Portland State University
PDXScholar

Dissertations and Theses Dissertations and Theses

1-1-2010

Practical Type Inference for the GADT Type System

Chuan-kai Lin
Portland State University

Let us know how access to this document benefits you.
Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Recommended Citation
Lin, Chuan-kai, "Practical Type Inference for the GADT Type System" (2010). *Dissertations and Theses.* Paper 367.

10.15760/etd.367

This Dissertation is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. For more information, please contact pdxscholar@pdx.edu.

Practical Type Inference for the GADT Type System

by

Chuan-kai Lin

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Computer Science

Dissertation Committee:
Tim Sheard, Chair
Andrew P. Black
James Hook
Mark P. Jones
Andrew Tolmach
Douglas V. Hall

Portland State University
© 2010

ABSTRACT

Generalized algebraic data types (GADTs) are a type system extension to algebraic data types that allows the type of an algebraic data value to vary with its shape. The GADT type system allows programmers to express detailed program properties as types (for example, that a function should return a list of the same length as its input), and a general-purpose type checker will automatically check those properties at compile time. Type inference for the GADT type system and the properties of the type system are both currently areas of active research.

In this dissertation, I attack both problems simultaneously by exploiting the symbiosis between *type system* research and *type inference* research. Deficiencies of GADT type inference algorithms motivate research on specific aspects of the type system, and discoveries about the type system bring in new insights that lead to improved GADT type inference algorithms. The technical contributions of this dissertation are therefore twofold: in addition to new GADT type system properties (such as the prevalence of pointwise type information flow in GADT patterns, a generalized notion of existential types, and the effects of enforcing the GADT branch reachability requirement), I will also present a new GADT type inference algorithm that is significantly more powerful than existing algorithms. These contributions should help programmers use the GADT type system more effectively, and they should also enable language implementers to provide better support for the GADT type system.

DEDICATION

To my wife Hwa-yi,

Who, like Powdermilk Biscuits,
Gives me the strength to get up
And do what needs to be done.

ACKNOWLEDGMENTS

The work I present in this dissertation is entirely my own, but I could not have done it without the assistance and support of many others. I would like to take this opportunity to express my everlasting gratitude.

Tim Sheard is a wonderful advisor. He believes that students should be free to pursue their own interests, and his support was invaluable during the difficult period when I kept digging and nothing turned up. We do not always agree on technical issues, but, after giving me his honest opinion, he always let me take the research in the direction of my choosing. He has a real talent for identifying important ideas, and we spent many hours sitting in his office trying to find the key sentence that was left unsaid in my confused writing. He worked as hard as I did in the months leading up to my defense, reading and marking up drafts at an incredible pace. I hope the end result makes him proud.

Andrew P. Black took me under his wings when I first started my graduate studies and taught me how to do research. I still remember when he told me "Chuan-kai, you stuttered because you put too many words on your slides, and you were having trouble finding different words to say the same things." That was quite a revelation.

The members of my dissertation committee — James Hook, Mark P. Jones, and Andrew Tolmach — worked diligently to help me improve this dissertation. Aaron Stump and Dimitrios Vytiniotis also provided useful feedback on a draft. This dissertation would certainly be a lot poorer without their insight.

I feel extremely lucky to have shared a cubicle with Emerson Murphy-Hill during pretty much the entirety of his doctoral studies. His can-do spirit and unorthodox wit are positively contagious, and his generous and engaging personality inspired me to be a better person. I am proud and honored to call Emerson my friend, and I spend much time wondering how much better the world would be if only there are more people like him.

I met Nancy Murphy when Emerson invited me to his family Thanksgiving gathering in 2004. She treats me like her own child, and her home becomes my favorite sanctuary whenever I feel weary and discouraged. She gave me a family in this foreign land, which, because of her, is now a much less foreign place with very many happy memories.

I am indebted to my parents in many different ways. Though they rarely knew what I was up to (or so I thought), they always supported my interests and my decisions without reservation. It was only with their support that I had an opportunity to develop my interests in computer science and to pursue graduate studies in the United States. Thanks Mom and Dad. I love you.

Above all, I want to thank my wife Hwa-yi Tang for her love, sacrifice, and encouragement. Maintaining a long-distance relationship takes a lot of hard work, but she took it in stride because she knows how much I love research. She makes my life complete and gives me the strength to do what needs to be done.

Academic research relies on the support of a stable, mature, and resourceful society, and I feel privileged to enjoy such support. Many people before me have worked hard to make the world a better place, and, one way or the other, they all contributed to the birth of this dissertation.

Seven years. It was an amazing adventure, and I treasure every moment of it.

CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

In this dissertation, I present my research on the type inference problem for generalized algebraic data types (GADTs). This chapter introduces the role of types in programming languages, motivates generalized algebraic data types with two examples of expression evaluators, and outlines both the technical challenges and the contributions of this dissertation.

1.1 TYPE SYSTEMS

Type systems are a programming language feature that consists of two parts. The first part of a type system is a formal language — the *language of types* — that describes a set of properties that programs may have. The second part of a type system is a set of *type rules* that defines a mathematical relation between types and programs. These two parts work together to provide a formal framework for describing program properties.

The first part of a type system, its language of types (or just *types* for short), names program properties that are relevant to the type system. For example, here are four types in the type system for the Haskell 98 programming language [14]:

`Int`	Integer
`[Int]`	List of integers
`Int → Bool`	Function that maps integers to Boolean values

`forall a. [a] → Int`	Function that, for any type `a`, maps lists of `a` elements to integers

One can think of types as a form of program documentation: like code comments, they provide a way for programmers to describe program properties. However, unlike code comments, which are free-form, types have a formally defined syntax, which specifies what can and cannot be expressed through types.

The second part of a type system, its set of type rules, defines how programs relate to types. In other words, the type rules define which programs have which types in the type system. For example, here are four Haskell 98 programs and (to the right of the `::` symbol in each line) the types they have in the type system of Haskell 98:

```
7+6 :: Int
7 == 6 :: Bool
λx → [x] :: Int → [Int]        -- One type of λx → [x]
λx → [x] :: Char → [Char]      -- Another type of λx → [x]
```

For example, you read the first line as "the program `7+6` has type `Int`." This kind of mathematical relation between programs and types is what makes types fundamentally different from code comments. Unlike comments, which may be ambiguous, or even wrong (with respect to the code that they describe), types are precise and can be checked for consistency with the programs that they describe. Such is the power of formalism.

In many type systems, the language of types is extensible: programmers can add new vocabulary to the language by defining new data types. For example, here is how one may define an algebraic data type of trees in Haskell 98:

```
data Tree a = Tip a | Fork (Tree a) (Tree a)
```

The declaration consists of two parts that are separated by the `=` symbol. The first names the data type (`Tree`) and its type argument (`a`). The second defines the two data constructors (`Tip` and `Fork`) that constitute the `Tree` data type.

This declaration of the `Tree` data type extends the language of types so that programmers can use types to describe programs that involve trees:

`Tree Bool`	Tree of Boolean values
`forall a. Tree a → a`	Function that, for any type `a`, maps trees of `a` elements to a value of type `a`

The declaration of the `Tree` data type also extends the mathematical relation between programs and types to cover programs that involve trees:

```
Tip 5 :: Tree Int
Tip (7 == 6) :: Tree Bool
Fork (Tip 'a') (Tip 'c') :: Tree Char
```

Since the type rules of a type system define a mathematical *relation* between programs and types, they do not require that every program must have exactly one type. The `λx → [x]` example (p. 2) demonstrates that a program may have multiple types; at the other end of the spectrum, there may also be programs that have *no* types. This possibility to deny a program a type provides a convenient way for programming language designers to specify, with mathematical precision, which programs they consider *good* and which ones they consider *bad*: formulate the type rules so that only good program are well-typed.[1]

From this perspective, a type system is a reflection of its designers' view on which programs are good and which ones are bad, and a different criterion of good vs. bad might lead to a different type system. One extremely common criterion is

[1] A well-typed program is a program that has at least one type.

to consider a program “good” if it has a well-defined dynamic semantics,[2] or, in other words, if its runtime behavior is well-defined. Using a type system to enforce this criterion leads to the long-established notion of *type soundness* (also called *type safety*): a well-typed program should not go wrong [27, §3].[3]

Type soundness is an important design goal for type systems because a sound type system is not only a formalism for program documentation, but also a formalism for *program verification*. In a programming language with a sound type system, even if a programmer does not know what types a program might have, merely knowing that the program has a type gives the programmer some basic assurance about the runtime behavior of the program. The type system for the Haskell 98 programming language is generally believed to be sound. However, this belief cannot be proved because there is no completely formal description of either the type system or the semantics for Haskell 98.

I should point out that, in addition to programming languages, type systems also play an important role in mathematical logic and in proof systems. Russell, for example, developed a *theory of types* to avoid logical paradoxes (that are due to impredicativity) in a logical foundation of mathematics [51]. Another example is the Curry-Howard correspondence [41], which suggests that typed programs are equivalent to constructive proofs. This correspondence led to the Coq proof assistant software [45], which allows its users to prove logical propositions by writing typed functional programs. These roles of type systems are fascinating topics but lie outside the scope of this dissertation.

[2] The dynamic semantics of a program is the result of running the program. In contrast, the static semantics of a program is a property (such as its type) that can be determined statically. The word “semantics” alone usually refers to dynamic semantics.

[3] The meaning of “go wrong” depends on the semantics of the programming language, which may consider certain exceptions (such as divide-by-zero) as well-defined behavior. As a result, the soundness of a type system is defined only with respect to a semantics of the language.

1.2 TYPE SYSTEM INCOMPLETENESS

Type system *soundness* (*i.e.,* a well-typed program should not go wrong) is a powerful idea. Type system *completeness* (*i.e.,* a program that cannot go wrong should be well-typed) is likewise a powerful idea. If a type system is both sound and complete with respect to an untyped dynamic semantics of the programming language, a program would be well-typed if and only if it cannot go wrong at runtime. This property makes a sound and complete type system a very valuable addition to a programming language.

Unfortunately, designing a practical type system that is sound and complete with respect to an untyped dynamic semantics of a general-purpose programming language turns out to be a very difficult problem. More specifically, results in computability theory (such as Rice's Theorem [34]) suggest that our inability to come up with a sound, complete, and practical type system for a Turing-complete programming language may be due to a fundamental limit of formal reasoning. Therefore, as a (necessary) practical compromise, type system researchers overwhelmingly choose to design type systems that are sound but incomplete. It is better to be safe (by rejecting some good programs) than to be sorry (by accepting some bad programs).

In a sound type system, all bad programs are ill-typed (*i.e.,* not well-typed). A programmer must take care to write only well-typed programs because implementations of a typed programming language (such as Haskell), in an attempt to prevent runtime errors, typically refuse to run (or to compile) ill-typed programs. However, in an incomplete type system, some good programs are also ill-typed. If a programming language implementation requires all programs to be well-typed in an incomplete type system, the programmer may be compelled to modify a perfectly good program just for the sake of satisfying the (incomplete) type system. As the following example shows, the need to appease an incomplete type system

$$i := \ldots, -2, -1, 0, 1, 2, \ldots$$

$$e := T \mid F \mid i \mid e \oplus e \mid e \leqslant e \mid (e, e) \mid \mathrm{fst}(e) \mid \mathrm{snd}(e)$$

$$\begin{array}{rl}
[\![T]\!] = \text{True} & \text{Boolean value} \\
[\![F]\!] = \text{False} & \text{Boolean value} \\
[\![i]\!] = i & \text{Integer} \\
[\![i \oplus j]\!] = [\![i]\!] + [\![j]\!] & \text{Integer addition} \\
[\![i \leqslant j]\!] = [\![i]\!] \leq [\![j]\!] & \text{Integer comparison} \\
[\![(x, y)]\!] = \langle [\![x]\!], [\![y]\!] \rangle & \text{Cartesian product} \\
[\![\mathrm{fst}(c)]\!] = \pi_1 [\![c]\!] & \text{Product left projection} \\
[\![\mathrm{snd}(c)]\!] = \pi_2 [\![c]\!] & \text{Product right projection}
\end{array}$$

Figure 1.1: This figure shows the syntax and the semantics of a small expression language e. The symbols $+$, $\leq$, $\langle\ \rangle$, π_1, and π_2 represent mathematical operators.

can make programming unnecessarily awkward.

Figure 1.1 (p. 6) defines an expression language e of integers, Boolean values, and Cartesian products. The operator $[\![\cdot]\!]$ provides a denotational semantics for the language by mapping expressions to mathematical objects. For example, here is how one calculates the meaning of a program in this language:

$$[\![\mathrm{snd}((2 \leqslant 5, 3))]\!] = \pi_2 [\![(2 \leqslant 5, 3)]\!] = \pi_2 \langle [\![2 \leqslant 5]\!], [\![3]\!] \rangle = \pi_2 \langle \text{True}, 3 \rangle = 3$$

The meaning of the program $\mathrm{snd}((2 \leqslant 5, 3))$ is the (mathematical) integer 3. How does one implement this expression language in Haskell 98?

Figure 1.2 (p. 7) shows an algebraic data type `Term`, whose structure directly corresponds to the syntax of the expression language e in Figure 1.1 (p. 6). The

```
data Term
    = RepT                      -- T
    | RepF                      -- F
    | RepInt Int                -- i
    | RepPlus Term Term         -- i ⊕ j
    | RepLeq Term Term          -- i ⩽ j
    | RepPair Term Term         -- (x, y)
    | RepFst Term               -- fst(c)
    | RepSnd Term               -- snd(c)
```

Figure 1.2: Algebraic data type for the expression language.

`Term` data type defines a concrete representation of programs in the expression language. For example, here is the representation of $\mathrm{snd}((2 \leqslant 5, 3))$:

```
RepSnd (RepPair (RepLeq (RepInt 2) (RepInt 5)) (RepInt 3))
```

Figure 1.3 (p. 8) shows the `eval` function, which evaluates programs in the expression language. The `eval` function uses `case` expressions to return different results depending on the structure of the expression. In a `case` expression, the sub-expression between `case` and `of` is its *scrutinee*, and the part after `of` is its *pattern-matching branches.* In a pattern-matching branch, the part to the left of the arrow (→) is its *pattern*, and the part to the right of the arrow is its *body.* A `case` expression evaluates to the body of the first branch whose pattern matches the structure of the scrutinee.

Since the evaluation result can have one of many types (*e.g.,* it can be an integer, a Boolean value, or a pair), I defined an auxiliary data type `Value` to encapsulate the evaluation result. Each data constructor of the `Value` data type (`ValBool`, `ValInt`, and `ValPair`) *tags* a value so that the type system can recover

```
data Value
  = ValBool Bool
  | ValInt Int
  | ValPair (Value, Value)

eval :: Term → Value
eval e = case e of
  RepT → ValBool True
  RepF → ValBool False
  RepInt i → ValInt i
  RepPlus i j → case eval i of
    ValInt vi → case eval j of
      ValInt vj → ValInt (vi + vj)
  RepLeq i j → case eval i of
    ValInt vi → case eval j of
      ValInt vj → ValBool (vi <= vj)
  RepPair x y → ValPair (eval x, eval y)
  RepFst c → case eval c of
    ValPair vc → fst vc
  RepSnd c → case eval c of
    ValPair vc → snd vc
```

Figure 1.3: Algebraic data type evaluator for the expression language.

the type of the value at a later time. The `eval` function matches its parameter `e` to the data constructors of the `Term` data type and evaluates each branch using the semantics of the expression language (Figure 1.1, p. 6). Even though the `eval` function faithfully implements the semantics of the expression language, it is not ideal for the following two reasons:

1. Encapsulating the evaluation result in the `Value` data type (*i.e., tagging*) is cumbersome. Both injecting results into and projecting results from the `Value` data type incur runtime overhead and clutter up the implementation. Due to its use of the `Value` type, this `eval` function is typically referred to as a *tagged* evaluator.

2. Applying `eval` to a program whose semantics is undefined causes a runtime pattern-matching failure. For example, applying `eval` to snd(3) (which is represented by `RepSnd (RepInt 3)`) causes a pattern-matching failure because `eval (RepInt 3)` returns `ValInt 3`, but the `case` expression for the result (in the last two lines of `eval`) expects a `ValPair` value. Adding error-handling code to avoid pattern-matching failures (not shown in Figure 1.3) clutters up the implementation even further.

These two aberrations have different causes. I introduced `Value` to appease the Haskell 98 type system, which does not allow a function to return an integer in some situations and a Boolean value in others. Runtime pattern-matching failures can occur because the `Term` data type is too permissive: it allows programmers to represent both *good* expressions (those whose semantics are defined according to Figure 1.1, p. 6) and *bad* expressions (those whose semantics are undefined). For example, snd(3) is a bad expression because its semantics is undefined: integers do not have a right projection.

$$[\![\mathrm{snd}(3)]\!] = \pi_2[\![3]\!] = \pi_2(3) = \,?$$

This inability to distinguish good expressions from bad ones is inherent in the formulation of algebraic data types, which is a feature of the Haskell 98 type system that governs user-defined types (such as `Tree` and `Term`).

