LangK[PBool, P, A]
6 {
7     def mapChildren[P2, B](
8         fn: Child[P, A] => Child[P2, B]
9     ): LangK[PBool, P2, B] = If(fn(cond), fn(con), fn(alt))
10 }

```

Listing 4.2: Implementation of If

A new function `fold` is introduced (Listing 4.3, line 3) to perform folds over the node tree. It makes use of a new type alias `Layer[+P, +A]` (line 1) to represent a layer of the fold. A `Layer[P, A]` is a node of phase `P`, but its children have been converted to `A`. The `fold` function accepts a node of type `Lang[P1]` and a function `fn` of type `Layer[P1, A] => A`, and returns a value of type `A`. The implementation (line 4) is identical to the implementation of `transform`. `fold` recurses on the child of each node applying it to `fold` with `fn`; once the leaf nodes are reached, `fn` returns a value of type `A`. As each child is replaced with an `A` value, each parent is passed to `fn`, until all nodes have been folded into a single value of type `A`.

```

1 type Layer[+P, +A] = LangK[P, Nothing, A]
2 extension [P1](node: Lang[P1]) {
3     def fold[A](fn: Layer[P1, A] => A): A =
4         fn(node.mapChildren(child => child.fold(fn)))
5 }

```

Listing 4.3: Implementation of function fold

With the use of `fold`, functions that previously required explicit recursion can factor out the recursive step. An example of this is the re-implementation of the `pretty` function (Listing 4.4) previously presented in Section 2.5 (Listing 2.18). `pretty` now calls the `fold` method directly (line 1) and is not recursive. Each case (lines 2-11) simply transforms each node layer to a `String`, using the already folded children. All `Child` nodes must be explicitly annotated with a `String` type; this is to work around a current limitation of the Scala type inference. The `fold` function accepts a function `fn` that in turn accepts a node of type `Layer[P, A]`, meaning its children are of type `Child[Nothing, A]`. The type

alias `Child[Nothing, A]` expands to `LangK[Nothing, Nothing, Nothing] | A`; by the invariants presented there are no nodes of phase `Nothing`. Indeed, because `LangK` is sealed, this can be asserted statically as the compiler can verify all subclasses of `LangK`. However, since the type checker does not take into consideration whether a type is uninhabited, the type `LangK[Nothing, Nothing, Nothing] | A` cannot be simplified to `A`. The explicit type ascriptions on patterns serve as a workaround because the type checker is decoupled from the pattern exhaustiveness checker. The type checker elaborates the rest of the branch assuming each child is of the type `A` (in this case `String`); the exhaustiveness checker later verifies that indeed `LangK[Nothing, Nothing, Nothing]` is uninhabited and need not be matched against, making the entire pattern match exhaustive. Future work on this topic could explore the possibility of integrating the knowledge of uninhabited types into the type checker.

```

1 def pretty(tree: Lang[Any]): String = tree.fold {
2   case True  => "True"
3   case False => "False"
4   case If(cond: String, con: String, alt: String) =>
5     s"(if $cond $con $alt)"
6   case Let(name, value: String, body: String) =>
7     s"(let ($name $value) $body)"
8   case Var(name) => s"$name"
9   case Lambda(name: String, body: String) =>
10    s"(lambda ($name) $body)"
11  case Apply(fun: String, arg: String) => s"($fun $arg)"
12 }

```

Listing 4.4: Implementation of `pretty` using `fold`

The complement to the `fold` function is `unfold` (Listing 4.5). This function accepts a `seed` value of type `A` (line 2) and a function `fn` which unfolds one layer of the tree (line 3). `unfold` applies `fn` to the `seed` which returns a new `Layer`; the children of the resulting `Layer` serve as new seeds to `unfold`, recursing until no new seeds are created (line 5). The type of `fn` is `(Lang[Nothing] | A) => Layer[P, A]` for the same reason type ascriptions are needed in the `pretty` function above. Ideally, the type of `fn` should be `A => Layer[P, A]` because `Lang[Nothing]` is uninhabited, but the compiler does not account for uninhabited types during the type checking phase. At use site, this requires the function `fn` to be η -expanded with a pattern match of type `A` to determine exhaustiveness.

```

1 def unfold[P, A](
2   seed: A,
3   fn: (Lang[Nothing] | A) => Layer[P, A]
4 ): Lang[P] = {
5   fn(seed).mapChildren { newSeed => unfold(newSeed, fn) }

```

6 }

Listing 4.5: Implementation of `unfold`

A use case for the `unfold` function is parsing S-Expressions [22] (Listing 4.6). The function `alg` unfolds one layer of S-Expressions into a `Layer[PBool | PLet, SExpr]` (line 3). A non-terminal case, such as parsing `If` (line 11), simply returns a node with its children still unparsed. Each child will be later parsed as a new seed for `unfold`. In terminal cases, such as parsing `True` (line 5), a node with no children is returned. `alg` is η -expanded when called by `unfold` in line 18; this is because it allows for the input `s` to be pattern-matched as an `SExpr` and deemed exhaustive. It is for this reason that `alg` can be defined to accept a value of `SExpr` even though `unfold` is instantiated to accept a function that accepts `Lang[Nothing] | SExpr`. `parse` also makes use of exceptions to return a `None` value (line 21) if the parse fails (line 14).

```
1 import SExpr.{SAtom as A, SList as L}
2 def parse(source: SExpr): Option[Lang[PBool | PLet]] = {
3   def alg(s: SExpr): Layer[PBool | PLet, SExpr] = s match {
4     // True
5     case A("True") => True
6     // False
7     case A("False") => False
8     // <name>
9     case A(name) => Var(name)
10    // (if <cond> <con> <alt>)
11    case L(A("if"), cond, con, alt) => If(cond, con, alt)
12    // (let (<name> <value>) <body>)
13    case L(A("let"), L(A(name), value), body) => Let(name, value, body)
14    case _ => throw Exception()
15  }
16
17  try {
18    val parsed = unfold(source, { case s: SExpr => alg(s) })
19    Some(parsed)
20  } catch {
21    case _ => None
22  }
23 }
```

Listing 4.6: Implementation of `parse` using `unfold`

The implementation of `transform` (Listing 4.7, line 3) remains the same as in Section 2.5, but its type signature changes. It makes use of a new type alias `Partial[+P1, +P2]` (line 1) to represent partially transformed trees. A `Partial[P1, P2]` is a tree whose root

node is of phase P1, but all its children nodes are in phase P2. Additionally, because `Partial` specifies that `LangK`'s fold variable is `Nothing`, it requires all children to be other nodes. The `transform` function then accepts a function `fn` of type `Partial[P1, P2] => Partial[P2, P2]`, meaning it only handles cases in which the children are other `Lang` nodes. `transform` can be used in the same way as shown in Section 2.5. This highlights how the implementation of `transform` is just a special case of `fold`, where the output is a different `Lang` node tree. It also highlights how `transform` is different from traditional `map` functions in that it can modify the structure of the tree.

```

1 type Partial[+P1, +P2] = LangK[P1, P2, Nothing]
2 extension [P1](node: Lang[P1]) {
3   def transform[P2](fn: Partial[P1, P2] => Partial[P2, P2]): Lang[P2] =
4     fn(node.mapChildren(child => child.transform(fn)))
5 }

```

Listing 4.7: Implementation of `transform`

4.2 Encoding with Match Types

An alternative encoding which allows for folding and unfolding a tree is to use ‘match’-types instead of type unions. ‘match’-types in Scala offer a lightweight form of type dependent functions [3]. Listing 4.8 presents the definition of the `XOr[I, +L, +R]` type. If the type variable `I` is `Rec` (line 4), then `XOr[I, L, R]` reduces to type `L`. If `I` is `Lay` (line 5), then `XOr[I, L, R]` reduces to type `R`. Otherwise, `XOr` does not reduce and type checking fails. `Rec` and `Lay` are used to index recursive trees and tree layers respectively.