1.3 GENERALIZED ALGEBRAIC DATA TYPES

To address these deficiencies, researchers recently proposed *generalized algebraic data types* (GADTs) [17], which extend algebraic data types by allowing the *type* of an algebraic data value to vary with the *structure* of the value. Figure 1.4 (p. 11) shows a Haskell implementation of the expression language that uses the GADT extension in the Glasgow Haskell Compiler [44, §7.5]. In this new implementation, the `Term` data type has a type argument that represents the type of the value that an expression should evaluate to. For example, here are three expressions in the GADT `Term` data type:

```
RepT :: Term Bool
RepPair RepT (RepInt 3) :: Term (Bool, Int)
RepSnd (RepPair RepT (RepInt 3)) :: Term Int
```

The type argument of `Term` also allows the (Haskell) type system to reject bad expressions. For example, the following two expressions are ill-typed:

```
RepSnd (RepInt 3)          -- snd can be applied only to pairs
RepPlus RepT (RepInt 3)    -- ⊕ can add only integers
```

In a way, the GADT `Term` data type extends the Haskell type system to cover the expression language of Figure 1.1 (p. 6) so that only good expressions have well-typed representations in the Haskell type system. The `eval` function for this GADT `Term` data type (Figure 1.4, p. 11) resembles the denotational semantics of the expression language (Figure 1.1, p. 6) and has neither of the problems in the previous implementation (Figure 1.3, p. 8):

```
data Term a where
  RepT    :: Term Bool
  RepF    :: Term Bool
  RepInt  :: Int → Term Int
  RepPlus :: Term Int → Term Int → Term Int
  RepLeq  :: Term Int → Term Int → Term Bool
  RepPair :: forall a b. Term a → Term b → Term (a,b)
  RepFst  :: forall a b. Term (a,b) → Term a
  RepSnd  :: forall a b. Term (a,b) → Term b

eval :: forall a. Term a → a
eval e = case e of
  RepT → True
  RepF → False
  RepInt i → i
  RepPlus i j → eval i + eval j
  RepLeq i j → eval i <= eval j
  RepPair x y → (eval x, eval y)
  RepFst c → fst (eval c)
  RepSnd c → snd (eval c)
```

Figure 1.4: Expression language in generalized algebraic data types.

1. In the type `forall a. Term a` → `a` of the `eval` function, the type of the evaluation result is now matched to the type argument `a` of the type (`Term a`) of the input expression, so there is no need to encapsulate (*i.e.,* to tag) the evaluation result in the `Value` data type. For this reason, this GADT `eval` function is typically referred to as a *tagless* evaluator.

2. The GADT `Term` data type can represent only good expressions, so a bad expression can no longer cause runtime failures in the `eval` function.

Generalized algebraic data types extend the Haskell 98 type system to recognize a wider range of good programs by tracking fine-grained type information (such as the expected result type of evaluating an expression). More specifically, they do so by allowing programmers to use types to track inductive properties of the shape of a data value. This ability to recognize more good programs reduces the need to appease the type system and allows programmers to implement the expression evaluation function in a style that is both more natural and more efficient.

Generalized algebraic data types are a general-purpose type system feature, and tagless expression evaluators are only one of their many applications, which also include balanced trees [38, §4.1], generic programming [37, §5.3], and monad libraries [23]. Generalized algebraic data types are very useful, and type system researchers have been working hard to improve GADT support in programming language implementations. In this dissertation, I present my research in one area of GADT support in programming language implementations: type inference for programs that use generalized algebraic data types.

1.4 TYPE INFERENCE

To bring the theoretical benefits of type systems into practice, an implementation of a typed programming language must accept only well-typed programs (which are definitely good) and reject all ill-typed programs (which are potentially bad).

In other words, the implementation must be able to distinguish, at compile time, well-typed programs from ill-typed ones. A language implementation typically identifies well-typed programs by either *type checking* or *type inference*:

Type checking algorithms check if a program has a specific type in a specific context. Since a type checker[4] checks a program against only one specific type, a language implementation that employs type checking must rely on programmer type annotations to inform the type checker what the expected type of a program is.

Type inference algorithms decide if a program is well-typed in a specific context, and, if so, then compute a type for that program. A language implementation that employs type inference does not require programmer type annotations because a type inference algorithm, by design, should consider all possible types when deciding if a program is well-typed.

Type inference reduces programmer workload because it allows language implementations to accept programs that do not have type annotations. This feature is especially useful in early stages of software development where types tend to be in a constant state of flux. Unfortunately, a type inference algorithm is much more difficult to design than a type checking algorithm because it must decide if a program is well-typed without knowing the expected type of the program. The type inference problem for GADT programs is particularly difficult: leading experts in the field have studied the problem [17, 32, 36, 42, 43], but there are still significant weaknesses with all of the solutions that have been proposed to date. For example, no previous type inference algorithms can infer a type for `eval` (Figure 1.4, p. 11) without the assistance of a programmer type annotation.

[4] A type checking algorithm is also commonly referred to as a *type checker*.

1.5 TECHNICAL CHALLENGES

What makes type inference for the GADT `eval` function so difficult? Previous work in GADT type inference points to two reasons:

1. In the `eval` function, different pattern-matching branches in the same `case` expression might have different types. For example, the body (`True`) of the `RepT` branch has type `Bool`, but the body (`i`) of the `RepInt` branch has type `Int`. The different types of different pattern-matching branches complicate the GADT type inference problem because a type inference algorithm must try to reconcile the differences.

2. The `eval` function uses polymorphic recursion [29], which allows it to invoke itself on an instance of its own type. For example, the first recursive call (`eval i`) in the body of the `RepLeq` branch assumes that `eval` has type `Term Int` → `Int`, which is an instance of the type `forall a. Term a` → `a` of the `eval` function. Since type inference with polymorphic recursion is computationally undecidable [12, 21], one may expect type inference for functions such as `eval` to be very difficult.

Note that `eval` is only one specific example of a GADT program, and this list of two technical difficulties is by no means exhaustive. Type inference for some other practical (and also seemingly reasonable) GADT programs may be difficult for entirely different reasons, which include:

3. A well-typed GADT function need not have a single most-general type; in fact, there are GADT functions that have infinitely many maximal types.[5] The existence of multiple maximal types complicates the GADT type inference problem because it suggests that one single type cannot adequately

[5] A type t of a program is *maximal* if the program does not have another type that is strictly more general than t. A program has a most-general type if it has exactly one maximal type.

represent all the type information that a type inference algorithm inferred from a program expression.

4. The type of a `case` scrutinee may depend on the structures of the GADT pattern-matching branches in the `case` expression. The dependency complicates the GADT type inference problem because it suggests that a type inference algorithm should propagate type information from a GADT pattern-matching branch to the `case` scrutinee in accordance with the dependency, but the dependency itself is rather difficult to characterize.

As far as I know, the extent of these last two technical difficulties had never been fully described in previous work. Paraphrasing Donald Rumsfeld, these unknown unknowns — the things we did not know that we did not know — were perhaps the most important reasons why GADT type inference was so difficult.

1.6 CONTRIBUTIONS

In this dissertation, I present Algorithm $\mathcal{P}$, a novel type inference algorithm for programs that use generalized algebraic data types. While Algorithm $\mathcal{P}$ does not completely solve the aforementioned technical difficulties (§1.5), it does represent a significant step toward understanding and solving these difficulties.

In contrast with previous algorithms that rely on type annotations [17, 32], I designed Algorithm $\mathcal{P}$ to infer types for programs without type annotations. Compared with previous algorithms that do not rely on type annotations [36, 42, 43], Algorithm $\mathcal{P}$ infers types for a much larger set of GADT programs, including the GADT `eval` function in Figure 1.4 (p. 11), which lies beyond the capabilities of all previous GADT type inference algorithms.

The design of Algorithm $\mathcal{P}$ relies heavily on the discoveries about generalized algebraic data types that I made in the course of this dissertation research. Here are two examples:

- The traditional notion of existential types [22] is insufficient to explain why some GADT programs are ill-typed. I extended existential types to *generalized existential types* (§4.2), which not only offers greater explanatory power than existential types, but also helps Algorithm $\mathcal{P}$ to decide when it should propagate type information from a GADT pattern-matching branch to the type of the `case` scrutinee. (In contrast, the OutsideIn type inference algorithm [36] never propagates type information in this direction, and its type inference power suffers as a result.)
- Contrary to common belief, a more-general type of a GADT program is *not necessarily* preferable to a less-general type. Making a `case` scrutinee type overly general brings no practical benefit to programmers but allows more opportunities for runtime pattern-matching failures. I proposed a criterion for deciding whether a `case` scrutinee type is overly general (§6.2) and used it to prevent Algorithm $\mathcal{P}$ from inferring overly-general scrutinee types.

I will discuss these discoveries in detail in this dissertation.

1.7 OUTLINE

I organized the remainder of this dissertation into seven chapters.

Chapter 2 provides background information on algebraic data types, generalized algebraic data types, and the state of the art of the GADT type inference problem. In §2.5, I classify the types related to a GADT pattern-matching branch into six distinct type roles and describe an intriguing symmetry between these six type roles.

Chapter 3 introduces the Pointwise GADT type system, which I designed to distinguish practical GADT programs from a specific class of pathological programs that have counter-intuitive types. It restricts the GADT type system of Peyton Jones et al. [17] by requiring type information flow in a GADT pattern to follow the pointwise structure, which I make precise by formally defining *pointwise unifiers* and *pointwise unification.*

Chapter 4 describes the Non-Dependent GADT type system, which further restricts the Pointwise GADT type system. I show that, contrary to conventional wisdom, GADT type refinements are not the only feature that makes GADT type inference difficult, and I propose the notion of *generalized existential types* to explain how the type of a `case` scrutinee may depend on the structures of the corresponding pattern-matching branches.

Chapter 5 is devoted to the GADT branch reachability requirement, which is a GADT type system design choice that requires every pattern-matching branch to be potentially reachable. I discuss the pros and cons of this choice and show how it breaks type preservation in GADT type systems. This chapter also illustrates why restricting a type system need not necessarily make the type inference problem easier.

Chapter 6 describes Algorithm $\mathcal{P}$. I explain why Algorithm $\mathcal{P}$, by design, does not always infer the most-general type even when it exists: the most-general type of a program is not necessarily its best type. I also explain how Algorithm $\mathcal{P}$ uses GADT type refinements to reconcile the type inconsistencies between GADT pattern-matching branches in a `case` expression.

Chapter 7 describes my Haskell implementation of Algorithm $\mathcal{P}$ and evaluates its type inference capability using 30 GADT programs that I selected from a wide range of application domains. Algorithm $\mathcal{P}$ infers types for 25 out of the 30 programs. In contrast, OutsideIn [36] infers a type for only 1 out of the same 30 programs. Algorithm $\mathcal{P}$ is incomplete,[6] and I discuss the technical

[6] Soundness and completeness of a *type inference algorithm* are different from soundness and

difficulties that lead to the five type inference failures in the evaluation.

Chapter 8 summarizes the contributions of this dissertation and discusses topics of the dissertation that will most likely benefit from future work.

completeness of a type system. For a type inference algorithm, soundness means that the algorithm accepts *only* well-typed programs, and completeness means that the algorithm accepts *all* well-typed programs. Therefore, the incompleteness of Algorithm $\mathcal{P}$ states that Algorithm $\mathcal{P}$ does not accept some well-typed programs.

Chapter 2

BACKGROUND

In this chapter, I introduce basic concepts such as programs and types, present the ADT and the GADT type systems, and discuss the current state of the GADT type inference problem.

2.1 NOTATION AND SYNTAX

The general area of this dissertation is programming languages and type systems, so I start by introducing the notation and the syntax for programs and types.

Programs and types both consist of symbols in specific syntactic categories. For example, an expression may contain variables and data constructors, and a type may contain type variables and type constructors. Figure 2.1 (p. 20) describes the meta-symbols that I use in this dissertation; each meta-symbol represents an element in the corresponding syntactic category. For example, the meta-symbol u represents a type (but not any specific type). Note that meta-symbols for type constructors range not only over user-defined ones, but also over built-in type constructors such as arrow (`Arrow`), pair (`Pair`), and triple (`Triple`). Figure 2.2 (p. 20) lists the mathematical notation I adopted in this dissertation. Note that I use italic uppercase letters for both type constructors and sets; the intended meaning should be clear from the context.

The ways that programmers can construct expressions from variables and data constructors (or types from type variables and type constructors) are specified by the *syntax* of programs and types. Figure 2.3 (p. 20) shows the syntax of types

a, b, c, …	Expressions
C, D, E, …	Data constructors
u, v, w, …	Variables
u, v, w, …	Types
S, T, U, …	Type constructors
α, β, γ, …	Type variables
$\mathbb{X}_\alpha$, $\mathbb{X}_\beta$, $\mathbb{X}_\gamma$, …	Skolem type constants
θ, η, σ, …	Type substitutions

Figure 2.1: Legend for program meta-symbols.

$\overline{a}$	A finite sequence of a entities (or the set of elements in the sequence)
$[\overline{s/\alpha}]$	The substitution of $\overline{s}$ for $\overline{\alpha}$
$\mathrm{dom}(\theta)$	The domain of the substitution θ
$mgu(s \sim t)$	A most-general unifier of s and t
$\mathcal{U}(s \sim t)$	Unification of s and t
$tyvar(s)$	The free type variables of s
$S \mathbin{\#} T$	Sets S and T are disjoint
$S \uplus T$	The union of disjoint sets S and T

Figure 2.2: Mathematical notation.

$$u = \alpha \mid T\,\overline{u}$$

$$\mathrm{e} = \mathrm{u} \mid \mathrm{C} \mid \lambda\mathrm{u}\,.\,\mathrm{e} \mid \mathrm{e}\ \mathrm{e} \mid \texttt{let}\ \mathrm{u} = \mathrm{e}\ \texttt{in}\ \mathrm{e} \mid \texttt{case}\ \mathrm{e}\ \texttt{of}\ \{\overline{\mathrm{T}\ \overline{\mathrm{u}} \to \mathrm{e}}\}$$

Figure 2.3: Syntax of well-formed types and programs.

and programs I use in this dissertation. These syntax definitions are standard for a statically-typed functional programming language. A type is either a type variable (α), or a type constructor (T) with a sequence of types ($\overline{u}$) as its arguments. Following convention, I require each type constructor to have a fixed arity (*i.e.,* a specific type constructor is always applied to the same number of arguments). Under this syntax, the types of functions, pairs, triples, quadruples, *etc.* have the following forms:

`Arrow` u v	Function from u to v
`Pair` u v	Pair of u and v
`Triple` u v w	Triple of u, v, and w
`Quad` u v w s	Quadruple of u, v, w, and s

These forms adhere completely to the syntax of types, but they are also awkward to write and difficult to read. In the interest of readability, I will instead present these types using the following conventional notation:

$u \rightarrow v$	Function from u to v
(u, v)	Pair of u and v
(u, v, w)	Triple of u, v, and w
(u, v, w, s)	Quadruple of u, v, w, and s

An expression[1] is either a variable (u), a data constructor (C), a function abstraction ($\lambda\mathrm{u}\,.\,\mathrm{e}$), a function application (e e), an expression under a local definition (`let` u=e `in` e), or a scrutinee with a set of pattern-matching branches (`case` e `of` { $\overline{\mathrm{T}\,\overline{\mathrm{u}} \rightarrow \mathrm{e}}$ }). When I must refer to a specific program expression, I adopt the syntactic conventions of the Haskell programming language (*i.e.,* variable names and type variable names begin with a lower-case letter, and data constructor and

[1] In a functional programming language, a program is nothing more than an expression, thus the syntax of programs is identical to the syntax of expressions.

type constructor names begin with an upper-case letter) [14] and set the entire expression, including variables and data constructors, in `typewriter font`.

Types and programs are closely related constructs, and the standard way to represent their relation is to use *type judgments*, which have the form $\Gamma \vdash \mathrm{e} : t$ (read "under the type environment Γ, the expression e has type t").[2] Different type systems include different type judgments, so the same expression under the same type environment may have different types in different type systems, or it may have a type in one type system but no type in another.

In the next two sections, I introduce two type systems that serve as the foundation of my dissertation research.

2.2 ALGEBRAIC DATA TYPES

Algebraic Data Types (ADTs) are a programming language and type system feature that serves as the core of many functional languages such as Haskell [14] and ML [28]. An algebraic data type[3] allows a program to organize and to process data using custom-defined data constructors. The following example[4] illustrates how a programmer can define an algebraic data type:

```
data Tree a where
  Tip  :: forall a. a → Tree a
  Fork :: forall a. Tree a → Tree a → Tree a
```

[2] Following convention, a type judgment uses a single colon (:) to separate an expression from its type. Program examples in this dissertation, which follows the syntax of the Haskell programming language, use double colons (`::`) for the same role. The difference is due to history and convention, and it has no significance otherwise.

[3] By convention, the term *algebraic data type* may refer to either a specific data type defined by a programmer (such as `Tree`), or the type system that I will introduce later in the section. The specific meaning is usually clear from the context.

[4] The program syntax in Figure 2.3 (p. 20) does not include the declaration of algebraic data constructors, and the technical development in this dissertation assumes that data constructors are defined a priori. When I need to discuss specifics in examples, I use the syntax proposed by Peyton Jones et al. [17] and adopted in the Glasgow Haskell Compiler [44, §7.5].

The `Tree` type consists of two data constructors (`Tip` and `Fork`) that can be used in conjunction to define binary trees. Each data constructor has a type, and each data constructor type consists of two parts. The part of a data constructor type after the right-most arrow is the *range type* of the constructor; it describes the type of the value that the data constructor constructs. `Tip` and `Fork` both have range types `Tree a` because they are both data constructors of the `Tree` type. The part of a data constructor type before the right-most arrow comprises the *argument types*; they describe the internal data values that each data constructor encapsulates.

2.2.1 Constructing algebraic data

Programmers can use data constructors to construct algebraic data values. For example, when applied to a value of type `a`, `Tip` constructs a `Tree a` value that contains the `a` value. The constructed value represents a leaf node of a binary tree (which contains a data element but no sub-trees). When applied to two values of type `Tree a`, `Fork` constructs a new `Tree a` value that contains two `Tree a` values. The constructed value represents an internal node of a binary tree (which connects two sub-trees).

Algebraic data types support type parametrization, which allows programmers to instantiate a data constructor type differently to accommodate different uses. For example, the `Tree` type constructor has one type argument, which represents the type of the data elements in the leaf nodes of a tree. By instantiating the type argument to different types, programmers can define many different types of trees using the same `Tip` and `Fork` data constructors. The following examples[5] illustrate type parametrization:

```
intTree1 :: Tree Int
```

[5] In the interest of readability, I use the concrete syntax of the Haskell programming language for the program examples in this dissertation. See the Haskell language report [14] for details.

```
intTree1 = Tip 5
intTree2 :: Tree Int
intTree2 = Fork (Tip 3) (Fork (Tip 9) (Tip 2))
boolTree :: Tree Bool
boolTree = Fork (Fork (Tip True) (Tip True)) (Tip False)
```

In the examples, the lines that contain a double colon (`::`) are *type annotations*; each annotation specifies the type of an upcoming expression definition. For example, the first line specifies that an upcoming definition will bind `intTree1` to an expression with type `Tree Int`. In this dissertation, I make extensive use of type annotations to document the types of expressions.

An algebraic data type may be parametrized by zero, one, or multiple types. To support type parametrization, all data constructor range type arguments (*i.e.,* all type arguments of the range type of a data constructor) in an algebraic data type declaration must be distinct type variables. The range type of a data constructor is *uniform* if it satisfies this requirement. For example, the following data constructors have uniform range types:

```
C1 :: forall a. a → T1 a
C2 :: forall a b. Int → T2 a b
C3 :: forall a b c. Bool → b → T3 a b c
```

The following data constructors have non-uniform range types:

```
C4 :: forall a. a → T4 [a]
C5 :: forall a. Int → T5 a a
C6 :: forall a b. Bool → b → T6 a b Bool
```

Using this new terminology, all data constructors in an algebraic data type must have uniform range types.

2.2.2 Destructing algebraic data

Data constructors serve a dual purpose: programmers can (as previously discussed) use them to *construct* data values, and programmers can use them to *destruct* data values. Here destruct does not mean destroy; rather it refers to the language feature that a program uses to analyze the structure of an algebraic data value. Typically this mechanism is provided by pattern-matching branches in a `case` expression. Pattern-matching branches provide two kinds of information: the data constructors to match, and the expression to evaluate for each possible match. Here is an example:

```
sumTree :: Tree Int → Int
sumTree t = case t of
  Tip i → i
  Fork l r → sumTree l + sumTree r
```

The `sumTree` function computes the sum of the data elements in a binary tree of integers. The body of the function is a `case` expression, which allows the function to return different values based on the structure (`Tip i` or `Fork l r`) of the binary tree. Each `case` expression consists of two parts: the *scrutinee* (`t`), which specifies the value to be analyzed, and the *pattern-matching branches* (the two lines after the keyword `of`).

A pattern-matching branch also consists of two parts. The part to the left of the arrow (*e.g.,* `Fork l r`) is the *pattern* of the branch; it describes a possible top-level structure of the scrutinee. The part to the right of the arrow (*e.g.,* `sumTree l + sumTree r`) is the *branch body*; it specifies the value to return when the structure of the scrutinee matches the pattern. The branch body accesses the internal data value in the scrutinee (carried by the data constructor) through the variable names listed in the pattern (`l` becomes the left sub-tree and `r` becomes the right sub-tree). Going back to the example, the `case` expression in the body of `sumTree` says that the function returns the integer data value in leaf nodes, and for internal nodes it returns the sum of the results from the sub-trees. The language I study in this dissertation (Figure 2.3, p. 20) does not support nested patterns, so every pattern must be a data constructor (`Fork`) followed by a sequence of distinct variable names (`l r`) as its arguments.

2.2.3 Type system

In addition to the language features that programmers can use directly, algebraic data types also come with a type system to ensure the internal coherence of programs. The ADT type system is not really a single type system, but rather a family of type systems with slightly different feature sets. For this dissertation I base my technical development on what is essentially the Milner type system [27] with algebraic data types, existential types [22], and polymorphic recursion [29].

Each type system consists of two parts. The first part (Figure 2.3, p. 20) specifies the structural properties of types in the type system. The ADT type system supports universal quantification over type variables — types with quantifiers are *polymorphic*, and types without quantification are *monomorphic*. The ADT type system requires that each data constructor should have a polymorphic type that universally quantifies all its type variables. If a data constructor type contains no type variables, this requirement is trivially satisfied without any quantification. As I discussed earlier, the ADT type system also requires data constructor range types to be uniform. In other words, the range type of a data constructor must be a type constructor with distinct type variables as its arguments.

The second part of a type system consists of a set of *type rules*, each of which defines a way to construct a valid type judgment for expressions in the language. Each type rule consists of two parts separated by a horizontal line: the part above the line states the requirements for the type judgment below the line. Figure 2.4 (p. 27) shows the type rules in the ADT type system, and I briefly describe each

VAR

$$\frac{\mathrm{x} : \forall\overline{\alpha}.\, t \in \Gamma \qquad s = \mathrm{inst}[\overline{\alpha}](t)}{\Gamma \vdash \mathrm{x} : s}$$

LAM

$$\frac{\Gamma\{\mathrm{u} : s\} \vdash \mathrm{e} : t}{\Gamma \vdash \lambda \mathrm{u}\,.\,\mathrm{e} : s \to t}$$

CONS

$$\frac{\mathrm{C} : \forall\overline{\alpha}.\, t \qquad s = \mathrm{inst}[\overline{\alpha}](t)}{\Gamma \vdash \mathrm{C} : s}$$

APP

$$\frac{\Gamma \vdash \mathrm{f} : t_1 \to t_2 \qquad \Gamma \vdash \mathrm{e} : t_1}{\Gamma \vdash \mathrm{f}\ \mathrm{e} : t_2}$$

LETREC-P

$$\frac{\Gamma\{\mathrm{u} : \forall\overline{\alpha}.\, s\} \vdash \mathrm{e} : s \qquad \overline{\alpha} \mathbin{\#} \mathit{tyvar}(\Gamma) \qquad \Gamma\{\mathrm{u} : \forall\overline{\alpha}.\, s\} \vdash \mathrm{d} : t}{\Gamma \vdash \texttt{let}\ \mathrm{u} = \mathrm{e}\ \texttt{in}\ \mathrm{d} : t}$$

CASE

$$\frac{\Gamma \vdash \mathrm{e} : s \qquad \Gamma \vdash_p \mathrm{p}_i \to \mathrm{c}_i : s \to t}{\Gamma \vdash \texttt{case}\ \mathrm{e}\ \texttt{of}\ \{\, \overline{\mathrm{p}_i \to \mathrm{c}_i}\,\} : t}$$

ALT-ADT

$$\frac{\begin{array}{c} \mathrm{C} : \forall\overline{\alpha}.\, \overline{w} \to T\ \overline{\gamma} \qquad \overline{\alpha} \mathbin{\#} \mathit{tyvar}(\Gamma, \overline{u}, t) \\ \theta = [\overline{u/\gamma}] \qquad \Gamma\{\overline{\mathrm{x} : \theta(w)}\} \vdash \mathrm{c} : t \end{array}}{\Gamma \vdash_p \mathrm{C}\ \overline{\mathrm{x}} \to \mathrm{c} : T\ \overline{u} \to t}$$

$\mathrm{inst}[\overline{\alpha}](t) = \theta(t)$, where $\theta = [\overline{s/\alpha}]$, and $\overline{s}$ are arbitrary types

Figure 2.4: The ADT type system.

rule in the type system:

VAR This rule looks up the type (t) of the variable (x) in the type environment (Γ), which is a set that maps variable names to polymorphic types. The rule also instantiates the universally-quantified type variables ($\overline{\alpha}$) in the type (t).

CONS This rule looks up the type (t) of a data constructor (C) in an implicit global type environment, and it instantiates the universally-quantified type variables ($\overline{\alpha}$) in the type (t).

LAM This rule types a function by requiring that the function body (e) has the body type (t) of the function under the extended type environment that maps the function argument (u) to the argument type (s) of the function.

APP This rule types a function application by typing the function (f) and its argument (e) separately and requiring that the function must have a function type. The domain of the function must have the same type (t_1) as the argument, and the range of the function must have the same type (t_2) as the entire application.

LETREC-P This rule types a `let` expression by checking that both the local definition (e) and the body of the expression (d) have the appropriate types (s and t) under the extended type environment that maps the locally-defined variable (u) to a polymorphic type ($\forall \overline{\alpha}.\, s$). Note that the `let` construct supports recursive definitions: the body of the local definition (e) can refer to the locally-defined variable (u).

CASE This rule types a `case` expression by typing the scrutinee (e) and the pattern-matching branches ($\mathrm{p}_i \to \mathrm{c}_i$) separately, and then requiring that all branch types must be an arrow from the scrutinee type (s) to the `case` expression type (t). This rule uses a special judgment for pattern-matching branches, which I describe next.

ALT-ADT This rule types a pattern-matching branch. Since a pattern-matching branch is not an expression, I use $\vdash_p$ (instead of $\vdash$) in the type judgment for pattern-matching branches to make the distinction clear.

Since a pattern destructs a value, this rule uses the data constructor type backwards: the range type ($T\ \overline{\gamma}$) of the constructor (C) becomes the type of the pattern (C $\overline{\mathrm{x}}$), which instantiates to the scrutinee type ($T\ \overline{u}$). The rule then uses the instantiation substitution (θ) to refine the constructor's argument types ($\overline{w}$), which now become the types of the pattern-bound variables ($\overline{\mathrm{x}}$). Finally, the rule types the branch body (c) under the type environment extended with the types of the pattern-bound variables.

From the perspective of a type checking algorithm, the type judgment below the horizontal line in each type rule — which includes the type environment, the program expression, and the expected type — is given as input, and the algorithm is responsible for filling in the remaining information to meet the requirements stated above the horizontal line. For example, in the APP type rule, only the type t_2 is given as input, and a type checking algorithm must find a type t_1 (if it exists) that allows successful type checking for both the function and its argument. This responsibility on type checking algorithms is a problem in the LETREC-P rule — Henglein [12] and Kfoury et al. [21] proved that type inference with polymorphic recursion (so named because the recursive binding u in e has a polymorphic type) is computationally undecidable. One common work-around for this problem is to replace LETREC-P with the two type rules in Figure 2.5 (p. 30).

The LETREC-M rule, which allows only monomorphic recursion (the recursive binding u in e is restricted to a monomorphic type), admits a complete type inference algorithm [8, 27]. The LETREC-A rule, which allows polymorphic recursion, requires programmer type annotation (by slightly extending the syntax for `let` expressions) so that a type checking algorithm is no longer required to infer the

LETREC-M

$$\frac{\Gamma\{\mathrm{u} : s\} \vdash \mathrm{e} : s \qquad \overline{\alpha} = \mathit{tyvar}(s) \setminus \mathit{tyvar}(\Gamma) \qquad \Gamma\{\mathrm{u} : \forall\overline{\alpha}.\, s\} \vdash \mathrm{d} : t}{\Gamma \vdash \mathtt{let}\ \mathrm{u}\!=\!\mathrm{e}\ \mathtt{in}\ \mathrm{d} : t}$$

LETREC-A

$$\frac{\Gamma\{\mathrm{u} : \forall\overline{\alpha}.\, s\} \vdash \mathrm{e} : s \qquad \overline{\alpha} \,\#\, \mathit{tyvar}(\Gamma) \qquad \Gamma\{\mathrm{u} : \forall\overline{\alpha}.\, s\} \vdash \mathrm{d} : t}{\Gamma \vdash \mathtt{let}\ \mathrm{u}\!=\!\mathrm{e} : \forall\overline{\alpha}.\, s\ \mathtt{in}\ \mathrm{d} : t}$$

Figure 2.5: Type rules for annotated polymorphic recursion.

type of the local definition.

I choose not to employ this workaround. This choice implies that any type checking or type inference algorithm for the ADT type system must necessarily be incomplete (*i.e.,* it must fail to accept some well-typed programs), but at the same time it allows for the possibility that a type inference algorithm may support some use of polymorphic recursion without programmer type annotations [12, 29, 48]. I will say more about type inference with polymorphic recursion in Chapter 6.

2.3 GENERALIZED ALGEBRAIC DATA TYPES

Generalized algebraic data types (GADTs) [17] are an extension to the ADT type system that support non-uniform data constructor range types. Researchers have found GADTs useful in a wide variety of programming tasks: generalized tries [6], AVL trees [38, §4.1], generic programming [6], arrow [30] and monad [23] libraries, parsing combinators [1, 23], and tagless language interpreters [1, 6, 39]. The Glasgow Haskell Compiler has supported GADTs since version 6.4 in 2005 [44, §7.5], and GADTs have found their way into major software projects such as Pugs [47], a leading Perl 6 implementation, and Darcs [46], a distributed revision control system with advanced merge features.

2.3.1 Previous work

Generalized algebraic data types, in their current form as a *programming language* feature, can be traced back to a 1994 manuscript by Augustsson and Petersson [1]. In the manuscript, they investigate an extension to algebraic data types that allows data constructors with non-uniform range types (*i.e.,* data constructors whose range type arguments are not distinct type variables), and they propose that the type rule for `case` expressions should combine scrutinee and pattern types with unification [35].

Eight years later, in 2002, researchers again turned their attention to the problem of defining data constructors with non-uniform range types. Two papers, one by Baars and Swierstra [2] and the other by Cheney and Hinze [5], independently demonstrated the technique of using Leibniz' Law (the identity of indiscernibles) [10] to restrict the range types of data constructors in Haskell. This technique, inspired by Weirich [50], encodes Leibniz' Law into Haskell types as follows:

```
data Equal a b where
  Leibniz :: forall a b. (forall f. f a → f b) → Equal a b
```

Note that the type variable `f` represents not a type, but a type constructor of arity one (or, more precisely, of kind `*` → `*`). Since `f` is universally quantified, the only non-diverging function that can be an argument of `Leibniz` is the identity function, so the types `a` and `b` must be equal. With the `Equal` type, programmers can now define data constructors with simulated non-uniform range types:

```
data Rep a where
  RepInt  :: Equal a Int → Rep a
  RepBool :: Equal a Bool → Rep a
```

For the `RepInt` data constructor, its range type argument `a` must be equal to `Int` because all values of type `Equal` x y that are constructed from non-diverging functions require $x = y$. Likewise, for the `RepBool` data constructor, its range type argument `a` must be equal to `Bool`. In a data type with (simulated) non-uniform constructor range types such as `Rep`, the type of an algebraic data value varies with its structure. In other words, the structure of an algebraic data value becomes connected to its type. This connection between the structure of values and types allows programmers to harvest some of the power of dependent type systems in the familiar setting of algebraic data types.

Once researchers recognized the usefulness of non-uniform constructor range types, they worked to extend the ADT type system so that programmers no longer had to manually invoke Leibniz equality. Two proposals, First-Class Phantom Types by Cheney and Hinze [6] and Equality-Qualified Types by Sheard and Pasalic [39], add type equations to data constructor declarations and extend the type system to support those type equations. These two proposals are directly comparable to the technique based on Leibniz' Law, except that type equations are more pleasant to use and incur no runtime overhead. Guarded Recursive Datatype Constructors by Xi et al. [52] do away with type equations in the surface language and instead allow programmers to specify non-uniform range types for data constructors directly. Their type system, however, still relies on type equations internally to check pattern-matching branches. All three proposals for extending the ADT type system to support non-uniform constructor range types appear to be equivalent [6, §6.6] [39, §6]. Compared to Intensional Polymorphism by Crary et al. [7], which proposes a type system with a built-in singleton type, these proposals provide a similar feature set, but with the additional benefit that programmers can define new data types that suit their own needs.

Peyton Jones et al. [17, 18] are the first to use the term "Generalized Algebraic Data Types" and the data type declaration syntax subsequently implemented in the Glasgow Haskell Compiler. In their type system, programmers specify nonuniform range types for data constructors directly without resorting to type equations, and the type system checks pattern-matching branches using unification without resorting to type equations. This type-equation-free formulation marks a significant departure from previous work, but again this proposal by Peyton Jones et al. appears to be equivalent to all previous proposals. This result should perhaps not be surprising: type equations still exist conceptually in this type system; only that they are now solved on-the-fly with unification and thus appear not as equations, but as substitutions.

2.3.2 Features

My formulation of a GADT type system closely follows the GADT type system of Peyton Jones et al. [17]. Since I will be introducing two new variations of the GADT type system later in the dissertation, I will refer to the system that I am now introducing as the *plain* GADT type system when there is a possibility of confusion. In the rest of this section, I will demonstrate the plain GADT type system through examples and then present its formal definition.

The GADT type system adds two new features to the ADT type system:

1. First, the GADT type system lifts the restriction that data constructor range type arguments must be distinct type variables. In other words, the range type arguments of a GADT data constructor can contain type constructors or duplicated type variables or both. I call this feature *GADT type arguments*. The `Term` data type in Figure 2.6 (p. 34) demonstrates this feature: the range type arguments of data constructors vary from one constructor to another, and the type argument `a` of an expression with type `Term a` represents the type of the object-level term encoded by the expression. Here are three examples:

```
data Term a where
  RepInt  :: Int → Term Int
  RepBool :: Bool → Term Bool
  RepCond :: forall a. Term Bool → Term a → Term a → Term a
  RepPair :: forall a b. Term a → Term b → Term (a, b)

eval :: forall a. Term a → a
eval e = case e of
  RepInt i → i
  RepBool b → b
  RepCond b u v → if eval b then eval u else eval v
  RepPair u v → (eval u, eval v)
```

Figure 2.6: This figure demonstrates a tagless expression evaluator in the GADT type system. This example is similar to the one I presented in Figure 1.4 (p. 11) but with a slightly different expression language. I assume that the programming language supports `if` conditional expressions with the usual type and semantics. The data constructors `RepInt`, `RepBool`, and `RepPair` of the `Term` data type have non-uniform range types: the GADT type argument for each data constructor represents the type of the object-level term built from the constructor. The evaluation function `eval` uses GADT type refinements to avoid tagging (*i.e.,* encapsulating the result of evaluation in another data type) and the associated runtime overhead. This is one of the prime motivating examples for the GADT type system.

```
term1 :: Term Int
term1 = RepInt 3
term2 :: Term (Bool, Int)
term2 = RepPair (RepBool True) (RepInt 5)
-- badTerm is not well-typed
badTerm = RepCond (RepBool True) (RepInt 3) (RepBool False)
```

The expression `badTerm` is not well-typed because its true branch has type `Term Int` but its false branch has type `Term Bool`, which violates one of the requirements encoded in the type of `RepCond` (*i.e.,* the object-level terms in the two branches of a conditional must have the same type). This example demonstrates that GADT type arguments allow the type checker to help programmers enforce structural constraints on algebraic data values.