```

1 sealed abstract class Rec
2 sealed abstract class Lay
3 type XOr[I, +L, +R] = I match {
4   case Rec => L
5   case Lay => R
6 }

```

Listing 4.8: Implementation of `XOr` type

The definition of `LangK[I, +PSelf, +PChild, +A]` makes use of the `XOr` type (Listing 4.9). The new type variable `I` serves to determine whether the given node is fully recursive (if `I` is `Rec`), or if it is a single layer (if `I` is `Lay`). The type alias for `Lang` is updated to reflect that it is a fully recursive tree (line 6), and the type alias for `Child` is updated so that the given index determines whether it is a fully recursive node of `Lang[P]` or a value of type `A` (line 7). The definition of `Child[I, +P, +A]` uses the type index to specify

whether it is recursive or not, without any overlap, meaning the type checker never needs to check if a type is uninhabited. The type signature of `mapChildren` is modified so that the given function `fn` has type `Child[I, PChild, A] => Child[I2, PChild2, B]` (line 2). The `Partial` type alias now specifies that it represents fully recursive trees (line 8). The `transform` function makes use of the modified `Partial` type in its signature, but is otherwise unchanged from the type union encoding (line 13).

The implementation of `fold` in the ‘match’-types encoding is similar to the one using type unions (line 18). It performs the same steps to recursively traverse the tree until the leaf nodes are reached, uses the given `fn` function to convert the leaves to type `A`, and folds each subsequent `Layer` with `fn`. The `Layer[+P, +A]` type alias (line 9) specifies a single layer of the fold, and defines the index `I` to be `Lay`.

```

1 sealed abstract class LangK[I, +PSelf, +PChild, +A] {
2   def mapChildren[I2, PChild2, B](
3     fn: Child[I, PChild, A] => Child[I2, PChild2, B]
4   ): LangK[I2, PSelf, PChild2, B]
5 }
6 type Lang[+P]          = LangK[Rec, P, P, Nothing]
7 type Child[I, +P, +A] = XOr[I, Lang[P], A]
8 type Partial[+P1, +P2] = LangK[Rec, P1, P2, Nothing]
9 type Layer[+P, +A]     = LangK[Lay, P, Nothing, A]
10
11 extension [P1](node: Lang[P1]) {
12
13   def transform[P2](
14     fn: Partial[P1, P2] => Partial[P2, P2]
15   ): Lang[P2] =
16     fn(node.map(child => child.transform(fn)))
17
18   def fold[A](fn: Layer[P1, A] => A): A =
19     fn(node.mapChildren(child => child.fold(fn)))
20 }

```

Listing 4.9: Representation of `LangK` with ‘match’-types

Implementing the tree nodes now requires each node’s children to be of type `Child[I, P, A]` (Listing 4.10, lines 9-11). The implementations of `mapChildren` have their type signatures updated (lines 3 and 13), but their bodies remain unchanged. There is a difference in the representation of leaf nodes, such as `True` (line 2), compared to the union type encoding. The type variable `I` must be invariant for ‘match’-types, such as `XOr`, to reduce, so `True` must be represented as class polymorphic over the phantom variable `I`. The types `True[Rec]` and `True[Lay]` are incompatible with each other, even though they have the same runtime representation.

```

1 sealed abstract class PBool
2 case class True[I]() extends LangK[I, PBool, Nothing, Nothing] {
3   def mapChildren[I2, PChild2, B](
4     fn: Child[I, PChild, A] => Child[I2, PChild2, B]
5   ): LangK[I2, PBool, PChild2, B] = True()
6 }
7
8 case class If[I, +P, +A](
9   cond: Child[I, P, A],
10  con: Child[I, P, A],
11  alt: Child[I, P, A]) extends LangK[I, PBool, P, A]
12 {
13   def mapChildren[I2, PChild2, B](
14     fn: Child[I, PChild, A] => Child[I2, PChild2, B]
15   ): LangK[I2, PBool, PChild2, B] = If(fn(cond), fn(con), fn(alt))
16 }

```

Listing 4.10: Implementation of True and If

The ‘match’-types encoding of the fold function can also be used to create a pretty printing function (Listing 4.11). Compared to the pretty function using type unions, this implementation does not need type ascriptions on the child nodes (lines 2-8). The definition of Child does not rely on Lang[Nothing] being uninhabited, and explicitly specifies whether or not it is recursive with the type index I. In the case of the function passed to fold (line 1), it is specified that the child is not recursive and indeed can only be a value of type String.

```

1 def pretty(tree: Lang[Any]): String = tree.fold {
2   case True()           => "True"
3   case False()          => "False"
4   case If(cond, con, alt) => s"(if $cond $con $alt)"
5   case Let(name, value, body) => s"(let ($name $value) $body)"
6   case Var(name)         => s"$name"
7   case Lambda(name, body) => s"(lambda ($name) $body)"
8   case Apply(fun, arg)   => s"($fun $arg)"
9 }

```

Listing 4.11: Implementation of pretty using fold

The ‘match’-types encoding allows the type to explicitly specify whether a node is fully recursive or not. This enables the implementation of unfold (Listing 4.12) to accept a function fn of type A => Layer[P, A] (line 3). Just as above, this does not rely on Lang[Nothing] being uninhabited and fn accepts a value of type A directly. The implementation of unfold is otherwise similar to the implementation in the type union encoding.

First `fn` is applied to the `seed` value; then each child of the resulting `Layer` is recursively mapped with `unfold` again (line 5).

```

1 def unfold[P, A](
2     seed: A,
3     fn: A => Layer[P, A]
4 ): Lang[P] = {
5     fn(seed).mapChildren { newSeed => unfold(newSeed, fn) }
6 }

```

Listing 4.12: Implementation of `unfold` with ‘match’-types

The `parse` function can be implemented using the ‘match’-types encoding of `unfold` (Listing 4.13). The implementation of `alg` (line 3) is unchanged from Listing 4.6. The difference compared to the type union encoding is that the function `alg` does not need to be η -expanded (line 7). This is because the `unfold` function accepts a function `fn` which accepts a value of type `A` (instead of a type union with the uninhabited `Lang[Nothing]` type). `fn` only needs to accept `A` because it is fed its own output, which is a `Layer` value, and `Layer` can only be a `LangK` node with children of type `A`. The ability to correctly specify the input to `fn` means that the user does not need to η -expand the function `alg` with a pattern match that is only later discovered to be exhaustive.