2. Let us move on to the second new feature that the GADT type system adds to the ADT type system. In a `case` branch that matches a GADT data constructor, the *pattern type* of the branch (which is the range type of the data constructor) can induce a type substitution that brings additional type information into scope. We call the substitution a *GADT type refinement.* GADT type refinements allow the branch body type and the types in the environment to vary from one branch to another. The `eval` function in Figure 2.6 (p. 34) illustrates this feature: the bodies of the `RepInt` and `RepBool` branches have different types (`Int` and `Bool`, respectively). From the type annotation, the `case` scrutinee `e` has type `Term a`, and the branch bodies have type `a`. However, since the pattern `RepInt i` has type `Term Int` (from the range type of `RepInt`), it induces a type refinement [`Int`/`a`] (by unifying `Term a` with `Term Int`), and thus we can use an integer `i` as the body of this particular branch (and similarly, the Boolean value `b` as the body of the `RepBool` branch).

I want to emphasize one particularly subtle consequence of the GADT type refinement feature. While this feature allows types to vary between different GADT pattern-matching branches,[6] not all inter-branch type variations are acceptable to the GADT type system. In other words, GADT type refinement weakens, but does not completely eliminate, restrictions on the types of GADT pattern-matching branches. One simple way to state the weakened restriction is that branch types can vary only in accordance with the GADT type arguments of the branch pattern types; this (weakened) restriction represents a basic type-coherence requirement between GADT pattern-matching branches. The `eval` function, as I have shown, obeys this restriction. The following function `cross`, in contrast, disobeys this restriction, so it is not well-typed in the GADT type system:

```
cross e = case e of
  RepInt i → True      -- GADT type argument: Int, body type: Bool
  RepBool b → 7        -- GADT type argument: Bool, body type: Int
```

Researchers have also studied type systems that further weaken the restriction on GADT pattern-matching branches. Guarded Algebraic Data Types (note the difference between *Guarded* and *Generalized*) by Simonet and Pottier [40] are a general framework that allows programming language designers to attach type constraints to data constructor types. Such constraints may include, but are not limited to, type equations. In the specific case where the type constraints are type equations, this work bears some resemblance to first-class phantom types [6]. In the context of type inference, many researchers compare this work directly with various formulations of the GADT type system [19, §6] [36, §9.2] [39, §7] [40, §1.5.2] [43, §1], which underlines the belief that this work (with type equations as constraints) is yet another formulation of the GADT type system.

[6] A GADT pattern-matching branch is a pattern-matching branch whose pattern refers to a GADT data constructor (*i.e.,* a data constructor with non-uniform range type arguments).

This belief is, however, misplaced. Guarded algebraic data types, even with type equations as constraints, are fundamentally different from the GADT type system. The plain GADT type system treats GADT type refinement as local type assumptions; guarded algebraic data types treat the same type equations as antecedent of a logical implication. This liberal treatment allows the `cross` function to have the following type in the guarded algebraic data type system:

```
forall a b. (a=Int ⊃ b=Bool) ∧ (a=Bool ⊃ b=Int) ⇒ a → b
```

The ⊃ symbol represents logical implication (*e.g.,* $A \supset B$ means "A implies B"), and the ⇒ symbol serves only as a separator between type implication constraints and the type of the function. Guarded algebraic data types accept more programs than the plain GADT type system (`cross` is one example), but this expressiveness comes at a cost: types become more complicated, and (more importantly) type checking no longer enforces basic type-coherence between pattern-matching branches. A programming error that triggers a type error in the plain GADT type system may, in guarded algebraic data types, leave the program well-typed but with a much more complicated type. This permissiveness makes guarded algebraic data types less useful from a software engineering perspective, and in the rest of this dissertation I will consider only type systems that enforce type-coherence between pattern-matching branches.

2.3.3 Type system

Figure 2.7 (p. 38) shows the definition of the GADT type system. Compared to the ADT type system in Figure 2.4 (p. 27), the only difference is that the ALT-ADT rule in the ADT type system is replaced by the ALT-GADT rule. For the rest of this section I will explain the design of the ALT-GADT type rule, which specifies type checking for GADT pattern-matching branches.

VAR

$$\frac{\mathrm{x} : \forall\overline{\alpha}.\, t \in \Gamma \qquad s = \mathrm{inst}[\overline{\alpha}](t)}{\Gamma \vdash \mathrm{x} : s}$$

LAM

$$\frac{\Gamma\{\mathrm{u} : s\} \vdash \mathrm{e} : t}{\Gamma \vdash \lambda\mathrm{u}\,.\,\mathrm{e} : s \to t}$$

CONS

$$\frac{\mathrm{C} : \forall\overline{\alpha}.\, t \qquad s = \mathrm{inst}[\overline{\alpha}](t)}{\Gamma \vdash \mathrm{C} : s}$$

APP

$$\frac{\Gamma \vdash \mathrm{f} : t_1 \to t_2 \qquad \Gamma \vdash \mathrm{e} : t_1}{\Gamma \vdash \mathrm{f}\ \mathrm{e} : t_2}$$

LETREC-P

$$\frac{\Gamma\{\mathrm{u} : \forall\overline{\alpha}.\, s\} \vdash \mathrm{e} : s \qquad \overline{\alpha} \# \mathit{tyvar}(\Gamma) \qquad \Gamma\{\mathrm{u} : \forall\overline{\alpha}.\, s\} \vdash \mathrm{d} : t}{\Gamma \vdash \texttt{let}\ \mathrm{u} = \mathrm{e}\ \texttt{in}\ \mathrm{d} : t}$$

CASE

$$\frac{\Gamma \vdash \mathrm{e} : s \qquad \Gamma \vdash_p \mathrm{p}_i \to \mathrm{c}_i : s \to t}{\Gamma \vdash \texttt{case}\ \mathrm{e}\ \texttt{of}\ \{\, \overline{\mathrm{p}_i \to \mathrm{c}_i}\,\} : t}$$

ALT-GADT

$$\frac{\begin{array}{cc} \mathrm{C} : \forall\overline{\alpha}.\, \overline{w} \to T\ \overline{s} & \overline{\alpha} \# \mathit{tyvar}(\Gamma, \overline{u}, t) \\ \theta = \mathit{mgu}(T\ \overline{u} \sim T\ \overline{s}) & \theta(\Gamma\{\overline{\mathrm{x} : w}\}) \vdash \mathrm{c} : \theta(t) \end{array}}{\Gamma \vdash_p \mathrm{C}\ \overline{\mathrm{x}} \to \mathrm{c} : T\ \overline{u} \to t}$$

$\mathrm{inst}[\overline{\alpha}](t) = \theta(t),$ where $\theta = [\overline{s/\alpha}],$ and $\overline{s}$ are arbitrary types

Figure 2.7: The plain GADT type system.

I mentioned earlier that the GADT type system adds two new features to the ADT type system, and each of these new features requires changing the type rule for pattern matching branches. The first feature, GADT type arguments, requires changing the ALT-ADT rule for two reasons:

1. The ALT-ADT rule propagates type information only from the scrutinee type to the pattern type, but GADT type arguments may contain type information that needs to be propagated the other way around, and

2. The ALT-ADT rule assumes that all pattern types in the `case` expression must have a common instance, but support for non-uniform constructor range types in the GADT type system violates this assumption. For example, in the `eval` function, the `RepInt` pattern has type `Term Int`, and the `RepBool` pattern has type `Term Bool`. These two types are not unifiable and thus have no common instance.

For these reasons, the ALT-GADT rule does not require that the scrutinee type ($T\ \overline{u}$) be an instance of the pattern type ($T\ \overline{s}$). Instead, it requires only that the scrutinee type be unifiable with the pattern type.

The second feature of the GADT type system, GADT type refinements, also requires changing the ALT-ADT rule so that branch body type and types in the environment may vary with the pattern type of each branch. To support this feature, the ALT-GADT rule applies the most-general unifier of the scrutinee and the pattern types (θ), not only to the types of the pattern-bound variables ($\overline{w}$), but also to the type environment (Γ) and to the type of the branch body (t).

The plain GADT type system is essentially the GADT type system of Peyton Jones et al. [17] but without nested patterns and type annotations. The plain GADT type system accepts only a subset of the programs that are well-typed in the GADT type system of Peyton Jones et al., so its soundness follows from the soundness of the GADT type system of Peyton Jones et al. [15, §3].

2.4 INFERENCE FOR THE GADT TYPE SYSTEM

Ever since the conception of the GADT type system (and its equivalents), the type inference problem for the GADT type system has been an area of active research [17, 32, 36, 42, 43]. To this day, however, the GADT type inference problem remains open, and leading language implementations such as the Glasgow Haskell Compiler require programmer type annotations for virtually all definitions that contain GADT patterns. In this section, I summarize the current state of the art in this research area.

2.4.1 Technical challenges

What makes the GADT type inference problem difficult? Previous work in this area points to three technical challenges:

1. The GADT type system lacks the principal type property. The *principal type property*, sometimes also referred to as principal types, is a type system property that every well-typed expression has a most-general type, which is called the *principal type* of the expression [8]. This property is important for type inference because it states that a type inference algorithm need only infer one type for each well-typed expression, and that single type can represent all other types of the expression.

   The GADT type system lacks this useful property [6, 17, 42, 43]. Figure 2.8 (p. 41) shows the `repId` function adopted from Sulzmann et al. [43, §2.1]. The `repId` function has infinitely many valid types, none of which is more general than any other. Since there is no reason to favor any one of these types over all others, a complete type inference algorithm must consider all of these possible types, which adds significant complication.

2. GADT type refinements allow pattern-matching branch bodies in a `case`

```
repId :: forall a. Term a → [a] → [Int]
repId :: forall a. Term a → [[a]] → [[Int]]
repId :: forall a. Term a → [[[a]]] → [[[Int]]]
             ⋮
repId e x = case e of
  RepInt i → x
  RepBool b → []     -- Branch body is an empty list
```

Figure 2.8: This figure demonstrates a plain GADT program that does not have a principal type. The `repId` function, which uses the `Term` data type I introduced earlier (Figure 2.6, p. 34), has an infinite number of maximal types (*i.e.,* types that are not an instance of a more general type). I adopted this example from Sulzmann et al. [43, §2.1], who first demonstrated that a plain GADT program may have an infinite number of maximal types.

expression to have different types [32, 36, 42, 43]. This feature makes type inference difficult because, in the GADT type system, there is no unique mapping from the body type of one branch to the body type of another, or from a branch body type to the type of the enclosing `case` expression. A type inference algorithm for the ADT type system can compute the type of a `case` expression by unifying the types of its branch bodies; this simple strategy does not work in the GADT type system. For example, in the `eval` function (Figure 2.6, p. 34), the `RepInt` branch has body type `Int`, and the `RepBool` branch has body type `Bool`. These two branch body types are not unifiable, but `eval` remains well-typed in the GADT type system.

3. Many GADT programs use polymorphic recursion [6, 17, 32, 42, 43]. Polymorphic recursion [29], which allows a recursive definition to invoke itself on an instance of its own type, is not a feature specific to the GADT type system. For example, I also included this feature in my ADT type system (§2.2). While polymorphic recursion is relatively rare in ADT programs, it appears to be quite common in recursive GADT programs — since GADT type arguments can reflect the structure of a value, even standard structural recursion over a generalized algebraic data type may require polymorphic recursion. For example, in the `eval` function, the first recursive call (`eval b`) in the `RepCond` branch applies `eval` to an expression of type `Term Bool`, which disagrees with the refined type (`Term a`) of the argument (`e`) in the definition of `eval`. Therefore `eval` is well-typed only in a type system that supports polymorphic recursion.

   Unfortunately, Henglein [12] and Kfoury et al. [21] proved independently, by reduction from semi-unification [20], that type inference with polymorphic recursion is computationally undecidable. Since many GADT programs rely on polymorphic recursion, type inference for the GADT type system might also be expected to be undecidable.

Even in the absence of polymorphic recursion, the decidability of the GADT type inference problem remains open.

2.4.2 Computational decidability

There have been at least three claims pertaining to the decidability of the GADT type inference problem. Simonet and Pottier [40, §4.2] claim that, with type equations as constraints, type inference for guarded algebraic data types reduces to a decidable constraint-solving problem, while Sulzmann et al. [43, §2.1] and Schrijvers et al. [36, §4.3] claim that GADT type inference is computationally undecidable. None of these claims settles the problem conclusively.

The claim by Simonet and Pottier has no bearing on the decidability of GADT type inference because, contrary to popular belief, the GADT type system is not equivalent to guarded algebraic data types with type equations as constraints. The argument used by Sulzmann et al. assumes, without justification, that a complete GADT type inference algorithm must enumerate all (possibly infinitely many) maximal types of an expression. The argument used by Schrijvers et al. is built on the equivalence of the following two problems:

- Simultaneous rigid E-unification, which is undecidable [9], and
- Type implication constraints from GADT pattern-matching branches.

This proposed equivalence is, however, invalid. There are at least two main differences between these two problems:

1. The GADT type system requires that the antecedent of any type implication generated from a pattern-matching branch must be consistent [36, §3.2]. In my terminology, the scrutinee type and the pattern type must be unifiable. The rigid E-unification problem, in contrast, places no such requirement.

2. In a GADT type system without the antecedent consistency requirement, a type implication with an inconsistent antecedent represents an unreachable pattern-matching branch and therefore has a trivial solution. In the rigid E-unification problem, a set of equations is interpreted as its congruence closure, which is nontrivial even for "inconsistent" equations.

These differences between the two classes of problems are clearly illustrated with this rigid E-unification example from Gallier et al. [11, Example 1.1]:

> Let $E = \{f(a) = a,\, g(g(x)) = f(a)\}$. The rigid E-unification problem of unifying $g(g(g(x))) \sim x$ under E has a solution $\theta = [g(a)/x]$. Since this derivation holds under $\theta(E) = \{f(a) = a,\, g(g(g(a))) = f(a)\}$:
>
> $$\theta(g(g(g(x)))) = g(g(g(g(a)))) = g(f(a)) = g(a) = \theta(x)$$
>
> The substitution θ is a solution of the E-unification problem. ❊

A GADT type implication constraint solver would not accept the inconsistent equations E as an antecedent. Even if it did, it would accept any substitution θ as solution, instead of requiring $\theta(g(g(g(x))))$ and $\theta(x)$ to be congruent modulo $\theta(E)$. This example demonstrates that type implication constraints from GADT pattern-matching branches are not equivalent to simultaneous rigid E-unification, thus the argument used by Schrijvers et al. is also invalid.

With this conclusion that the computational decidability of GADT type inference remains open, I now move on to discuss the designs of existing GADT type inference algorithms.

2.4.3 Previous work

The designs of existing GADT type inference algorithms mostly revolve around the three technical difficulties that I discussed earlier in this section: the lack of principal types, GADT type refinements, and polymorphic recursion. There is a solution that works for all three difficulties: *ask the programmer*. In other words, let the type inference algorithm rely on programmer type annotations in difficult spots. Though conceptually simple, this solution is not at all easy to implement: the algorithm designers must identify where type annotations are necessary and figure out how to propagate type information from the annotations (typically at `let` bindings) to the places where they are needed.

Wobbly types The wobbly types algorithm by Peyton Jones et al. [17] is representative of this design. This algorithm, which is based on Milner's Algorithm $\mathcal{W}$ [27], attaches a label to each type: a type is *rigid* if it can be traced back to a programmer type annotation; otherwise the type is *wobbly*. The algorithm operates in one of two modes depending on the label: it checks rigid types and infers wobbly types. GADT type refinements apply only to rigid (*i.e.,* programmer annotated) types, so there is no ambiguity about whether a type should be refined. The algorithm is sound, and it is also complete for suitably annotated programs.

Stratified type inference This work by Pottier and Régis-Gianas [32] improves the modularity of the wobbly types algorithm by separating it into two strata: the top stratum is a source-to-source transformation (called *shape inference*) that adds type annotations by propagating existing annotations, and the bottom stratum is a simple type inference algorithm that generates and solves type constraints from program expressions. This separation makes it easier to change the way type annotations propagate in a program, and the authors take advantage of this feature by proposing and comparing two shape inference algorithms.

OutsideIn The OutsideIn algorithm by Schrijvers et al. [36] functions by generating and solving type constraints from program expressions. The name reflects the design principle that the algorithm can propagate type information from the context into GADT pattern-matching branches, but not the other way around.

A good way to understand the OutsideIn algorithm is to consider it (conceptually) as having two strata, similar to how Pottier and Régis-Gianas described their type inference algorithm. The top stratum of OutsideIn propagates type information aggressively by inferring the type of the entire program except for the GADT pattern-matching branches, and the bottom stratum of OutsideIn uses the inferred context type information to check the types of GADT pattern-matching branches.