```

1 import SExpr.{SAtom as A, SList as L}
2 def parse(source: SExpr): Option[Lang[PBool | PLet]] = {
3     def alg(s: SExpr): Layer[PBool | PLet, SExpr] =
4         /* same as Listing 4.6 */
5
6     try {
7         val parsed = unfold(source, alg)
8         Some(parsed)
9     } catch {
10        case _ => None
11    }
12 }

```

Listing 4.13: Implementation of `parse` using `unfold`

4.3 Discussion

Both the encoding using type unions and the one using ‘match’-types come with tradeoffs. The type union encoding makes use of uninhabited types and relies on the exhaustiveness checker proving that these values are not possible. Even then, the type union encoding

requires extra type ascriptions when pattern matching, and superfluous η -expansions. A modification to the type checker to account for uninhabited types could make this encoding more ergonomic, but it would come at additional cost in complexity to the compiler implementation.

The encoding with ‘match’-types allows for more precise encoding of recursion, but suffers from different drawbacks. Each node must have an additional type parameter so that it may be used in a recursive or non-recursive context. Leaf nodes can no longer simply extend `LangK` with `Nothing` as the phase for their children and can no longer be reused across contexts. Because every node must be indexed by an invariant type variable `I`, leaf nodes must be reinstantiated every time they are used.

Chapter 5

Case Study: Lacs Compiler

This chapter discusses the compiler for the Lacs programming language, and how to modify it to make it more type-safe, without compromising on boilerplate and flexibility. Lacs is an instructional language and compiler for CS 241E, the second year advanced compilers course at the University of Waterloo. The language is mostly a subset of the Scala programming language, containing: arithmetic expressions, variables, ‘if’-expressions, procedures, and closures.

The course is structured for students to build a compiler from the backend to the frontend, starting by encoding MIPS assembly language instructions to binary machine code, and progressively adding language constructs to the intermediate representation, parsing, and type-checking. The changes primarily focus on the middle-end of the compiler, the transformations to the intermediate representation and code generation.

Section 5.1 discusses the existing implementation of the Lacs compiler. Section 5.2 discusses the changes implemented in the Lacs compiler and the benefits they afford. Section 5.3 discusses challenges and additional costs in adding precise-typing to the Lacs compiler.

5.1 Existing Implementation

5.1.1 Program Representation

The reference implementation uses `Code` (Listing 5.1) as the all encompassing intermediate representation (IR) for the compiler, similar to the “uni-typed compiler” discussed in Section 2.3. It contains high-level language constructs like `Closure` and `CallClosure`, all

the way down to low-level machine `CodeWord`. Working back to front, the compiler builds support for higher-level nodes by lowering them to lower-level nodes.

The compiler targets a subset of MIPS assembly¹ represented as 32-bit `Words`, which are wrapped as `CodeWord` nodes (line 7). `CodeWord` is the lowest node in the compiler pipeline; all other nodes are eventually lowered to a sequence of `CodeWords`.

The next group of nodes are primarily related to translating labels to machine instructions. Labels are represented by `Label` (line 4), and are attached to the tree with `Define` and `Use` nodes (lines 8 and 9). The `Define` node marks its address in the program as the value pointed to by the `Label`, whereas `Use` inserts the address of the given label in the program. A branching instruction to the address of a `Label` is represented by the `BeqBne` node (line 10). Also in this group is the translation of `Comment` (line 11) and `Block` nodes (line 12) to `Words`. `Comment` nodes are used for debugging the intermediate code generation, and are stripped from the final output. `Block` nodes are a sequence of other `Code` nodes, and are lowered to a sequence of machine instructions.

The third group of nodes deals with `Variables` (line 15) and how to allocate them on a stack frame. The `VarAccess` node (line 17) represents both reading from a `Variable` to a register and writing from a register to a `Variable`, distinguished by the `read` boolean flag. Whenever `Code` is emitted for an expression, the compiler may use two registers directly, `$3` and `$4`. By convention, `$3` is used to store the final result of an expression, and `$4` is used as a scratch register. Other temporary values created by an expression are stored in compiler synthesized `Variables` which are accessed from a `VarAccess` node. At a later transformation of the `Code` tree, both compiler-synthesized and user-created `Variables` are allocated in each `Procedure`'s stack frame. Conceptually, `Variables` should be thought of as “virtual registers”, and in a different implementation of the compiler they could be allocated as machine registers.

Lexical scopes and ‘if’-expressions are introduced next, represented by `Scope` (line 21) and `IfStmt` (line 22) nodes respectively. `IfStmt` is compiled to evaluate `e1` and `e2` and use `comp` to either fallthrough to the `thens` branch, or jump to the `elseLabel` at the start of the `elses` branch. `Scope` serves to declare `Variables` in a given `Code` tree so that they may be allocated together.

The `Procedure` class (line 26) and the `Call` node (line 32) introduce procedures and procedure calls in the fifth group of nodes. The `Procedure` class does not extend `Code` and is not included directly in the tree. It contains a mutable instance of `Code` in line 28, which is written to after the `Procedure` instance is created to allow for cyclic references

¹<https://student.cs.uwaterloo.ca/~cs241e/current/mipsref.pdf>

and recursion. From this point onwards, the compiler represents a program as a collection of Procedures.

The last two nodes to be introduced are `Closure` and `CallClosure` (lines 37 and 38). `Closure` represents the creation of a procedure closure which captures the current runtime environment, and `CallClosure` represents the calling of the procedure closure.

The `ProgramRepresentation` object is structured so that low-level nodes are defined first, and high-level nodes are defined after. As each group of nodes builds on top of the previous, once support for a node is added to the compiler, it will be supported for all future assignments. Since all nodes extend a single `Code` type, the type checker cannot distinguish between them, and the programmer relies on comments (lines: 3, 14, 20, 25, and 36) to enforce this invariant. This makes it difficult in other parts of the code base to keep track of which `Code` nodes are allowed in which phase.

```
1 object ProgramRepresentation {
2
3   /* The following are used starting in Assignment 2. */
4   class Label(val name: String)
5
6   sealed abstract class Code
7   case class CodeWord(word: Word) extends Code
8   case class Define(label: Label) extends Code
9   case class Use(label: Label) extends Code
10  case class BeqBne(bits: Seq[Boolean], label: Label) extends Code
11  case class Comment(message: String) extends Code
12  case class Block(stmts: Seq[Code]) extends Code
13
14  /* The following are used starting in Assignment 3. */
15  class Variable(val name: String, val isPointer: Boolean = false)
16
17  case class VarAccess(register: Reg, variable: Variable, read: Boolean)
18    extends Code
19
20  /* The following are used starting in Assignment 4. */
21  case class Scope(variables: Seq[Variable], code: Code) extends Code
22  case class IfStmt(elseLabel: Label, e1: Code, comp: Code, e2: Code,
23                  thens: Code, elses: Code) extends Code
24
25  /* The following are used starting in Assignment 5. */
26  class Procedure(val name: String, val parameters: Seq[Variable],
27                val outer: Option[Procedure] = None) {
28    var code: Code = null
29    // ...
30  }
```

```

31
32
33 case class Call(procedure: Procedure, arguments: Seq[Code],
34                 isTail: Boolean = false) extends Code
35
36 /* The following are used starting in Assignment 6. */
37 case class Closure(procedure: Procedure) extends Code
38 case class CallClosure(closure: Code, arguments: Seq[Code],
39                       parameters: Seq[Variable],
40                       isTail: Boolean = false) extends Code
41 }

```

Listing 5.1: Existing minimal definition of Code

5.1.2 Shallow Embedding

Throughout the compiler, shallow embeddings are used to provide support for arithmetic expressions and ‘while’-loops, by representing them in terms of other nodes. Shallow embedding is a technique to represent constructs from a source language as a direct translation to a target language. In the case of Lacs, this is done by using functions that build a subtree of `Code` nodes without creating a new subclass of `Code`. This serves both to reinforce the idea that complex language constructs can be represented from simpler ones, as well as to simplify the compiler’s intermediate representation.

An excerpt from the `CodeBuilders` object (Listing 5.2) shows an example of both shallow embedding and the compiler’s use of synthesized `Variables` mentioned in Section 5.1.1. `binOp` creates a binary expression given `Code` for two subexpressions and `Code` for an operator to combine them. Once the first expression is evaluated (line 5), the compiler synthesizes a temporary `Variable` to store its result (line 6). The second expression is then evaluated (line 7), and the previous intermediate result is loaded from the temporary `Variable` (line 8). Finally, `Code` for the operation evaluates with the results of the two input expressions (line 9).

```

1 object CodeBuilders {
2   def binOp(e1: Code, op: Code, e2: Code): Code = {
3     val temp = new Variable("temp")
4     Scope(Seq(temp), block(
5       e1, // - Evaluate 'e1' and store result in $3
6       write(temp, Reg.result), // - Write $3 into 'temp'
7       e2, // - Evaluate 'e2' and store result in $3
8       read(Reg.scratch, temp), // - Read 'temp' into $4
9       op)) // - Evaluate 'op' with results of 'e1'

```

```

10                                     //      and 'e2' in $3 and $4
11     }
12     // ...
13 }

```

Listing 5.2: An example of shallow embedding of binary operations in `CodeBuilders`

5.1.3 Tree Transformations

The reference implementation defines a helper function `transformCodeTotal` (line 3 in Listing 5.3) and uses it extensively to facilitate tree transformations in the compiler. `transformCodeTotal` is mutually recursive with its inner function `processChildren` (line 5). `processChildren` calls `transformCodeTotal` on the direct children of each node (lines 5-24), while `transformCodeTotal` calls `processChildren` on the root of the given node, and applies the given function `fun` to its result (line 27). Combined, the functions traverse the `Code` tree from the leaves to the root, applying the transformation function `fun` to all nodes, one layer at a time. `transformCodeTotal` is usually used from `transformCode` (line 30), which makes use of Scala’s `PartialFunction` as a convenience to keep nodes unchanged if the given partial function is not defined for them. Because Scala is an impure language, `transformCode` can also be used for side-effecting tree traversals, without using the resulting transformed tree.

While implementing `transformCodeTotal`, it is important for the student to apply the given function `fun` to every `Code` node exactly once. Missing a node would mean the transformation is incomplete, while applying it to a node twice could potentially result in non-termination. The existing type of `transformCodeTotal` does not help the compiler check this; as long as the function returns a `Code` node, the type checker will gladly accept it.

```

1 object Transformations {
2
3     def transformCodeTotal(code: Code, fun: Code=>Code): Code = {
4
5         def processChildren(code: Code): Code = code match {
6             case Block(children) =>
7                 Block(children.map(transformCodeTotal(_, fun)))
8             case Scope(variables, body) =>
9                 Scope(variables, transformCodeTotal(body, fun))
10            case IfStmt(elseLabel, e1, comp, e2, thens, elses) =>
11                IfStmt(
12                    elseLabel,

```

```

13         transformCodeTotal(e1, fun),
14         transformCodeTotal(comp, fun),
15         transformCodeTotal(e2, fun),
16         transformCodeTotal(thens, fun),
17         transformCodeTotal(elses, fun))
18     case Call(procedure, args, isTail) =>
19         Call(procedure, args.map(transformCodeTotal(_, fun)), isTail)
20     case CallClosure(closure, args, params, isTail) =>
21         CallClosure(
22             transformCodeTotal(closure, fun),
23             args.map(transformCodeTotal(_, fun)), params, isTail)
24     case _ => code
25 }
26
27 fun(processChildren(code))
28 }
29
30 def transformCode(
31     code: Code,
32     fun: PartialFunction[Code, Code]
33 ): Code = {
34     transformCodeTotal(
35         code,
36         code => if(fun.isDefinedAt(code)) fun(code) else code
37     )
38 }
39
40 // ...
41 }

```

Listing 5.3: Existing transformCode and transformCodeTotal

eliminateScopes in Listing 5.4 provides an example of a transformation that selectively modifies the tree. It makes use of transformCode (line 3), which accepts a PartialFunction and only applies the transformation if the function is defined at that node. Additionally, the function is an effectful operation to collect the scoped variables from the tree (line 5). The developer must write documentation to convey that eliminateScopes removes all instances of Scope. Because fun is not defined for any other nodes, all other instances of Code are unchanged.

```

1 def eliminateScopes(code: Code): (Code, Seq[Variable]) = {
2     var variables = Seq[Variable]()
3     val newCode = transformCode(code, {
4         case Scope(vars, code) =>
5             variables += vars

```

```

6     code
7   })
8   (newCode, variables)
9 }

```

Listing 5.4: Existing implementation of `eliminateScopes`

5.1.4 Garbage Collection

The reference implementation of Lacs includes a garbage collector for automatic memory management. The compiler allocates simple and non-escaping values like `Ints` on the stack, and complex or escaping values like `Closures` on the heap. Additionally, stack frames of `Procedures` captured by a closure are also allocated on the heap. The `GarbageCollector` is implemented directly as `Code` in the compiler source code. This is done because the compiler generates an in-memory Scala representation of MIPS instructions and executes it using a MIPS CPU emulator implemented in Scala.

It is imperative that the implementation of the `GarbageCollector` itself does not allocate any objects on the heap during a collection cycle. Doing so could trigger a new collection cycle before the existing collection cycle is complete. This possibility would be both difficult to trigger during testing and hard to debug. Since one of the ways Lacs can allocate objects on the heap is to create `Closures`, it is necessary to forbid this within the `GarbageCollector` procedures. This is currently done by convention, and is not enforced by the type system.

```

1 object GarbageCollector extends MemoryAllocator with HeapSettings {
2
3   def procedures: Seq[Procedure] =
4     Seq(allocateProc, collectGarbage, traverse, forwardPtrs)
5
6   val allocateProc = new Procedure("allocateProc", /* parameters */)
7
8   allocateProc.code = /* Code (must not contain closures) */
9
10  // ...
11 }

```

Listing 5.5: Existing `GarbageCollector`

5.2 Modified Implementation

5.2.1 Program Representation

The imprecise nature of the `Code` tree in the existing implementation of Lacs means the developer needs to resort to runtime assertions to ensure program invariants. To introduce more precisely-typed trees (Listing 5.6), the original definition of `Code` is replaced with a new class `CodeK` (line 4). As before, the class needs to be sealed and abstract to allow for exhaustiveness checking. The new class has two type variables: `PSelf` representing phase of this node, and `PChild` representing the phase of its child nodes. Both type variables are covariant to allow the union of nodes of different phases as the tree is built up. For convenience, a `Code` type alias is defined in line 5 for nodes with the same phase as their children.

Each group of nodes should define its own phase tags, such as `PWord` and `PBlock` in lines 10 and 16. A node with no children, such as `CodeWord` in line 11, should extend the base type with its own phase tag for `PSelf`, and `Nothing` for `PChild`. A node with children, such as `IfStmt` in line 17, should extend the base type with its own phase tag, and a single tag `P` of all its children.

Finally, type aliases are defined for groups of phase tags starting with `WithLabels` in line 26, and ending with `WithClosures` in line 34. For convenience, each type alias roughly corresponds to a group of nodes supported in each assignment. With the addition of phase tags, the following subsections show how the type checker can now enforce the invariants of when and where each `Code` phase is valid.