The top stratum of OutsideIn derives type information not only from type annotations but also through type inference. In principle, this approach can reduce the amount of mandatory type annotation by deriving the same type information through type inference. Here is an example from the paper [36, §5.3]:

```
data T a where
  T1 :: Int → T Bool
  T2 :: forall a. [a] → T a

outIn e = case e of
  T1 n  → n > 0
  T2 xs → null xs
```

From the `T2` branch (which is not a GADT pattern-matching branch because the data constructor `T2` has a uniform range type), the top stratum of OutsideIn infers `forall a. T a → Bool` as the type of `outIn`. The bottom stratum then uses this type to check the `T1` branch. Even though `outIn` contains a GADT pattern, the OutsideIn algorithm accepts it without a type annotation because the top stratum can derive the same type information from the `T2` branch. (Neither wobbly types nor stratified type inference can accept `outIn` without a type annotation.)

In practice, however, I believe that the benefits of using the OutsideIn strategy to allow fewer type annotations are negligible because:

1. Many GADT functions consist of nothing but a collection of GADT pattern-matching branches that match the function arguments. The `eval` function (Figure 2.6, p. 34) is one such example. The top stratum cannot derive any type information from the context if there is no code in the context.

2. It may not be possible to create a context that has the right types to replace a type annotation. For example, to make OutsideIn accept `eval` without type annotations, a programmer must create an expression with type `Term a` → `a` (or, equivalently, two expressions with types `Term a` and `a`) without using type annotations or GADT patterns. This task turns out to be impossible: the data constructors of `Term` build only values with types `Term Int`, `Term Bool`, or `Term (`x`,`y`)` where x, y are both types. Therefore type annotation remains necessary for the `eval` function.

The general characterization of the OutsideIn algorithm is similar to that of wobbly types: OutsideIn is sound, and it is also complete for suitably annotated (or suitably structured) programs.

Type inference via Herbrand constraint abduction This work by Sulzmann et al. [43] also takes a constraint-based approach to type inference. The algorithm generates type implication constraints from program expressions, and it solves the constraints by constraint abduction over the Herbrand domain [24]. To work around the lack of principal types in the GADT type system, Sulzmann et al. restricts their constraint solver to *intuitive solutions*, which formalize the principle that a maximal type should not contain any superfluous type information. Combined with the requirement that solutions must also be *sensible* (which is analogous to consistency in OutsideIn [36, §3.2]), restricting types to those that correspond to intuitive solutions restores principal types in the GADT type system. For example, this algorithm infers the following intuitive and sensible type for the `repId` function (code reproduced from Figure 2.8, p. 41):

```
repId :: forall a b. Term a → [b] → [b]
repId e x = case e of
  RepInt i → x
  RepBool b → []
```

The type inference via Herbrand constraint abduction algorithm is sound and complete for the GADT type system restricted to "intuitive and sensible" types.

Type inference for guarded recursive data types This work by Stuckey and Sulzmann [42] appears to be a precursor to type inference via Herbrand constraint abduction [43]. It does not describe a concrete type inference algorithm, but instead sketches a few ideas that may be applicable to the GADT type inference problem. Stuckey and Sulzmann suggested that a type inference algorithm can reconcile the (different) types of GADT pattern-matching branches by exhaustive search over a finite solution space [42, §5]. To combat polymorphic recursion, they sketched an iterative method that appears similar to the extension to Algorithm $\mathcal{W}$ proposed by Mycroft [29, §6]. Stuckey and Sulzmann suggested that this iterative method may allow a type inference algorithm to accept some unannotated plain GADT programs that use polymorphic recursion.

2.5 TYPES IN GADT PATTERN-MATCHING BRANCHES

As a part of this dissertation research, I conducted a systematic investigation of how types relate in a GADT pattern-matching branch. The ALT-GADT type rule (Figure 2.7, p. 38) completely specifies how the various types associated with a GADT pattern-matching branch relate to each other. It is, however, difficult to study the type relations directly from the rule, because the relations are obscured by many trivial details — the name of the data constructor, variable names in the pattern, the assumptions that the branch body makes on the type environment — details that are only marginally relevant to the important question, which is

	The Parts of a Pattern Matching Branch		
Sources of Types	Pattern Matching	Variable Binding	Branch Body
Data Constructor	Pattern	Declared	–
Local Scope	–	Instantiated	Refined
Global Context	Scrutinee	–	Outer

Figure 2.9: This figure shows the six type roles involved in the process of typing a GADT pattern-matching branch.

how the plain GADT type system relates the various types associated with GADT pattern-matching branches.

In this section, I introduce a classification of types in GADT pattern-matching branches that I developed to illustrate the relations between those types.

2.5.1 Identifying attributes

Figure 2.9 (p. 49) shows how I classify the types in a GADT pattern-matching branch into six type roles. This classification identifies the role of every type in the ALT-GADT type rule (Figure 2.7, p. 38, also reproduced on p. 51) using two of its attributes: the source of the type information (rows in Figure 2.9) and the part of the pattern-matching branch that it describes (columns in Figure 2.9). I now elaborate these two attributes.

Every type in the ALT-GADT type rule comes from one of the following three sources, each of which corresponds to a row in Figure 2.9:

Data Constructor Types in this row come from the type of the data constructor that appears in the pattern of the branch. These types are declared by the programmer using `data` declarations in an enclosing context.

Global Context Types in this row come from the type judgment being defined by the ALT-GADT rule. In other words, they show up below the horizontal line in the ALT-GADT rule.

Local Scope Types in this row do not have a single source; they are generated by combining the data constructor types with the global context types.

Every type in the ALT-GADT type rule is associated with one of the following three parts of a pattern-matching branch, each of which corresponds to a column in Figure 2.9 (p. 49):

Pattern Matching Types in this column describe the `case` scrutinee and the pattern part of a pattern-matching branch.

Variable Binding Types in this column describe the pattern-bound variables in a pattern-matching branch.

Branch Body Types in this column either appear in the type environment or describe the body of a pattern-matching branch. Types in the type environment are closely related (by the variable rule VAR) to types describing the branch body, so it makes sense to treat them as a group. Schrijvers et al. [36, §4.4, Footnote 4] also made a similar observation.

2.5.2 Decomposing ALT-GADT

Multiplying three rows by three columns gives nine positions in Figure 2.9 (p. 49), each of which corresponds to a possible role for types in the ALT-GADT type rule. Three of those roles are uninhabited: for example, nothing from the data constructor type directly describes the branch body, and nothing from the global context directly describes pattern-bound variables. The remaining six inhabited roles correspond to the six labeled positions in Figure 2.9. Here are how types in the ALT-GADT type rule (reproduced here from Figure 2.7, p. 38) map to these six inhabited roles:

$$\text{ALT-GADT}\quad \frac{\begin{array}{cc} \mathrm{C} : \forall\overline{\alpha}.\,\overline{w} \to T\ \overline{s} & \overline{\alpha} \,\#\, \mathit{tyvar}(\Gamma, \overline{u}, t) \\ \theta = \mathit{mgu}(T\ \overline{u} \sim T\ \overline{s}) & \theta(\Gamma\{\overline{\mathrm{x} : w}\}) \vdash \mathrm{c} : \theta(t) \end{array}}{\Gamma \vdash_p \mathrm{C}\ \overline{\mathrm{x}} \to \mathrm{c} : T\ \overline{u} \to t}$$

Pattern (Data Constructor, Pattern Matching)

This role is filled by the pattern type ($T\ \overline{s}$), which is also the range type of the data constructor (C) in the pattern.

Scrutinee (Global Context, Pattern Matching)

This role is filled by the type ($T\ \overline{u}$) of the `case` scrutinee, which appears in the context of the pattern-matching branch.

Declared (Data Constructor, Variable Binding)

This role is filled by the argument types ($\overline{w}$) of the data constructor (C) in the pattern. These types are a template for the types of the pattern-bound variables, which I will describe next.

Instantiated (Local Scope, Variable Binding)

This role is filled by the types ($\theta(\overline{w})$) of the pattern-bound variables ($\overline{\mathrm{x}}$). The ALT-GADT rule generates them by instantiating the data constructor argument types ($\overline{w}$) with the most-general unifier (θ) of the scrutinee and the pattern types. Intuitively, the most-general unifier carries type parametrization information from the scrutinee to the pattern-bound variables.

Outer (Global Context, Branch Body)

This role is filled by types in the type environment (Γ) and the type (t) of the entire `case` expression.

$$\theta = mgu(\text{Scrutinee} \sim \text{Pattern})$$

$$\text{Instantiated} = \theta(\text{Declared})$$

$$\text{Refined} = \theta(\text{Outer})$$

Figure 2.10: Relations between type roles in ALT-GADT.

Refined (Local Scope, Branch Body)

This role is filled by types in the refined type environment ($\theta(\Gamma)$), which the ALT-GADT rule uses to type the branch body, and the type ($\theta(t)$) of the branch body (c). The ALT-GADT rule generates them by instantiating the type environment (Γ) and the `case` expression type (t) with the most-general unifier (θ) of the scrutinee and the pattern types. Intuitively, the most-general unifier derives GADT type refinement from the GADT data constructor and introduces them as local type assumptions in the scope of the pattern-matching branch.

In the next subsection, I describe how this classification of type roles brings out a hidden symmetry in the ALT-GADT type rule.

2.5.3 Symmetry in type roles

In this subsection, I describe an intriguing symmetry between the six type roles of a GADT pattern-matching branch.[7] The three equations in Figure 2.10 (p. 52) summarize the relations between the six type roles in the ALT-GADT type rule. At the most abstract level, type checking a GADT pattern-matching branch is equivalent to checking the validity of these three type equations:

[7] The ideas that I describe in this subsection are not essential for the technical development in the remainder of this dissertation.

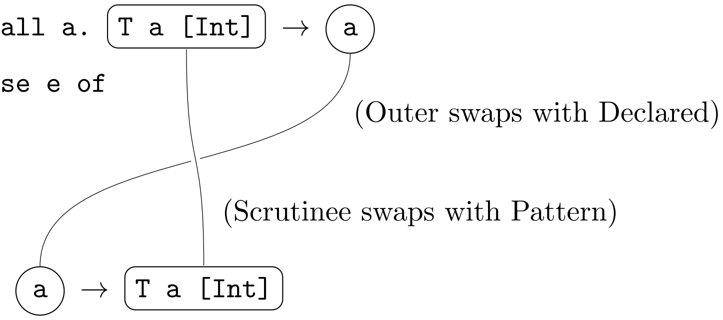
- The first equation, $\theta = mgu(\text{Scrutinee} \sim \text{Pattern})$, relates the two Pattern Matching type roles (first column, Figure 2.9, p. 49). It uses unification [35] to combine type information from Data Constructor with type information from Global Context.

- The second equation, $\text{Instantiated} = \theta(\text{Declared})$, relates the two Variable Binding type roles (Figure 2.9, second column). It generates the types in Local Scope by instantiating the types from Data Constructor.

- The third equation, $\text{Refined} = \theta(\text{Outer})$, relates the two Branch Body type roles (Figure 2.9, third column). It generates the types in Local Scope by refining the types from Global Context.

The invariance of these equations under a permutation of type roles exhibits a symmetry in the type roles of a GADT pattern-matching branch. This symmetry explains the feeling of *déjà vu* when a programmer sees the following two programs (do not worry about what they do; instead focus on how the same types appear in different places in the two programs):

```
data T a b where
  C :: forall b. b → T Int b
debitcard :: forall a. T a [Int] → a
debitcard e = case e of
  C x → head x
```



(Outer swaps with Declared)

(Scrutinee swaps with Pattern)

```
data T a b where
  C :: forall a. a → T a [Int]
badcredit :: forall b. T Int b → b
badcredit e = case e of
  C y → [y]
```

	The Parts of a Pattern Matching Branch		
Sources of Types	Pattern Matching	Variable Binding	Branch Body
Data Constructor	Pattern	Declared	–
Local Scope	–	Instantiated	Refined
Global Context	Scrutinee	–	Outer

Data Constructor (row 1) ↔ Global Context (row 3)

Variable Binding (column 2) ↔ Branch Body (column 3)

Sources of Types	Pattern Matching	Branch Body	Variable Binding
Global Context	Scrutinee	Outer	–
Local Scope	–	Refined	Instantiated
Data Constructor	Pattern	–	Declared

Figure 2.11: This figure illustrates a symmetry in the type roles of plain GADT pattern-matching branches. Applying the attribute permutation in the middle to the top table (reproduced from Figure 2.9, p. 49) produces the bottom table. The attribute permutation naturally induces a type-role permutation (shown here as the changing labels between the corresponding cells of the two tables), and the invariance of the role relations (Figure 2.10, p. 52) under this induced type-role permutation demonstrates the symmetry.

I omitted the line between the unannotated type `[Int]` of `x` in `debitcard` (Instantiated) and the unannotated type `[Int]` of `[y]` in `badcredit` (Refined). I also omitted the lines that connect the types `b`, `T Int b`, and `Int` (of `head x` in `debitcard` and `y` in `badcredit`) in the two programs.

Figure 2.11 (p. 54) illustrates this type symmetry using type roles. Consider the following permutation of attributes that I used to classify types into roles:

Data Constructor ↔ Global Context

Variable Binding ↔ Branch Body

Comparing the two tables in Figure 2.11 (p. 54) reveals that the attribute permutation induces the following permutation of type roles:

Scrutinee ↔ Pattern

Instantiated ↔ Refined

Declared ↔ Outer

The equations in Figure 2.10 (p. 52), considered as a whole, are an invariant under this role permutation, and the invariance exhibits a symmetry in the type roles. This symmetry reveals two properties about the ALT-GADT type rule (and any type checker that implements this rule):

1. At the abstract level, ALT-GADT is oblivious to the distinction between the types from the data constructor and types from the context. This property comes from the first attribute permutation:

   Data Constructor ↔ Global Context

2. At the abstract level, ALT-GADT is oblivious to the distinction between the types of the pattern-bound variables and types of the branch body. This property comes from the second attribute permutation:

   Variable Binding ↔ Branch Body

The properties hold because the permutation invariant, discussed in the beginning of this subsection, is an abstract model of the ALT-GADT type rule.

To put it differently, this symmetry shows that, even though the instantiation of pattern-bound variable types and the refinement of GADT branch body types are two distinct features, they both rely on the same underlying mechanism. It is possible, at least in principle, to permute the type roles in the input of a suitably structured ALT-GADT type checker without affecting its proper functioning. In other words, the new features that the GADT type system adds to algebraic data types (§2.3) are not an ad hoc design, but a natural extension that transfers type information symmetrically to Data Constructor and to Global Context types.

This type-role symmetry provides deep insight into the structure of a GADT type *checking* algorithm. Unfortunately, it reveals very little about the structure of GADT type *inference* algorithms. This difference is due to the first attribute permutation that I used earlier:

$$\text{Data Constructor} \leftrightarrow \text{Global Context}$$

While the distinction between the types from the data constructor and types from the context may be superficial to a type checker, the distinction is fundamental to a GADT type inference algorithm: the former (types from the data constructor) are declared by the programmer, but the latter (types from the context) must be inferred by the algorithm. This asymmetry between Data Constructor and Global Context type roles partly explains why GADT type inference turns out to be more difficult than ADT type inference.

Chapter 3

THE POINTWISE GADT TYPE SYSTEM

The plain GADT type system uses unification [35] to combine type information from the scrutinee and the pattern types of a pattern-matching branch. This approach is very general, but it also makes the plain GADT type system accept certain pathological functions with only counter-intuitive types. Thus, even though programmers rarely take advantage of this generality, its mere existence significantly complicates the type inference problem. In this chapter, I investigate this phenomenon in depth and propose a variation of the plain GADT type system — which I call the *Pointwise* GADT type system — that accepts most typical plain GADT programs but rejects the pathological functions.

3.1 TYPES IN GADT PATTERNS

This chapter is dedicated to the Pattern Matching types (first column, Figure 2.9, p. 49) in a GADT pattern-matching branch, so let us start by observing how type information flows between scrutinee types and pattern types in a typical GADT program.

3.1.1 Pointwise type information flow

Figure 3.1 (p. 58) shows a function `isNat`, which takes an integer list as argument and returns as result a Boolean list of the same length. The scrutinee type and the pattern type for its `Nil` branch are as follows:

```
data Z

data S n

data L n a where
  Nil  :: forall a. L Z a
  Cons :: forall n a. a → L n a → L (S n) a

isNat :: forall m. L m Int → L m Bool
isNat xs = case xs of
  Nil       → Nil
  Cons y ys → Cons (y >= 0) (isNat ys)
```

Figure 3.1: This figure demonstrates a length-preserving list transformation function in the GADT type system. The function `isNat` maps each element in a list of integers to a Boolean value: `True` if the element is a natural number, and `False` otherwise. The `L` data type is a list type with two type arguments: the first tracks the length of the list using Peano's encoding of natural numbers, and the second represents the type of the list elements. The type constructor `Z` represents the length zero, and the type constructor `S` represents a length increment. Since both the argument and the result of `isNat` are lists of length `m`, the type system guarantees that `isNat` must be a length-preserving list transformation function. This is one of the standard applications of the GADT type system.