```
1 object ProgramRepresentation {
2
3   // New base type
4   sealed abstract class CodeK[+PSelf, +PChild]
5   type Code[+P] = CodeK[P, P]
6
7   // ...
8
9   // Node with no children
10  sealed abstract class PWord
11  case class CodeWord(word: Word) extends CodeK[PWord, Nothing]
12
13  // ...
14
15  // Node with children
16  sealed abstract class PIf
```

```

17 case class IfStmt[+P](
18     elseLabel: Label,
19     e1: Code[P], comp: Code[P], e2: Code[P],
20     thens: Code[P],
21     elses: Code[P]) extends CodeK[PIf, P]
22
23 // ...
24
25 // Phase groups
26 type WithLabels           = PWord | PLabel
27 type WithComments        = WithLabels | PComment
28 type WithBlocks          = WithComments | PBlock
29 type WithVars            = WithBlocks | PVar
30 type WithScopes          = WithVars | PScope
31 type WithIfs             = WithScopes | PIf
32 type WithCalls           = WithIfs | PCall
33 type WithCallsAndCallClosures = WithCalls | PCallClosure
34 type WithClosures        = WithCalls | PMakeClosure | PCallClosure
35
36 }

```

Listing 5.6: Changes to Code

5.2.2 Tree Transformations

For tree transformations, the changes replace both the `transformCode` and `transformCodeTotal` functions with `transform` (line 4 in Listing 5.7), and add two new type variables `P1` and `P2` to represent the phase of the tree before and after the transformation. `transform` accepts a function `fn` that converts a partially transformed node (whose children have already been transformed) to a fully transformed node (line 5). The type of `fn` is `CodeK[P1, P2] => CodeK[P2, P2]`, specifying that the input `CodeK` is of phase `P1` but its children have already been converted to phase `P2`, while the output is fully converted to phase `P2`. The function then proceeds to recursively transform the tree bottom-up (lines 6-31). Notably, because the function must transform the child nodes, there is no longer a distinction between `transformCode` and `transformCodeTotal`. With the changes implemented to the Scala compiler (See Chapter 3), the caller of `transform` does not lose the ability to partially specify transformations.

```

1 object Transformations {
2
3     extension [P1](code: Code[P1]) {
4         def transform[P2](

```

```

5     fn: CodeK[P1, P2] => CodeK[P2, P2]
6 ): Code[P2] = code match {
7     case CodeWord(word) => fn(CodeWord(word))
8     case Define(label) => fn(Define(label))
9     case Use(label) => fn(Use(label))
10    case BeqBne(bits, label) => fn(BeqBne(bits, label))
11    case Comment(message) => fn(Comment(message))
12    case Block(stmts) => fn(Block(stmts.map(_.transform(fn))))
13    case VarAccess(register, variable, read) =>
14      fn(VarAccess(register, variable, read))
15    case Scope(variables, code) =>
16      fn(Scope(variables, code.transform(fn)))
17    case IfStmt(elseLabel, e1, comp, e2, thens, elses) =>
18      fn(IfStmt(
19        elseLabel,
20        e1.transform(fn),
21        comp.transform(fn),
22        e2.transform(fn),
23        thens.transform(fn),
24        elses.transform(fn)))
25    case Call(procedure, arguments, isTail) =>
26      fn(Call(procedure, arguments.map(_.transform(fn)), isTail))
27    case CallClosure(closure, arguments, parameters, isTail) =>
28      fn(CallClosure(closure.transform(fn),
29                    arguments.map(_.transform(fn)),
30                    parameters, isTail))
31    case Closure(procedure) => fn(Closure(procedure))
32  }
33 }
34
35 // ...
36 }

```

Listing 5.7: New transform

5.2.3 Type Signatures

Precisely-typing the Lacs compiler involved a large number of small changes, but most of these changes were only done at the type level, modifying the type signatures of functions, not their runtime behaviour. An example of this can be seen in the implementation of `binOp` in `CodeBuilders`, previously shown in Listing 5.2. The changes in Listing 5.8 (highlighted in green), only make explicit many of the assumptions already held throughout the compiler pipeline.

```

1 object CodeBuilders {
2   def binOp[P](e1: Code[P], op: Code[P], e2: Code[P]):
3     Code[P | WithScopes] =
4     {
5       val temp = new Variable("temp")
6       Scope(Seq(temp), block(
7         e1, // - Evaluate 'e1' and store result in $3
8         write(temp, Reg.result), // - Write $3 into 'temp'
9         e2, // - Evaluate 'e2' and store result in $3
10        read(Reg.scratch, temp), // - Read 'temp' into $4
11        op)) // - Evaluate 'op' with results of 'e1'
12            // and 'e2' in $3 and $4
13     }
14     // ...
15 }

```

Listing 5.8: Modified CodeBuilders, green highlight represents the changes made to the modified implementation.

5.3 Discussion

5.3.1 Selective Transformations

Transformations that selectively modify the tree are changed to use the new `transform` method instead of the existing `transformCode` function. An example of this is the `eliminateVarAccessesA3` function in Listing 5.9. The original implementation makes use of the existing `transformCode` (line 18) to replace `VarAccess` nodes with other nodes. The developer must rely on documentation to convey that `load` and `store` do not introduce new `VarAccess` nodes.

```

1 class Chunk {
2   def load(Reg, Reg, Variable): Code[PWord] = /* ... */
3   def store(Reg, Variable, Reg): Code[PWord] = /* ... */
4   // ...
5 }
6
7 def eliminateVarAccessesA3(code: Code[WithVars], frame: Chunk)
8   : Code[WithBlocks] = {
9   + code.transform {
10  - def fun: PartialFunction[Code, Code] = {
11    case va: VarAccess =>

```

```

12     if(va.read)
13         frame.load(Reg.framePointer, va.register, va.variable)
14     else frame.store(Reg.framePointer, va.variable, va.register)
15 +     // Passthrough other nodes
16 +     case other => other
17 }
18 - transformCode(code, fun)
19 }

```

Listing 5.9: Changes to `eliminateVarAccessesA3`. Red highlight represents removals, green highlight represents additions.

Changes to the Lacs compiler add new phase tags (`WithVars` and `WithBlocks`) to the input and output of `eliminateVarAccessesA3` (line 7). Combined with the changes to the `Chunk` class (line 2), the type checker can now enforce that only valid `Code` nodes are present before and after the transformation.

The new implementation of `eliminateVarAccessesA3` provides a total function to the `transform` method (line 9). However, it only needs to provide an identity case (line 16) for unchanged nodes. In the existing Scala compiler, this would cause a type error, because `other` would be inferred to have type `Code[WithVars]`, whereas the function needs to return `Code[WithBlocks]`.

To develop an intuition for why this is accepted, some definitions from Listing 5.6 and 5.7 are presented again in Listing 5.10. Recall also that `VarAccess` is the only class that extends `CodeK[PVar, ?]`. In `eliminateVarAccessesA3`, `P1` and `P2` are instantiated to `WithVars` and `WithBlocks`, respectively. The only way the `other` branch is executed is if the node failed to match `VarAccess`, meaning it is not an instance of `CodeK[PVar, WithBlocks]`. The compiler can then safely infer `other` to be of type `CodeK[WithBlocks, WithBlocks]`, allowing the function to compile. More details on the changes to the Scala compiler are presented in Chapter 3.

```

1 type WithVars = WithBlocks | PVar
2 type Code[+P] = CodeK[P, P]
3 extension [P1](code: Code[P1]) {
4     def transform[P2](fn: CodeK[P1, P2] => CodeK[P2, P2]): Code[P2] =
5         /* ... */
6 }

```

Listing 5.10: Relevant definitions

5.3.2 Compile-time Exhaustiveness

The `eliminateLabels` function (Listing 5.11) provides an example of where the compiler changes now allow for an exhaustiveness guarantee. The function iterates over a sequence of `Code` nodes in two passes to collect label offsets and replace them with absolute addresses. It is called at the end of the compilation pipeline, and any higher level `Code` nodes should have been eliminated by this point. Looking at line 24, the existing implementation has an impossible assertion claiming that the only nodes possible at this stage are: `Define`, `CodeWord`, `Use`, and `BeqBne`. Without the help of the Scala type checker, this invariant must be carefully maintained throughout the different phases of the compiler, as well as through any changes to the implementation over time.

Changes to the Lacs implementation allow the developer to statically prove these invariants to be correct. The ability to tag the `Code` tree with the nodes it is allowed to contain allows this to be guaranteed statically. Once the `WithLabels` tag is added in line 1, the type checker can correctly enforce the exhaustiveness of the pattern match in the new implementation (line 23) and the assertion can be removed! By expressing this in the type system, `eliminateLabels` is also guarded against future changes to the implementation, warning the developer if its assumptions are ever violated.