Pattern	`Nil`,	Pattern Type	`L Z a`
Scrutinee	`xs`,	Scrutinee Type	`L m Int`

Since the `isNat` function is well-typed, these two types must be unifiable (see the ALT-GADT rule in Figure 2.7, p. 38). The type information flow between these two types is captured by their most-general unifier $\theta = [\mathtt{Z}/\mathtt{m}, \mathtt{Int}/\mathtt{a}]$. The following diagram visualizes the type information flow that θ represents:

$$\begin{tikzcd}
\text{Pattern Type} & \mathtt{L} \arrow[d, equal] & \mathtt{Z} \arrow[d] & \mathtt{a} \\
\text{Scrutinee Type} & \mathtt{L} & \mathtt{m} & \mathtt{Int} \arrow[u]
\end{tikzcd}$$

In the diagram, the type constructors `L` match, so they are linked by a double line. Type information, represented by arrows, flows from the first type argument (`Z`) of the pattern type to the corresponding part `m` in the scrutinee type, and from the second type argument (`Int`) of the scrutinee type to the corresponding part `a` in the pattern type. Type information flow in θ is *pointwise*: the arrows, which point either straight up or straight down, connect only the corresponding parts of the scrutinee type and the pattern type. The diagram for the `Cons` branch in `isNat` is similar:

$$\begin{tikzcd}
\text{Pattern Type} & \mathtt{L} \arrow[d, equal] & \mathtt{(S\ n)} \arrow[d] & \mathtt{a} \\
\text{Scrutinee Type} & \mathtt{L} & \mathtt{m} & \mathtt{Int} \arrow[u]
\end{tikzcd}$$

Again, type information flows only between the corresponding parts of the two types. The `isNat` function is not a special case: GADT patterns in a wide range of applications exhibit the same pointwise structure in their type information flow. To facilitate discussion, I use the following names to describe the two kinds of pointwise type information flow in GADT patterns:

- *Parametric instantiation* is pointwise type information flow from the scrutinee type to the pattern type. It corresponds to arrows that go straight up in the diagrams. Parametric instantiation may occur in both ADT and GADT patterns, and it is the only kind of type information flow supported by the ADT type system.

- *Type indexing* is pointwise type information flow from the pattern type to the scrutinee type. It corresponds to arrows that go straight down in the diagrams. Type indexing is specific to GADT patterns and is not applicable to ADT patterns: since an ADT data constructor must have a uniform range type, an ADT pattern type contains no type information that must flow to the scrutinee type, and thus there is no need for type indexing. As the `isNat` function demonstrates, parametric instantiation and type indexing can coexist in the same GADT pattern.

Given a GADT pattern-matching branch, the ALT-GADT type rule compares the scrutinee type to the pattern type, and it propagates type information in the *specific-to-generic* direction. Parametric instantiation occurs when the scrutinee type is more specific than the pattern type, and type indexing occurs when the converse is true.

In this chapter, I focus exclusively on the scrutinee and the pattern types in a GADT pattern-matching branch; both parametric instantiation and type indexing describe only type information flow between scrutinee types and pattern types. In contrast, GADT type refinement, which I introduced in §2.3, describes type information flow from a pattern type to the type of the enclosing `case` expression. Type information flow that involves pattern-bound variables or the branch body also brings up interesting issues, but I will save that discussion for later chapters and limit the scope of this chapter to scrutinee types and pattern types.

Why is pointwise type information flow so prevalent between scrutinee types and pattern types? I attribute the prevalence to the principle of orthogonal design [33, Chapter 4], which, in the context of generalized algebraic data types, suggests that programmers should represent orthogonal properties of an algebraic data value with different type arguments of the type constructor. Let us consider homogeneous lists as an example. Since the length of a list and the type of list elements are orthogonal properties (*e.g.,* an integer list may contain any number of elements, and a list of length five may contain elements of any type), orthogonal design calls for using different type arguments to represent these two properties. Indeed, the `L` data type uses its first type argument to track the length of the list, and its second type argument to represent the type of list elements. Since the two type arguments of `L` represent orthogonal properties, the absence of cross type information flow between them is only natural.

From this perspective, pointwise type information flow between scrutinee types and pattern types is not a new invention — it is only a new phenomenon that results from the long-standing principle of orthogonal design.

3.1.2 Unrestricted type information flow

The plain GADT type system places no restrictions on the type information flow between the scrutinee type and the pattern type in a pattern-matching branch. Since the ALT-GADT type rule (Figure 2.7, p. 38) uses unification [35] to compute the type information flow between scrutinee types and pattern types, the plain GADT type system has no problem dealing with GADT patterns that require non-pointwise type information flow. Most GADT programs, however, require only pointwise type information flow between scrutinee types and pattern types. How does the plain GADT type system, which supports unrestricted type information flow between scrutinee types and pattern types, compare against a *Pointwise* GADT type system, which supports only pointwise type information

```
data EquLR a b c where
  EquL :: forall a b. EquLR a a b
  EquR :: forall a b. EquLR a b b

vary e y = case e of
  EquL → y == 'c'
  EquR → [if y then 5 else 7]
```

Figure 3.2: This figure demonstrates a plain GADT program that requires non-pointwise type information flow between the scrutinee types and the pattern types of its GADT pattern-matching branches. The range type of the `EquL` data constructor equates the left-two type arguments of `EquLR`, and `EquR` does the same for the right-two type arguments. Even though the two pattern-matching branches in the `vary` function make arbitrarily different assumptions about the type environment and have arbitrarily different types, the function remains well typed in the plain GADT type system. All of its (infinitely many) types, however, make essential use of non-pointwise type information flow between scrutinee types and pattern types.

flow?

The plain GADT type system is obviously more expressive (*i.e.,* it accepts more programs) than the Pointwise GADT type system. This additional expressiveness is, however, a mixed blessing: it also allows the plain GADT type system to accept a class of pathological programs that have only counter-intuitive types. Figure 3.2 (p. 62) shows one such program. The `vary` function contains two pattern matching branches, and the branch bodies require very different type judgments:

$$\{\texttt{y} : \texttt{Char}\} \vdash (\texttt{y == 'c'}) : \texttt{Bool}$$

$$\{\texttt{y} : \texttt{Bool}\} \vdash \texttt{[if y then 5 else 7]} : \texttt{[Int]}$$

Typically, GADT type refinements allow branch body types to vary only with the pattern type (or, more precisely, only with the GADT type arguments of the matched data constructor). For example, in the `eval` function (Figure 2.6, p. 34), the body of the `RepInt` branch (which has pattern type `Term Int`) has type `Int`, and the body of the `RepBool` branch (which has pattern type `Term Bool`) has type `Bool`. This close correspondence between the pattern type and the branch body type helps programmers understand the structure of `case` expressions with GADT patterns. The `vary` function does not share this nice property; its pattern types say nothing about characters, Booleans, or lists of integers.

It is hard to imagine what types `vary` could possibly have. Surely `vary` must not be well-typed in the plain GADT type system? Alas, it turns out that the function *is* well-typed. Here are three of its infinitely many types:

```
forall q r. EquLR (Char, Bool) (q, r) (Bool, [Int]) → q → r

forall q r. EquLR (Bool, Char) (r, q) ([Int], Bool) → q → r

forall q r. EquLR (Bool, [Char]) (r, [q]) ([Int], [Bool]) → q → r
```

The function `vary` is well-typed, not because of parametric instantiation or type indexing, but because of something else altogether. The repeated occurrences of type variables in the data-constructor range types (`a` in the range type of `EquL` and `b` in the range type of `EquR`) *reflect* parametric instantiations *back* as branch-specific type indexing, so the type variables `q` and `r` can refine to different types in different pattern-matching branches. The following diagram illustrates the type information flow for the `EquL` pattern under the first type (of the three listed):

$$\begin{tikzcd}
\text{Pattern Type} & \texttt{EquLR} \arrow[d, equal] & \texttt{a} \arrow[r, dashed] & \texttt{a} \arrow[d] & \texttt{b} \\
\text{Scrutinee Type} & \texttt{EquLR} & \texttt{(Char, Bool)} \arrow[u] & \texttt{(q, r)} & \texttt{(Bool, [Int])} \arrow[u]
\end{tikzcd}$$

The type information flow illustrated in this diagram is not pointwise because type information propagates from the first type argument to the second. The occurrences of the type variable `a` serve as a back channel of type information between different parts of the scrutinee type.

This example suggests that unification may be too powerful for combining type information from scrutinee types and pattern types. Not only is the function `vary` well-typed in the plain GADT type system, but so are the following variations, which have different branch body types:

```
vary1 e y = case e of
  EquL → y == [’c’]            -- [Char], not Char
  EquR → [if y then 5 else 7]

vary2 e y = case e of
  EquL → y == ’c’
  EquR → if y then 5 else 7    -- Int, not [Int]
```

Let expressions e_1 and e_2 be given such that each expression is well-typed under the assumption that `y` has a monomorphic type. Regardless of what types e_1 and e_2 have, or what requirements they place on `y`, the following variation `varyX` of the function `vary` is well-typed in the plain GADT type system:

```
varyX e y = case e of
  EquL → $e_1$
  EquR → $e_2$
```

The existence of such functions seriously undermines programmers’ expectation that the plain GADT type system always enforces type coherence across pattern-matching branches. Current language implementations require type annotations to accept functions like `vary`, and there are good arguments for continuing the status quo:

1. Designing a type inference algorithm for `vary` is difficult because the types of the data constructors `EquL` and `EquR` are totally useless in computing the type refinement for each branch.

2. The function `vary` has an infinite number of types that are not instances of each other. Sulzmann et al. [43, §2.1] also demonstrated a GADT function that has an infinite number of types (§2.4), but there is an important difference between the two examples. While the example by Sulzmann et al. has infinite variations in the environment types, `vary` has infinite variations in the scrutinee type, thus presenting a different challenge to type inference algorithms.

3. Even if someone comes up with an algorithm that can infer a type for functions such as `vary`, such a type inference algorithm may be undesirable from a software engineering perspective. A type inference algorithm should be nothing more than a labor-saving device; if it can infer types for programs that are not obviously well-typed to programmers, then the inference algorithm might just turn out be too clever for its own good.

Type inference for functions like `vary` is not only hard (arguments 1, 2) and counterproductive (argument 3), it is also pointless because programmers rarely write these kinds of functions (I present evidence in §3.3). However, to exclude functions like `vary` from the design goals of a practical type inference algorithm, there needs to be a formal criterion that separates `vary` from practical GADT programs. I propose pointwise type information flow in GADT patterns as such a criterion, and I will, in the remainder of the chapter, develop the *Pointwise GADT* type system to formalize the pointwise type information flow restriction.

3.2 TYPE SYSTEM

In this section, I propose pointwise unifiers, which are special idempotent most-general unifiers that allow information to flow only between the corresponding parts of two types.

3.2.1 Pointwise unifiers

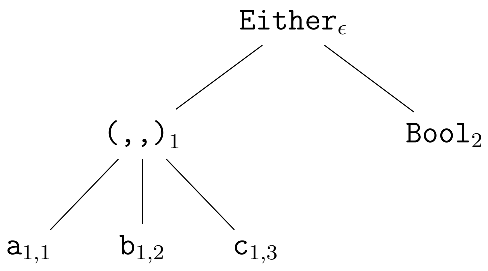
Let us start with a way to identify a part of a type by address.

Definition 1. A *path* is a sequence of positive integers $p = a_1, a_2, \ldots, a_n$ that serve as an "address" in a type. The expression $t \triangleright p$ (read "subterm of t at p") identifies a specific subterm of a type t at path p:

$$\begin{array}{rrl} & t \triangleright \epsilon = t & \\ 1 \leq a \leq m & (T\ t_1 \ldots t_m) \triangleright (a : r) = t_a \triangleright r & \\ a > m & (T\ t_1 \ldots t_m) \triangleright (a : r) & \text{is undefined} \\ & \alpha \triangleright (a : r) & \text{is undefined} \end{array}$$

where ϵ represents the empty sequence, and $(a : r)$ represents the number a followed by the sequence r. Path p is *valid* in t if $t \triangleright p$ is defined. If a path p is valid in both s and t, $s \triangleright p$ and $t \triangleright p$ are *pointwise counterparts* of s and t. ❋

Example Let us consider paths in $s =$ `Either (a,b,c) Bool`, illustrated here as a type tree. The subscript next to each tree node label represents the path to that particular subtree.



To put it another way,

$$s \triangleright \epsilon = \texttt{Either (a,b,c) Bool} \qquad s \triangleright 2 = \texttt{Bool}$$

$$s \triangleright 1 = \texttt{(a,b,c)} \qquad s \triangleright 1,1 = \texttt{a} \qquad s \triangleright 1,3 = \texttt{c}$$

Definition 2. Given types s and t, a substitution θ is a *pointwise unifier* of $s \sim t$ (written as $\theta = \mathit{pwu}(s \sim t)$) if the following conditions hold:

A1. $\mathrm{dom}(\theta) \subseteq \mathit{tyvar}(s, t)$,

A2. $\theta(s) = \theta(t)$, and

A3. Let $\alpha \in \mathrm{dom}(\theta)$. If $s \triangleright p = \alpha$, then $t \triangleright p = \theta(\alpha)$. Similarly, if $t \triangleright p = \alpha$, then $s \triangleright p = \theta(\alpha)$.

If the unification problem $s \sim t$ has a pointwise unifier, the types s and t are *pointwise unifiable.* ❋

Examples Here I list two positive and three negative examples of pointwise unifiers to demonstrate how Condition A3 in Definition 2 works:

1. $\theta = [\texttt{Int/a}, \texttt{Bool/b}]$ is a pointwise unifier of $\texttt{(Int,Bool)} \sim \texttt{(a,b)}$.

2. $\theta = [\texttt{Int/a}, \texttt{Int/b}]$ is *not* a pointwise unifier of $\texttt{a} \sim \texttt{b}$ because $\texttt{a} \in \mathrm{dom}(\theta)$ but $\theta(\texttt{a}) = \texttt{Int} \neq \texttt{b}$, violating A3.

3. $\theta = [\texttt{a/c}, \texttt{a/b}]$ is a pointwise unifier of $\texttt{(b,a)} \sim \texttt{(a,c)}$.

4. θ = [`c/a`, `c/b`] is *not* a pointwise unifier of `(b,a)` $\sim$ `(a,c)` because `b` $\in$ $\mathrm{dom}(\theta)$ but θ(`b`) = `c` $\neq$ `a`, violating A3.

5. θ = [`Int/a`, `Int/b`] is *not* a pointwise unifier of `(Int,a)` $\sim$ `(a,b)` because `a` $\in$ $\mathrm{dom}(\theta)$ but θ(`a`) = `Int` $\neq$ `b`, violating A3. This particular problem does not have a pointwise unifier.

I now state and prove some properties of pointwise unifiers, which I will use in the upcoming analysis of the pointwise unification algorithm and the Pointwise GADT type system. Since the definition of pointwise unifiers is symmetric, all theorems still hold after switching s and t around.

I start with three lemmas that decide, in a pointwise manner, whether two types are equal under substitution.

Definition 3. If a type t is built from a type constructor, then $\textsc{Tc}(t)$ extracts the name of the type constructor at the root of the type t. In mathematical notation,

$$\textsc{Tc}(T\ t_1 \ldots t_m) = T$$

$\textsc{Tc}(t)$ is undefined if t is a type variable. ❊

Lemma 1. Let types s, t, and substitution θ be given. $\theta(s) = t$ iff the following conditions hold for all paths p valid in s:

1. p is valid in t,

2. If $s \triangleright p$ is a type variable, then $\theta(s \triangleright p) = t \triangleright p$.

3. If $s \triangleright p$ is built from a type constructor, then $t \triangleright p$ is also built from a type constructor, and $\textsc{Tc}(s \triangleright p) = \textsc{Tc}(t \triangleright p)$.

Proof. $\Rightarrow$ Trivial.

$\Leftarrow$ I conduct this part of the proof by structural induction over s.

Base Case. If s is a type variable, the only path valid in s is the empty path ϵ, which must also be valid in t. From $\theta(s \triangleright \epsilon) = t \triangleright \epsilon$ we know $\theta(s) = t$.

Induction Step. If s is built from a type constructor U, then we can assume $s = U\, s_1 \ldots s_m$. From $\textsc{Tc}(s \triangleright \epsilon) = U = \textsc{Tc}(t \triangleright \epsilon)$ we know $t = U\, t_1 \ldots t_m$, and I now prove that $\theta(s_a) = t_a$ for all $1 \leq a \leq m$.

Let a be given and r be a path valid in s_a.

1. r is valid in t_a because $(a : r)$ is valid in s and t.

2. If $s_a \triangleright r$ is a type variable, then
   $\theta(s_a \triangleright r) = \theta(s \triangleright (a : r)) = t \triangleright (a : r) = t_a \triangleright r$.

3. If $s_a \triangleright r$ is built from a type constructor, then $t_a \triangleright r$ is also built from a type constructor, and $\textsc{Tc}(s_a \triangleright r) = \textsc{Tc}(t_a \triangleright r)$. This condition holds because $\textsc{Tc}(s \triangleright (a : r)) = \textsc{Tc}(t \triangleright (a : r))$.

By induction, $\theta(s_a) = t_a$, and therefore $\theta(s) = t$. □

Lemma 2. Let types s, t be given such that, for all paths q valid in s and t, $\textsc{Tc}(s \triangleright q) = \textsc{Tc}(t \triangleright q)$ if they both exist. Then, if a path p is valid in s but not valid in t, there exists a prefix p' of p such that $t \triangleright p'$ is a type variable.

Proof. I conduct this proof by structural induction on t.

Base Case. If t is a type variable, then $p' = \epsilon$ satisfies the lemma.

Induction Step. If t is built from a type constructor, then $t = U\, t_1 \ldots t_m$. Since p is valid in s but not valid in t, we can assume $p = (a : r)$ where $1 \leq a \leq m$. Since $\textsc{Tc}(s \triangleright \epsilon) = \textsc{Tc}(t \triangleright \epsilon) = U$, we can assume $s = U\, s_1 \ldots s_m$.

We know that, for all paths q such that $\textsc{Tc}(s_a \triangleright q)$ and $\textsc{Tc}(t_a \triangleright q)$ are both defined, $\textsc{Tc}(s_a \triangleright q) = \textsc{Tc}(t_a \triangleright q)$ because

$$\textsc{Tc}(s_a \triangleright q) = \textsc{Tc}(s \triangleright (a : q)) = \textsc{Tc}(t \triangleright (a : q)) = \textsc{Tc}(t_a \triangleright q)$$

The path r is valid in s_a but not valid in t_a, so by induction there exists a prefix r' of r such that $t_a \triangleright r'$ is a type variable. Then $t \triangleright (a : r')$ is a type variable, and $p' = (a : r')$ satisfies the lemma. □

Lemma 3. Let types s, t, and substitution θ be given. $\theta(s) = \theta(t)$ iff the following holds for all paths p valid in both s and t:

1. If $s \triangleright p$ or $t \triangleright p$ is a type variable, then $\theta(s \triangleright p) = \theta(t \triangleright p)$.

2. If $s \triangleright p$ and $t \triangleright p$ are both built from type constructors, then $\textsc{Tc}(s \triangleright p) = \textsc{Tc}(t \triangleright p)$.

Proof. $\Rightarrow$ Trivial by Lemma 1.

$\Leftarrow$ I conduct this proof by reduction to Lemma 1. Let p be a path valid in s, then p may relate to s and t in the following ways:

1. p is valid in t, and either $s \triangleright p$ or $t \triangleright p$ is a type variable. Then $\theta(s \triangleright p) = \theta(t \triangleright p) = \theta(t) \triangleright p$.

2. p is valid in t, and both $s \triangleright p$ and $t \triangleright p$ are built from type constructors. Then $\textsc{Tc}(s \triangleright p) = \textsc{Tc}(t \triangleright p) = \textsc{Tc}(\theta(t) \triangleright p)$.

3. p is not valid in t. By Lemma 2, there exists a prefix p' of p such that p' is valid in t, $t \triangleright p'$ is a type variable, and $\theta(s \triangleright p') = \theta(t \triangleright p')$. Without loss of generality, let p be the concatenation of p' and q. If $s \triangleright p$ is a type variable, then

$$\theta(s \triangleright p) = \theta((s \triangleright p') \triangleright q) = \theta(t \triangleright p') \triangleright q = \theta(t) \triangleright p$$

   If $s \triangleright p$ is built from a type constructor, then

$$\textsc{Tc}(s \triangleright p) = \textsc{Tc}((s \triangleright p') \triangleright q) = \textsc{Tc}(\theta(t \triangleright p') \triangleright q) = \textsc{Tc}(\theta(t) \triangleright p)$$

Applying Lemma 1 to s and $\theta(t)$ proves that $\theta(s) = \theta(t)$. □

The next two theorems state that a pointwise unifier is an idempotent most-general unifier.

Theorem 4. Let θ be a pointwise unifier of $s \sim t$. θ is idempotent.

Proof. Let $\alpha \in \mathrm{dom}(\theta)$. Without loss of generality, I assume that there exists a path p such that $\alpha = s \triangleright p$. By Definition 2 and Lemma 3,

$$\theta(\alpha) = \theta(s \triangleright p) = t \triangleright p = \theta(t \triangleright p) = \theta(\theta(\alpha))$$

Therefore θ is idempotent. □

Theorem 5. Let θ be a pointwise unifier of $s \sim t$. Then θ is also a most-general unifier of $s \sim t$.

Proof. Let η be a unifier of $s \sim t$. I show that, for any type variable α, $\eta(\alpha) = (\eta \circ \theta)(\alpha)$, so θ is a most-general unifier of $s \sim t$.

Let α be given. If $\alpha \notin \mathrm{dom}(\theta)$, then $(\eta \circ \theta)(\alpha) = \eta(\alpha)$.

If $\alpha \in \mathrm{dom}(\theta)$, then $\alpha \in \mathit{tyvar}(s, t)$. Without loss of generality, I assume that $\alpha \in \mathit{tyvar}(s)$, so there exists a path p such that $s \triangleright p = \alpha$. Then $(\eta \circ \theta)(\alpha) = \eta(\theta(s \triangleright p)) = \eta(t \triangleright p) = \eta(\alpha)$.

Since $\eta = \eta \circ \theta$ for all unifiers η, θ is a most-general unifier of $s \sim t$. □

Definition 4. I define $\theta \barwedge S$ as the substitution θ with its domain restricted to the set of type variables S. ❊

Definition 5. Given types s, t, and a path p valid in both s and t, I define the substitution $(\theta \triangleright p)$ as $\theta \barwedge \mathit{tyvar}(s \triangleright p, t \triangleright p)$. ❊

I use $(\theta \triangleright p)$ only when s and t are clear from the context. The next theorem shows how a pointwise unifier of two types relates to the pointwise unifiers of their pointwise counterparts.

Theorem 6. Let θ be a pointwise unifier of $s \sim t$. For every path p that is valid in both s and t, $(\theta \triangleright p)$ is a pointwise unifier of $s \triangleright p \sim t \triangleright p$.

Proof. Trivial by applying the definition of pointwise unifiers. □

The next theorem states that a unifier that transfers information in only one direction is a pointwise unifier.

Theorem 7. Let types u, v, and a substitution σ be given such that they satisfy $\mathrm{dom}(\sigma) \mathrel{\#} \mathit{tyvar}(u)$ and $\mathrm{dom}(\sigma) \subseteq \mathit{tyvar}(v)$. If $u = \sigma(v)$, then σ is a pointwise unifier of $u \sim v$.

Proof. Trivial by applying the definition of pointwise unifiers. □

The next theorem states that, if two pointwise-unifiable types do not share type variables, you can factor their pointwise unifier into two separate substitutions.

Theorem 8. Let θ be a pointwise unifier of $s \sim t$, and

$$\theta_s = \theta \barwedge \mathit{tyvar}(s) \qquad \theta_t = \theta \barwedge \mathit{tyvar}(t)$$

If $\mathit{tyvar}(s) \mathrel{\#} \mathit{tyvar}(t)$, there exists a type u such that:

$$\theta_s(s) = \theta(s) = \theta(t) = \theta_t(t) \qquad \theta_s(u) = t \qquad \theta_t(u) = s$$

In other words, the following diagram commutes.

$$\begin{tikzcd}
u \arrow[r, "\theta_t"] \arrow[d, "\theta_s"'] & s \arrow[d, "\theta_s"] \\
t \arrow[r, "\theta_t"'] & \theta(t) = \theta(s)
\end{tikzcd}$$

Proof. It is trivial to prove that

$$\theta_s(s) = \theta(s) = \theta(t) = \theta_t(t)$$

Let $u = s \Yup_\theta t$ where I define the prune operator $(\Yup_\theta)$ as follows:

$$\begin{aligned} \text{if } \alpha \in \mathrm{dom}(\theta) \quad & \alpha \Yup_\theta y = \alpha \\ \text{if } \alpha \in \mathrm{dom}(\theta) \quad & x \Yup_\theta \alpha = \alpha \\ (T\ x_1 \ldots x_n) \Yup_\theta (T\ y_1 \ldots y_n) &= T\ (x_1 \Yup_\theta y_1) \ldots (x_n \Yup_\theta y_n) \end{aligned}$$

I can prove, by structural induction, that the following properties hold:

1. u is uniquely defined,

2. If path p is valid in u and $u \triangleright p = \alpha$ is a type variable, then $\alpha \in \mathrm{dom}(\theta)$, and either $s \triangleright p = \alpha$ or $t \triangleright p = \alpha$, and

3. If path p is valid in u and $u \triangleright p$ is built from a type constructor, then $\textsc{Tc}(s \triangleright p) = \textsc{Tc}(t \triangleright p) = \textsc{Tc}(u \triangleright p)$.

I omit the proof. Now I use these properties to prove that $\theta_s(u) = t$ (the proof for $\theta_t(u) = s$ is similar and omitted). Let p be a path such that $u \triangleright p$ is a type variable. If $u \triangleright p = s \triangleright p$, then

$$\theta_s(u \triangleright p) = \theta_s(s \triangleright p) = \theta(s \triangleright p) = t \triangleright p$$

because θ is a pointwise unifier of $s \sim t$. If $u \triangleright p = t \triangleright p$, then

$$\theta_s(u \triangleright p) = \theta_s(t \triangleright p) = t \triangleright p$$

as $t \triangleright p \notin \mathrm{dom}(\theta_s)$. Applying Lemma 1 proves $\theta_s(u) = t$. □

3.2.2 Pointwise unification

In this subsection, I present a pointwise unification algorithm. Intuitively, the algorithm finds type variables that should be in the domain of a unifier, and checks that all occurrences of those type variables satisfy condition A3. It works in three phases: counterpart collection, conflict resolution, and unifier generation. I present each phase in turn, along with the lemmas that I use to prove the correctness of pointwise unification at the end of this subsection.

Counterpart collection This phase computes $s \curlyvee t$, the unifiable pointwise counterparts of s and t that have all matching top-level type constructors stripped off. Here are some examples:

$$(\texttt{x} \rightarrow \texttt{y}) \curlyvee (\texttt{Int} \rightarrow \texttt{Bool}) = \{(\texttt{x}, \texttt{Int}), (\texttt{y}, \texttt{Bool})\}$$

$$(\texttt{x} \rightarrow \texttt{x}) \curlyvee (\texttt{y} \rightarrow \texttt{z}) = \{(\texttt{x}, \texttt{y}), (\texttt{x}, \texttt{z})\}$$

$$\texttt{(Bool, b, Char)} \curlyvee \texttt{(a, Int, a)} = \{(\texttt{Bool}, \texttt{a}), (\texttt{b}, \texttt{Int}), (\texttt{Char}, \texttt{a})\}$$

$$(\texttt{Either a Int}) \curlyvee (\texttt{Either b Bool}) = \bot$$

Counterpart collection for the last example fails because `Int` and `Bool` are not unifiable. Here is the formal definition:

$$\alpha \curlyvee t = \{(\alpha, t)\}$$

$$t \curlyvee \alpha = \{(t, \alpha)\}$$

$$(T\ s_1 \ldots s_n) \curlyvee (T\ t_1 \ldots t_n) = (s_1 \curlyvee t_1) \cup \cdots \cup (s_n \curlyvee t_n)$$

$$(T_1\ s_1 \ldots s_m) \curlyvee (T_2\ t_1 \ldots t_n) = \bot \qquad \text{if } T_1 \neq T_2$$

The symbol $\bot$ indicates failure. I define $\bot \cup e = e \cup \bot = \bot$, so any local failure implies global failure. When two pairs (x, y) and (y, x) are both present in a set, I arbitrarily keep one and drop the other, so for example $\{(x, y), (y, x)\}$ becomes $\{(x, y)\}$. Counterpart collection does not check any property specific to pointwise unifiers; $s \curlyvee t = \bot$ implies that s and t are not unifiable.

Lemma 9. $s \curlyvee t \neq \bot$ iff for all path p such that $\textsc{Tc}(s \triangleright p)$ and $\textsc{Tc}(t \triangleright p)$ both exist, $\textsc{Tc}(s \triangleright p) = \textsc{Tc}(t \triangleright p)$.

Lemma 10. If $(x, y) \in (s \curlyvee t)$, then x or y is a type variable, and there exists a path p such that $x = s \triangleright p$ and $y = t \triangleright p$.

Lemma 11. Given path p valid in s and t, if $s \triangleright p$ or $t \triangleright p$ is a type variable, then $(s \triangleright p, t \triangleright p) \in (s \curlyvee t)$ or $(t \triangleright p, s \triangleright p) \in (s \curlyvee t)$.

These three lemmas are easily proved by induction over the definition of $s \curlyvee t$.

Conflict resolution The conflict resolution phase enforces conditions A2 and A3 in Definition 2. This phase is defined by a rewrite system $\longmapsto$ that orients a set of pairs, so that the first component of each pair is a type variable that appears nowhere else. The set rewrites to $\bot$ if no such orientation exists. Some examples:

$$
\begin{array}{rl}
\{(\texttt{x}, \texttt{Int}), (\texttt{y}, \texttt{Bool})\} & \quad \text{is in normal form} \\
\{(\texttt{x}, \texttt{y}), (\texttt{x}, \texttt{z})\} & \longmapsto \{(\texttt{y}, \texttt{x}), (\texttt{z}, \texttt{x})\} \\
\{(\texttt{Bool}, \texttt{a}), (\texttt{b}, \texttt{Int}), (\texttt{Char}, \texttt{a})\} & \longmapsto \bot
\end{array}
$$

And here is the definition.

$$
\begin{array}{rl}
\alpha \in \mathit{tyvar}(E),\ \beta \notin \mathit{tyvar}(E) & \{(\alpha, \beta)\} \uplus E \longmapsto \{(\beta, \alpha)\} \uplus E \\
\alpha \neq \beta,\ \{\alpha, \beta\} \subseteq \mathit{tyvar}(E) & \{(\alpha, \beta)\} \uplus E \longmapsto \bot \\
\alpha \notin \mathit{tyvar}(E) \cup \mathit{tyvar}(T\ \overline{s}) & \{(T\ \overline{s}, \alpha)\} \uplus E \longmapsto \{(\alpha, T\ \overline{s})\} \uplus E \\
\alpha \in \mathit{tyvar}(E) \cup \mathit{tyvar}(T\ \overline{s}) & \{(T\ \overline{s}, \alpha)\} \uplus E \longmapsto \bot \\
\alpha \in \mathit{tyvar}(E) \cup \mathit{tyvar}(T\ \overline{s}) & \{(\alpha, T\ \overline{s})\} \uplus E \longmapsto \bot
\end{array}
$$

Conflict resolution fails if $s \curlyvee t \longmapsto^* \bot$. Note that each rewrite rule transforms only one element in the set; this property reflects the pointwise nature of pointwise unifiers.

Lemma 12. The rewrite system $\longmapsto$ is confluent and strongly normalizing over finite sets of pairs of types.

Proof. I establish strong normalization through a measure $n(E)$ that counts the number of one-step rewrites for E. The value $n(E)$ is finite for every finite E, every rewrite reduces $n(E)$ by at least 1, and n is always nonnegative. Thus every rewrite sequence must be finite.

I establish confluence through local confluence and strong normalization. Proof of local confluence is trivial because each element in E can trigger at most one rewrite, and the rewrite associated with an element (x_1, y_1) is not affected by replacing another element (x_2, y_2) with (y_2, x_2). □

Lemma 13. If $X \neq \bot$ and $X \longmapsto^* \bot$, then $X \longmapsto \bot$.

Proof. I prove this by induction over the rewrite sequence.

Base Case. If $X \longmapsto^* \bot$ happens in one step, then $X \longmapsto \bot$.

Induction Step. If $X \longmapsto^* \bot$ happens in n steps $(n > 1)$, then there exists X' and X'' such that $X \longmapsto^* X' \longmapsto X'' \longmapsto \bot$. I show that $X' \longmapsto X'' \longmapsto \bot$ implies $X' \longmapsto \bot$, and applying the induction principle completes the proof.

There are six rewrites for $X' \longmapsto X'' \longmapsto \bot$; since they are all similar, I will prove only one case and omit the others. Suppose

$$\begin{aligned} X' &= \{(\alpha', \beta')\} \uplus E' \longmapsto \{(\beta', \alpha')\} \uplus E' = X'' \\ X'' &= \{(\alpha'', \beta'')\} \uplus E'' \longmapsto \bot \\ &\qquad \alpha' \in \mathit{tyvar}(E'),\ \beta' \notin \mathit{tyvar}(E'),\ \alpha'' \neq \beta'',\ \{\alpha'', \beta''\} \subseteq \mathit{tyvar}(E'') \end{aligned}$$

Then clearly $\beta' \neq \alpha''$, $\beta' \neq \beta''$, $(\alpha'', \beta'') \in E'$, thus

$$X' = \{(\alpha'', \beta'')\} \uplus Y' \qquad \alpha'' \neq \beta'',\ \{\alpha'', \beta''\} \subseteq \mathit{tyvar}(Y')$$

And therefore $X' \longmapsto \bot$. □

Unifier generation This phase turns a properly-oriented set of type pairs into a substitution. If $E \neq \bot$,

$$\mathrm{sub}_E(\alpha) = \begin{cases} t & \text{if } (\alpha, t) \in E \\ \alpha & \text{otherwise} \end{cases}$$

Pointwise unification I have now introduced all three phases of the pointwise unification algorithm. To compute a pointwise unifier of two types s and t, simply compute $s \curlyvee t \longmapsto^* E$ such that E is in normal form. If $E \neq \bot$, then sub_E is a pointwise unifier of $s \sim t$. The following theorem proves the soundness and completeness of the algorithm.

Theorem 14. If types s, t are pointwise unifiable, counterpart collection and conflict resolution both succeed. In addition, if $s \curlyvee t \longmapsto^* E$ such that E is in normal form and $E \neq \bot$, then sub_E is a pointwise unifier of $s \sim t$.

Proof. I start with the first statement. Assuming that s, t are pointwise unifiable, then there exists a pointwise unifier θ of $s \sim t$. We know that $\textsc{Tc}(s \triangleright p) = \textsc{Tc}(t \triangleright p)$ for all paths p such that $\textsc{Tc}(s \triangleright p)$ and $\textsc{Tc}(t \triangleright p)$ both exist (Lemma 3), so $s \curlyvee t \neq \bot$ (Lemma 9).