```
1 def eliminateLabels(code: Seq[Code[WithLabels]]): Seq[Word] = {
2   val symbolTable = mutable.Map[Label, Int]()
3
4   def setLabels(): Unit = /* ... */
5
6   def translate(): Seq[Word] = {
7     var location = 0
8     code.flatMap {
9       case Define(label) => Seq()
10      case CodeWord(word) => location += 4; Seq(word)
11      case Use(label) =>
12        if (!symbolTable.isDefinedAt(label))
13          sys.error(s"Undefined label $label")
14        location += 4
15        Seq(Word(Assembler.encodeUnsigned(symbolTable(label))))
16      case BeqBne(bits, label) =>
17        if (!symbolTable.isDefinedAt(label))
18          sys.error(s"Undefined label $label")
19        location += 4
20        Seq(Word(bits
21          ++ Assembler.encodeSigned(
22            (symbolTable(label) - location) / 4, 16)))
23 + // Compiler ensures exhaustiveness
```

```

24 -     case _ => impossible(s"Encountered unsupported code $code.")
25     }
26 }
27
28     setLabels()
29     translate()
30 }

```

Listing 5.11: Modified `eliminateLabels` transformation. Red highlight represents removals, green highlight represents additions.

5.3.3 Preventing Common Mistakes

A common mistake students make when implementing the compiler is to reintroduce high level nodes at a late stage of compilation, when they are assumed to have been eliminated in an early stage of compilation. Recall from Subsection 5.1.1 that the Lacs compiler makes use of synthesized `Variables` to store temporary values. A consequence of this is that the programmer must be careful not to introduce any new `VarAccess` nodes during or after the phase that eliminates them; a challenge is especially notable in the `compileProcedure` function (Listing 5.12). `compileProcedure` first eliminates high-level nodes like `Call` and `IfStmt`, so that the procedure only has nodes like `VarAccess` and lower (line 5). The `Variables` are then collected and stored in a `Chunk` of memory representing the `Procedure`'s stack frame (line 8). The function `eliminateVarAccessesA5` (line 10) replaces `VarAccess` nodes with direct reads and writes to `frame` (lines 15-18). Once there are no more local variables, the function `addEntryExit` (line 25) can add the prologue and epilogue to the procedure. In previous sections of the compiler, a transformation like this would make use of synthesized `Variables` to help in lowering nodes (Listing 5.2), but this is not possible at this stage. This is because the `VarAccess` nodes have already been removed, and also because the Lacs compiler stores `Variables` in a procedure's stack frame, so it cannot access any `Variables` while setting up the stack frame itself.

The type changes prohibit these kinds of mistakes at compile time. The signature of the `addEntryExit` function is changed to `Code[WithBlocks] => Code[WithBlocks]` (line 25), so it cannot accept or return any `VarAccess` nodes. Attempting to add a `VarAccess` node to the tree would result in a compile-time type error at the source of the problem in the implementation, instead of a runtime assertion failure much later in the pipeline, which would require debugging how the compiler generated the invalid `Code` tree.

```

1 def compileProcedure(currentProcedure: Procedure AndCode[WithCalls]):
2   Code[WithBlocks] =

```

```

3 {
4 // ...
5 val (code3: Code[WithVars], variables: Seq[Variable]) =
6     { /* eliminate procedure calls, and if-expressions */ }
7
8     val frame = Chunk( /* variables */ )
9
10    def eliminateVarAccessesA5(code: Code[WithVars]): Code[WithBlocks] =
11    {
12-    def fun: PartialFunction[Code, Code] = {
13+    code.transform {
14        case va: VarAccess =>
15            /* ... */
16            if va.read
17            then frame.load(Reg.framePointer, va.register, va.variable)
18            else frame.store(Reg.framePointer, va.variable, va.register)
19            /* ... */
20+    case other => other
21    }
22-    transformCode(code, fun)
23    }
24
25    def addEntryExit(code: Code[WithBlocks]): Code[WithBlocks] = {
26        val enter: Code[WithBlocks] = block(
27            /* ... */
28            // frame.store : (Reg, Variable, Reg) => Code[PWord]
29            frame.store(
30                Reg.result, // : Reg
31                currentProcedure.dynamicLink, // : Variable
32                Reg.framePointer), // : Reg
33            /* ... */
34        )
35        val exit: Code[WithBlocks] = block( /* ... */ )
36        block(Define(currentProcedure.label), enter, code, exit)
37    }
38
39    val code4 = eliminateVarAccessesA5(code3)
40    addEntryExit(code4)
41
42 }

```

Listing 5.12: High level view of modified compileProcedure for Assignment 5

Another case where precise typing prevents programmer errors is the implementation of the GarbageCollector. Listing 5.13 shows that a potential source of bugs, creating

Closures in during a collection cycle, is now disallowed; this is the case because all procedures used in the `GarbageCollector` are now typed as `WithCalls` but not `WithClosures` (line 3). Precise typing the `GarbageCollector` is necessary, but not sufficient, to ensure that it does not allocate objects on the heap during a collection cycle. The implementation of the `GarbageCollector` itself does not need to change, as it already adheres to these constraints.

```

1 object GarbageCollector extends MemoryAllocator with HeapSettings {
2
3   def procedures: Seq[ProcedureAndCode[WithCalls]] = /* ... */
4
5   // ...
6 }

```

Listing 5.13: Modified `GarbageCollector`

5.3.4 Static Type Conversions

Having a more precise type representation is not without cost, however. In the existing implementation, the phase invariants only need to be found correct at runtime, while in the new implementation, they need to be proven correct statically as well. An example of this is the function `eliminateComments` in Listing 5.14. The existing version filters out `Comment` nodes with a dynamic `isInstanceOf` check (line 5). However, the type of `filter` is `(Seq[A], A => Boolean) => Seq[A]`, meaning it cannot change the type of the returned sequence. In order to satisfy the type checker, it is necessary to `flatMap` over the nodes; this works because `flatMap` has type `(Seq[A], A => Seq[B]) => Seq[B]`. The new version must pattern match on the node (lines 6-9) and replace the undesired `Comment` node with an empty `Seq`.

```

1 def eliminateComments(
2   codes: Seq[Code[WithComments]]
3 ): Seq[Code[WithLabels]] =
4   codes
5   - .filterNot(_.isInstanceOf[Comment])
6   + .flatMap {
7   +   case Comment(message) => Seq() // Ignore comments
8   +   case code             => Seq(code)
9   + }

```

Listing 5.14: Existing and modified `eliminateComments`

5.3.5 Handling Variance

The existing implementation of `Procedure` (Listing 5.1, line 25) contains a mutable variable `code` to store the procedure's body. This presents a problem when dealing with variance as mutable variables are invariant, but `Code` is covariant. In addition to containing its body, `Procedure` also contains meta-information for calling and compiling itself, including its label and formal parameters. To represent recursion, the developer needs to create a `Procedure` instance (Listing 5.15, line 2) and modify it after creation (line 3).

```
1 // Before
2 val procedure = new Procedure( /* ... */ )
3 procedure.code = block(
4   // ...
5   Call(procedure, /* ... */)
6 )
7
8 // After
9 val procedure = new Procedure( /* ... */ )
10 val code = block(
11   // ...
12   Call(procedure, /* ... */)
13 )
14 val procedureAndCode = new ProcedureAndCode(procedure, code)
```

Listing 5.15: Example use of recursion

In the modified implementation of `Lacs`, the meta-information and the procedure body are separated: `Procedure` contains all the meta information, and `ProcedureAndCode` contains a `Procedure` and the `Code` tree (Listing 5.16). Since `Procedure` no longer contains `code`, it does not need to be polymorphic. `ProcedureAndCode` is immutable and can be covariant. With the modifications, the `Procedure` instance is created first (Listing 5.15, line 9), but the `Code` tree is created independently (line 10). The procedure and its body are only united when `ProcedureAndCode` is created (line 14).

```
1 class Procedure( /* ... */ ) {
2   - var code: Code = null
3
4   + def withCode [P](code: Code [P]): ProcedureAndCode [P] =
5     +   ProcedureAndCode(this, code)
6
7   // ...
8 }
9
10 case class ProcedureAndCode[+P](procedure: Procedure, code: Code [P]) {
```

```
11 // Forwarding methods to Procedure
12 // ...
13 }
```

Listing 5.16: Changes to Procedure

Chapter 6

Related Work

6.1 Recursion Schemes

Meijer et al. [23] introduce recursion schemes as a way to perform structured recursion. Recursion schemes provide various operators to perform different recursive algorithms, including *catamorphisms* and *anamorphisms*. The *catamorphism* operator accepts the base case and the collapsing function as parameters, while the *anamorphism* operator accepts an expanding function and a predicate. Our implementation represents *catamorphisms* and *anamorphisms* as `fold` and `unfold` respectively (Section 4). The *Recursion Schemes* approach can perform transformations on trees, but it makes no attempt at typing partially translated trees.

Implementations of recursion schemes [14] often make use of two types: a traditional recursive type, and a type representing one layer of the tree. This one-layer type is sometimes called a *base functor*, as it is a functor [12] over the inner nodes of the tree. Using this technique means duplicating every node in the tree and often requires the use of meta-programming to reduce boilerplate.

6.2 Data types à la carte

Swierstra [32] presents a technique for composing types from individual variants. The *Data types à la carte* technique involves representing each node as an open recursive type, composing nodes using a *coproduct* type, and tying the recursive knot using a fixpoint type.

This technique makes heavy use of Haskell’s type class [34] mechanism to construct and select the appropriate nodes in a tree, including advanced extensions such as overlapping instances¹. This is presented as a potential solution to the *Expression Problem* [33] by allowing the programmer to extend the original tree by defining new nodes later; this is not something we attempt to solve. Swierstra’s approach requires boilerplate to create smart constructors for each node and cannot make use of traditional pattern matching constructs, relying instead on creating new type classes for each operation on a node. By creating new type classes for each operation, the *à la carte* technique often requires the logic to be spread out across multiple type class instances instead of contained within a single function. Additionally, the use of overlapping type classes in their approach requires the use of top-level type annotations to aid in type class resolution, while our approach requires type annotations due to the use of subtyping and *GADTs*.

6.3 Trees that Grow

Najd et al. [26] present *Trees that Grow (TtG)*, a pattern for extensible trees in Haskell. Their approach allows for adding and removing variants from a tree, as well as adding extra fields to an existing variant. They accomplish this by making heavy use of type families², which are a way of expressing type level functions in Haskell. A key difference in the way that Haskell and Scala represent sum types is that in Scala, each variant is its own type; this allows the tree to be refined using subtyping instead of type families. The tree type is indexed by a type called an *extension descriptor* which is used as the input to the different type families, one for each variant. Their approach allows for adding extra fields to existing variants, something our approach cannot express. However, it is unable to represent a partially transformed tree, and thus cannot support generic traversals and folds over the tree. Another difference is that their approach assumes an open world, allowing the type definition to be extended after the fact, contrasting with our approach which requires a closed world to allow for exhaustiveness checking. Finally, though not strictly required, the *TtG* approach makes extensive use of pattern synonyms³ to make the patterns more ergonomic. The primary goal in their approach is to reduce duplication when declaring similar tree types, while ours is to perform tree transformations that are precisely-typed and have little boilerplate.

¹https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/instances.html#overlapping-instances

²https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/type_families.html

³https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/pattern_synonyms.html

6.4 Scrap Your Boilerplate

Scrap Your Boilerplate (SYB) is a technique presented by Lämmel and Jones [16] to traverse and transform mutually recursive datatypes. The technique makes use of runtime type information, dynamic type coercions, rank-2 types [18], and a one-layer map function to create generalized combinators for transforming and querying a tree. The use of a one-layer map is very similar to our use of `mapChildren` (Section 2.5), bringing the same benefits. One difference is that the *SYB* approach allows for transforming and querying any node in a tree as long as it implements the `Term` type class; this allows the programmer to map over general types that are present in the tree, such as `String`'s, `Int`'s and `List`'s. However, this technique does not change the type of the tree, nor does it allow the user to represent partially transformed trees.

6.5 Restrictable Variants

Restrictable Variants is an approach proposed by Madsen et al. [20] to allow the typing and tracking of non-exhaustive match expressions. The approach makes use of a type level set formula to allow the programmer to specify which variant labels are present or absent in a type. They introduce the *Co-domain Problem* to refer to the ability to capture the introduction and elimination of variants in a 'match'-expression. This technique closely aligns with our goal of typing which nodes may be present in a tree before and after a tree transformation phase. A benefit of this approach is that it supports full Hindley-Milner type inference [25, 10], something our match narrowing approach cannot. However, it is unable to express that a tree has been partially transformed, meaning that all transformations must be expressed as fully recursive functions. It is unclear if *Restrictable Variants* allows for a group of nodes to be named and referred to as one, or if the programmer must enumerate all nodes that are added/removed from the tree individually.

6.6 Nanopass Compilers

The Nanopass framework [31, 13] provides a technique to build a compiler out of numerous small transformation passes. A nanopass compiler defines multiple intermediate languages, usually as a derivation from a previously defined language; each compiler pass is then a small translation step between successive languages. A step only needs to define the interesting work done during the translation; the framework makes use of meta-programming

techniques to recursively apply it to all other nodes. This framework is originally implemented for Scheme and does not attempt to type each translation step.

Mercier et al. [24] present a typed implementation of the Nanopass framework for OCaml⁴. It uses one-layer nodes which are parameterized by their inner nodes to represent partially translated trees. Similarly to the original Nanopass framework, it makes extensive use of meta-programming to generate the different types and folds. Additionally, it makes extensive use of OCaml’s structurally typed polymorphic variants in order to allow the same nodes to appear in different types; it is unclear how to extend this to a nominally typed language like Scala.

6.7 Flow Typing

Flow typing provides a mechanism for variables to have different types at different points in the program depending on control-flow [28, 5]. This is often used in conjunction with negation and intersection types, and with runtime type tests. Under a flow type system, a variable x of type A can be checked to be of type B ; if the type test passes then x can be retyped to be of type $A \ \& \ B$ in the true branch, and $A - B$ in the false branch (where ‘ $-$ ’ is type subtraction).

Match narrowing as presented in Chapter 3 is an attempt at supporting flow typing within the constraints of the Scala type system. Since the scrutinee expression cannot have different types within the different ‘case’-branches, the ‘match’-expression introduces new variables (through patterns) that are of the desired type. This solution fits well with the existing Scala type system, which already contains ‘match’-expressions, and does not currently support a notion of flow typing or negation types.

6.8 Expression Problem

The *Expression Problem* refers to a challenge when designing statically typed languages [33]. When defining a datatype by cases, a language should allow programmers to add new functions on the datatype and new cases without recompiling the original code. Statically typed languages often either allow programmers to define a fixed set of cases and an open set of functions, or a fixed set of functions and an open set of cases. Precisely-typed trees

⁴<https://ocaml.org>

do not attempt to solve this problem. Match narrowing requires a closed-world assumption to ensure exhaustiveness when narrowing types in ‘match’-expressions; allowing the programmer to add new cases to the tree would violate this requirement.

6.9 Exhaustiveness Checking

Exhaustiveness checking is used to determine if the combination of patterns in a ‘match’-expression covers all possible values of the scrutinee expression. This is usually concerned with raising warnings if patterns are incomplete or overlapping [21]. A ‘match’-expression with a variable pattern is deemed exhaustive because any possible value can be matched against it. In contrast, match narrowing focuses on changing how the type checker elaborates the last variable pattern. As explained in Chapter 3, it elaborates the last variable pattern to the difference of the types of the previously matched patterns. Since exhaustiveness checking does not affect type elaboration, it can be performed during a separate compiler phase, whereas match narrowing cannot.

Liu [19] presents a technique to decouple exhaustiveness checking from a specific type system. They make use of “space” as an abstraction that represents values which may be present in an expression or covered by a pattern. However, once a type is converted to a space, it cannot be converted back to a type. Since match narrowing needs to determine if a pattern covers part of a type, it cannot make use of spaces directly.

Chapter 7

Conclusion

Many compilers are implemented using types that are far too broad. This thesis shows that it is practical to implement compilers with more precise tree types without code boilerplate or duplication. The techniques presented enabled trees to be typed with a superset of allowed nodes as well as to type nodes that have been partially transformed from one phase to another.

To accomplish this we introduced the concepts of type decomposition, type difference, and complete patterns to the Scala type checker. Together these concepts allow the scrutinee type to be progressively narrowed in a ‘match’-expression. Type decomposition allows a tree type to be decomposed into a union of node variants. Type difference enables the ability to restrict which variants of a union may be present in a term. Complete patterns are needed to allow the type checker to apply the type difference operator to subsequent patterns.

Precisely-typed tree transformations were extended with fold and unfold operations. We showed two possible ways to encode these operations, using union types and match types. The encoding with match types requires each node to have an extra type index to determine if it is recursive or not. The union types encoding forgoes this at the cost of requiring some extra η -expansions.

Finally, we demonstrated the practicality of this approach by modifying the existing Lacs compiler to restrict which nodes may be present at different compiler phases. This enables developers to catch frequent mistakes at the point in the source code they were introduced, instead of relying on runtime assertions at a later phase.

References

- [1] Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. The essence of dependent object types. *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 249–272, 2016.
- [2] David Aspinall and Adriana Compagnoni. Subtyping dependent types. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*, pages 86–97. IEEE, 1996.
- [3] Olivier Blanvillain, Jonathan Immanuel Brachthäuser, Maxime Kjaer, and Martin Odersky. Type-level programming with match types. *Proc. ACM Program. Lang.*, 6(POPL), January 2022.
- [4] Luca Cardelli. Type systems. *ACM Computing Surveys (CSUR)*, 28(1):263–264, 1996.
- [5] Giuseppe Castagna, Mickaël Laurent, Kim Nguyen, and Matthew Lutze. On type-cases, union elimination, and occurrence typing. *Proc. ACM Program. Lang.*, 6(POPL):1–31, 2022.
- [6] Oege de Moor, Jeremy Gibbons, and Geraint Jones. *The Fun of Programming*. Palgrave Macmillan, 2003.
- [7] Sébastien Doeraene. Sip-56 - proper specification for match types. <https://docs.scala-lang.org/sips/match-types-spec.html>, 2023. Accessed: 2024-12-05.
- [8] gkepka, WilliamTakeshi, PhoenixmitX, artemkorsakov, raulsorrentino, mhghafari, julienrf, adpi2, JurgisSi, ckipp01, BaeJi77, mlachkar, mehuled, ashawley, puorc, asakaev, bkellerman, Michael Kinsey, and heathermiller. Pattern matching: Sealed types. <https://docs.scala-lang.org/tour/pattern-matching.html#sealed-types>, 2017. Accessed: 2024-11-19.

- [9] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The L^AT_EX Companion*. Addison-Wesley, Reading, Massachusetts, 1994.
- [10] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.
- [11] Graham Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, 1999.
- [12] Mark P Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 52–61, 1993.
- [13] Andrew W Keep and R Kent Dybvig. A nanopass framework for commercial compiler development. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 343–350, 2013.
- [14] Edward A. Kmett. recursion-schemes: Representing common recursion patterns as higher-order functions. <https://hackage.haskell.org/package/recursion-schemes>, 2008. Accessed: 2024-11-08.
- [15] Donald Knuth. *The T_EXbook*. Addison-Wesley, Reading, Massachusetts, 1986.
- [16] Ralf Lämmel and Simon L. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In Zhong Shao and Peter Lee, editors, *Proceedings of TLDI’03: 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, New Orleans, Louisiana, USA, January 18, 2003*, pages 26–37. ACM, 2003.
- [17] Leslie Lamport. *L^AT_EX — A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, second edition, 1994.
- [18] Daniel Leivant. Polymorphic type inference. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 88–98, 1983.
- [19] Fengyun Liu. A generic algorithm for checking exhaustivity of pattern matching (short paper). In Aggelos Biboudis, Manohar Jonnalagedda, Sandro Stucki, and Vlad Ureche, editors, *Proceedings of the 7th ACM SIGPLAN Symposium on Scala, SCALA@SPLASH 2016, Amsterdam, Netherlands, October 30 - November 4, 2016*, pages 61–64. ACM, 2016.

- [20] Magnus Madsen, Jonathan Lindegaard Starup, and Matthew Lutze. Restrictable variants: A simple and practical alternative to extensible variants. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States*, volume 263 of *LIPICs*, pages 17:1–17:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.
- [21] Luc Maranget. Warnings for pattern matching. *Journal of Functional Programming*, 17(3):387–421, 2007.
- [22] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.
- [23] Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer, 1991.
- [24] Daniel Mercier and Boris Yakobowski. Type-safe nanopasses: How to write a safe and modern compiler front-end. <https://www.youtube.com/watch?v=a7fLyZgrk0o>, 2024. Accessed: 2024-11-08.
- [25] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.
- [26] Shayan Najd and Simon Peyton Jones. Trees that grow. *CoRR*, abs/1610.04799, 2016.
- [27] Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored local type inference. In Chris Hankin and Dave Schmidt, editors, *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 41–53. ACM, 2001.
- [28] David J. Pearce. Sound and complete flow typing with unions, intersections and negations. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, volume 7737 of *Lecture Notes in Computer Science*, pages 335–354. Springer, 2013.

- [29] Marianna Rapoport, Ifaz Kabir, Paul He, and Ondřej Lhoták. A simple soundness proof for dependent object types. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–27, 2017.
- [30] Tiark Rumpf and Nada Amin. Type soundness for dependent object types (dot). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 624–641, 2016.
- [31] Dipanwita Sarkar, Oscar Waddell, and R Kent Dybvig. A nanopass infrastructure for compiler education. *ACM SIGPLAN Notices*, 39(9):201–212, 2004.
- [32] Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, 2008.
- [33] Philip Wadler. The expression problem. <https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt>, 1998. Accessed: 2024-11-08.
- [34] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, 1989.