Now I prove that $s \curlyvee t \not\longmapsto^* \bot$. From Lemma 13, it is sufficient to prove that $s \curlyvee t \not\longmapsto \bot$. Let us consider each possible rewrite to $\bot$ as follows:

1. If $s \curlyvee t = \{(\alpha, \beta)\} \uplus E'$, then there exists a path p such that $\alpha = s \triangleright p$ and $\beta = t \triangleright p$ (Lemma 10). If $\alpha \neq \beta$, then either $\alpha \in \mathrm{dom}(\theta)$ or $\beta \in \mathrm{dom}(\theta)$. Without loss of generality I assume $\alpha \in \mathrm{dom}(\theta)$. For all paths q, $s \triangleright q = \alpha$ implies $t \triangleright q = \beta$, $t \triangleright q = \alpha$ implies $s \triangleright q = \beta$, so $\alpha \notin \mathit{tyvar}(E')$, and $\{(\alpha, \beta)\} \uplus E' \not\longmapsto \bot$.

2. If $s \curlyvee t = \{(T\ \overline{s}, \alpha)\} \uplus E'$, then there exists a path p such that $T\ \overline{s} = s \triangleright p$, $\alpha = t \triangleright p$ (Lemma 10), and $\alpha \in \mathrm{dom}(\theta)$ (Lemma 3). For all paths q, $s \triangleright q = \alpha$

implies $t \triangleright q = T\ \overline{s}$, and $t \triangleright q = \alpha$ implies $s \triangleright q = T\ \overline{s}$. Therefore $\alpha \notin \mathit{tyvar}(E')$, and $\{(T\ \overline{s}, \alpha)\} \uplus E' \not\longmapsto \bot$.

3. If $s \curlyvee t = \{(\alpha, T\ \overline{s})\} \uplus E'$, then the same reasoning as in the previous case shows that $\{(\alpha, T\ \overline{s})\} \uplus E' \not\longmapsto \bot$.

The analysis shows that $s \curlyvee t \not\longmapsto \bot$, therefore $s \curlyvee t \not\longmapsto^* \bot$.

I now prove the second statement: sub_E is a pointwise unifier of $s \sim t$. I start by showing that sub_E is a substitution. Let $(\alpha, y) \in E$ and $(\alpha, y') \in E$. If $y \neq y'$, then $E \longmapsto \bot$, which contradicts the assumption that E is in normal form. Therefore $y = y'$ and $\mathrm{sub}_E(\alpha)$ is uniquely defined.

Finally, I check sub_E against Definition 2.

A1. $\mathrm{dom}(\mathrm{sub}_E) \subseteq \mathit{tyvar}(E) = \mathit{tyvar}(s \curlyvee t) = \mathit{tyvar}(s, t)$.

A2. θ is a unifier of $x \sim y$ for any $(x, y) \in s \curlyvee t$; from Lemma 3 and Lemma 11, we know θ is a unifier of $s \sim t$.

A3. Let $\alpha \in \mathrm{dom}(\mathrm{sub}_E)$, and without loss of generality assume $\alpha \in \mathit{tyvar}(s)$. Then there exists a path p such that $s \triangleright p = \alpha$. I first prove p is valid in t by contradiction. Assume to the contrary that p is invalid in t. Since $s \curlyvee t$ exists, there is a prefix q of p such that $t \triangleright q = \beta$. Then,

$$\{(\beta, s \triangleright q), (\alpha, \mathrm{sub}_E(\alpha))\} \subseteq E \quad \text{or}$$
$$\{(s \triangleright q, \beta), (\alpha, \mathrm{sub}_E(\alpha))\} \subseteq E$$

Since q is a prefix of p, $\alpha \in \mathit{tyvar}(s \triangleright q)$, so E is not in normal form, contradicting the assumption. Therefore p is valid in t.

Now I prove $t \triangleright p = \mathrm{sub}_E(\alpha)$ by contradiction. Assume to the contrary that $t \triangleright p \neq \mathrm{sub}_E(\alpha)$. Then, given $E \neq \bot$,

$$\{(\alpha, t \triangleright p), (\alpha, \mathrm{sub}_E(\alpha))\} \subseteq E \quad \text{or}$$
$$\{(t \triangleright p, \alpha), (\alpha, \mathrm{sub}_E(\alpha))\} \subseteq E$$

> Both cases imply that E is not in normal form, contradicting the assumption. Thus $t \triangleright p = \mathrm{sub}_E(s \triangleright p)$. The case where $\alpha \in \mathit{tyvar}(t)$ is similar and omitted here.

Therefore sub_E is a pointwise unifier of $s \sim t$. □

This proof shows that pointwise unification is sound and complete. Given two input types, pointwise unification computes their pointwise unifier if it exists, and the algorithm fails if the input types are not pointwise unifiable.

In the last part of this section, I present a formal definition of the Pointwise GADT type system.

3.2.3 Formal definition

Parametric instantiation and type indexing are easy for programmers to understand because they have a simple structure: type information flows only between the pointwise counterparts of the scrutinee type and the constructor range type. In this subsection, I define the Pointwise GADT type system, which accepts only pattern-matches that follow this simple structure of information flow.

The Pointwise GADT type system is nearly identical to the plain GADT type system (§2.3): they are built on the same programming language and place identical restrictions on data constructor types. Figure 3.3 (p. 80) shows the type rules of the Pointwise GADT type system, which are identical to the plain GADT type system (Figure 2.7, p. 38) except that here pointwise unifier (*pwu*) in the ALT-PTWISE type rule replaces most-general unifier (*mgu*) in the ALT-GADT type rule. The Pointwise GADT type system is a minimal adaptation of the plain GADT type system that enforces pointwise type information flow between scrutinee types and pattern types.

Since $\theta = \mathit{pwu}(s \sim t)$ implies $\theta = \mathit{mgu}(s \sim t)$ (Theorem 5, p. 71), the ALT-GADT type rule accepts all type judgments that the ALT-PTWISE type rule does.

VAR

$$\frac{\mathrm{x} : \forall\overline{\alpha}.\, t \in \Gamma \qquad s = \mathrm{inst}[\overline{\alpha}](t)}{\Gamma \vdash \mathrm{x} : s}$$

LAM

$$\frac{\Gamma\{\mathrm{u} : s\} \vdash \mathrm{e} : t}{\Gamma \vdash \lambda \mathrm{u}\,.\,\mathrm{e} : s \to t}$$

CONS

$$\frac{\mathrm{C} : \forall\overline{\alpha}.\, t \qquad s = \mathrm{inst}[\overline{\alpha}](t)}{\Gamma \vdash \mathrm{C} : s}$$

APP

$$\frac{\Gamma \vdash \mathrm{f} : t_1 \to t_2 \qquad \Gamma \vdash \mathrm{e} : t_1}{\Gamma \vdash \mathrm{f\ e} : t_2}$$

LETREC-P

$$\frac{\Gamma\{\mathrm{u} : \forall\overline{\alpha}.\, s\} \vdash \mathrm{e} : s \qquad \overline{\alpha} \,\#\, \mathit{tyvar}(\Gamma) \qquad \Gamma\{\mathrm{u} : \forall\overline{\alpha}.\, s\} \vdash \mathrm{d} : t}{\Gamma \vdash \texttt{let}\ \mathrm{u} = \mathrm{e}\ \texttt{in}\ \mathrm{d} : t}$$

CASE

$$\frac{\Gamma \vdash \mathrm{e} : s \qquad \Gamma \vdash_p \mathrm{p}_i \to \mathrm{c}_i : s \to t}{\Gamma \vdash \texttt{case}\ \mathrm{e}\ \texttt{of}\ \{\, \overline{\mathrm{p}_i \to \mathrm{c}_i}\,\} : t}$$

ALT-PTWISE

$$\frac{\begin{array}{cc} \mathrm{C} : \forall\overline{\alpha}.\, \overline{w} \to T\ \overline{s} & \overline{\alpha} \,\#\, \mathit{tyvar}(\Gamma, \overline{u}, t) \\ \theta = \mathit{pwu}(T\ \overline{u} \sim T\ \overline{s}) & \theta(\Gamma\{\overline{\mathrm{x} : w}\}) \vdash \mathrm{c} : \theta(t) \end{array}}{\Gamma \vdash_p \mathrm{C}\ \overline{\mathrm{x}} \to \mathrm{c} : T\ \overline{u} \to t}$$

$\mathrm{inst}[\overline{\alpha}](t) = \theta(t),$ where $\theta = [\overline{s/\alpha}],$ and $\overline{s}$ are arbitrary types

Figure 3.3: The Pointwise GADT type system.

Similarly, since $\theta = [\overline{u/\gamma}]$ implies $\theta = pwu(T\ \overline{\gamma} \sim T\ \overline{u})$ (Theorem 7, p. 72), the ALT-PTWISE type rule also accepts all type judgments that the ALT-ADT type rule does. In other words, every expression well-typed in the ADT type system is also well-typed in the Pointwise GADT type system, and every expression well-typed in the Pointwise GADT type system is also well-typed in the plain GADT type system.

Like the plain GADT type system, the Pointwise GADT type system also guarantees the type safety of well-typed programs. The proof of soundness is trivial because every well-typed Pointwise GADT program is also a well-typed plain GADT program.

3.3 EXPRESSIVENESS

The Pointwise GADT type system is more expressive than the ADT type system, but less expressive than plain GADT type system. This statement, while correct, says very little about how the Pointwise GADT type system measures up to the plain GADT type system. Can it accept the plain GADT programs that practical programmers write? What are some of the plain GADT programs that it rejects? Is it difficult to reimplement such examples so that they become well-typed in the Pointwise GADT type system?

The answers to these questions have profound implications for the practicality of a type inference algorithm designed for the Pointwise GADT type system. In this section, I answer these questions.

3.3.1 Case studies

As a rough gauge on how practical plain GADT programs in the wild fare in the Pointwise GADT type system, I studied the Omega programs that Sheard prepared for the 2006 Spring School on Generic Programming [37] and the 2007

Central European Functional Programming School [38].[1] These program examples cover a wide range of applications; all involving significant use of GADTs:

- Dimensional types [37, §4],
- Tagless term interpreter [38, §5.7],
- N-way zip (two implementations) [37, §5.3],
- Shape-indexed binary-tree paths [38, §3.3],
- AVL tree insertion and deletion [38, §4.1],
- Red-black tree insertion [38, Appendix A], and
- Witness for integer-arithmetic theorems [38, §3.7].

I conducted the case studies by copying the programs from the aforementioned two lecture notes, translating nested patterns into nested `case` expressions, and checking that the scrutinee type and the pattern type of each pattern-matching branch are pointwise unifiable. I performed all the checking manually, which is not as difficult as it may sound — in the common case where neither the scrutinee type nor the pattern type contains repeated occurrences of the same type variable, unifiability of the two types (which is required by the plain GADT type system) directly guarantees pointwise unifiability.

The studies find that Pointwise GADTs are expressive enough to type *all* of these computations. The AVL-tree example, which uses GADTs to enforce the balance invariant, requires some refactoring; all other examples are well-typed in the Pointwise GADT type system as they were written. (I will discuss the AVL-tree example in greater detail later in this section.) This result is, in some sense, not surprising: since programmers tend to follow the principle of orthogonal design by

[1] I studied all the programs in the lecture notes, except those that depend on other features of the Omega language, such as staged computation and type-level functions.

encoding different properties of a data value with different type indices, pointwise unification — which compares only the pointwise counterparts of the scrutinee type and the constructor range type — should be all that is necessary to type programs written in this manner.

In addition to these practical plain GADT programs, the Pointwise GADT type system also accepts some manufactured plain GADT programs that are not designed for a practical application. For example, the `repId` function (Figure 2.8, p. 41) is also well-typed (and in fact has the same infinite set of maximal types) in the Pointwise GADT type system. This example demonstrates that the Pointwise GADT type system, like the plain GADT type system, also lacks the principal type property.

3.3.2 Rejected programs

Another way to study the expressiveness of the Pointwise GADT type system is to study GADT programs that require non-pointwise type information flow between scrutinee types and pattern types — in other words, the well-typed plain GADT programs that the Pointwise GADT type system rejects. In this subsection, I first show three such program examples, and then I explain how to rewrite them as well-typed Pointwise GADT programs.

First example My first example is the function `vary` (Figure 3.2, p. 62), which I discussed in §3.1. Here I complement the earlier discussion by presenting a proof that `vary` is not well-typed in the Pointwise GADT type system.

Theorem 15. The function `vary` (Figure 3.2, p. 62) is not well-typed in the Pointwise GADT type system.

Proof. Assume to the contrary that `vary` is well-typed in the Pointwise GADT type system. From its definition, we can assume that its argument `e` has type

`EquLR` u v w, where u, v, and w are types. I perform the proof by analyzing what types u, v, and w can be.

1. If `vary` is well-typed in the Pointwise GADT type system, the types u, v, and w must not all be type variables. *Proof.* Assume to the contrary that they are all type variables, then `e` has five possible types (modulo type variable renaming):

   ```
   EquLR p p p
   EquLR p p q
   EquLR p q p
   EquLR p q q
   EquLR p q r
   ```

   None of them leads to a valid type of `vary`. For example, assume `e` has type `EquLR p q r`. The only options left for the type of `y` are `p`, `q`, `r`, or `s` (another type variable), or $T\ \overline{z}$, and none of them allows `y` to have type `Char` in the `EquL` branch and type `Bool` in the `EquR` branch. Therefore, u, v, and w are not all type variables.

2. If `vary` is well-typed in the Pointwise GADT type system, the types u, v, and w must be identical. *Proof.* Without loss of generality, assume that u is not a type variable. Since the `EquL` branch is well-typed, we know that the following unification problem has a pointwise unifier θ.

   $$\texttt{EquLR}\ \ u\ \ v\ \ w \sim \texttt{EquLR a a b}$$

   Since u is not a type variable, $\theta(\texttt{a}) = u$, and by Definition 2 we know $u = v$. Applying a similar argument to the `EquR` branch shows $v = w$.

3. If `vary` is well-typed in the Pointwise GADT type system, the branch bodies of the `EquL` and the `EquR` branches must be well-typed under the same type

> environment as the `case` expression. *Proof.* Consider the `EquL` branch. Given that t is not a type variable, the unification problem
>
> $$\mathtt{EquLR}\ t\ t\ t \sim \mathtt{EquLR\ a\ a\ b}$$
>
> has only one pointwise unifier $\theta = [t/\mathtt{a}, t/\mathtt{b}]$. Since $\{\mathtt{a}, \mathtt{b}\} \# \mathit{tyvar}(\Gamma)$, we know $\theta(\Gamma) = \Gamma$. The case for the `EquR` branch is similar and omitted.

The final fact requires that the argument `y` of `vary` has the same type in the `EquL` and the `EquR` pattern-matching branches, contradicting the assumption. So `vary` is not well-typed in the Pointwise GADT type system. □

Even though the proof is quite involved, there is an intuitive explanation on why `vary` is not well-typed in the Pointwise GADT type system. Typing `vary` in the plain GADT type system requires a trick: a type of `vary` must associate different occurrences of a pattern type variable with different types in the scrutinee type. For example, in the first type that I gave in §3.1, the type variable `a` in the range type of `EquL` corresponds to both `(Char,Bool)` and `(q,r)`.

Pointwise unification requires that a type variable must always correspond to a single type. The following diagram illustrates this requirement: $u = v$ must hold because $(\mathtt{EquLR\ a\ a\ b}) \triangleright 1 = (\mathtt{EquLR\ a\ a\ b}) \triangleright 2$. This requirement prevents the aforementioned trick, thus `vary` must be ill-typed in Pointwise GADTs.

$$\begin{tikzcd}
\mathtt{EquLR\ a\ a\ b} \arrow[dd, "\triangleright 1"'] \arrow[dr, "\triangleright 2"] & & \mathtt{EquLR}\ u\ v\ w \arrow[dd, "\triangleright 1"'] \arrow[dr, "\triangleright 2"] & \\
& \mathtt{a} \arrow[rr, "\theta"] \arrow[dl, equal] & & v \arrow[dl, equal] \\
\mathtt{a} \arrow[rr, "\theta"] & & u &
\end{tikzcd}$$

```
data Split a b where
  Whole :: Split Int Int
  Parts :: forall a b. Split (Int, a) (b, Bool)

joint :: forall x. Split x x → x
joint e = case e of
  Whole → 7
  Parts → (3, True)
```

Figure 3.4: A non-pointwise plain GADT program.

Second example Figure 3.4 (p. 86) shows another well-typed plain GADT program that is rejected by the Pointwise GADT type system. The `joint` function is, in some ways, a mirror image of `vary`, which is my first example. Here the repeated type variable occurrences appear in the scrutinee type instead of in the pattern type, and the `join` function relies on the non-pointwise type information flow to obtain the GADT type refinement [`(Int, Bool)`/`x`] in the `Parts` branch. The following diagram illustrates the type information flow in the `Parts` pattern:

$$\begin{tikzcd}
\text{Pattern Type} & \texttt{Split} \arrow[d, equal] & \texttt{(Int, a)} \arrow[d, leftrightarrow] & \texttt{(b, Bool)} \arrow[d, leftrightarrow] \\
\text{Scrutinee Type} & \texttt{Split} & \texttt{x} \arrow[r, leftrightarrow, dashed] & \texttt{x}
\end{tikzcd}$$

Theorem 16. The function `joint` is not well-typed in Pointwise GADTs.

Proof. Assume to the contrary that `joint` is well-typed in Pointwise GADTs. From the definition, we can assume that its argument `e` has type `Split` u v where u and v each represents a type. To make `joint` well typed, the following unification

```
data AVL n where
  Leaf :: AVL Z
  Node :: forall l r m. Balance l r m →
          AVL l → Int → AVL r → AVL (S m)

data Balance l r m where
  Less :: forall n. Balance n     (S n) (S n)
  Same :: forall n. Balance n     n     n
  More :: forall n. Balance (S n) n     (S n)
```

Figure 3.5: This figure demonstrates the `AVL` data type by Sheard [38, §4.1]. The `AVL` data type defines AVL balanced binary trees, and it uses Peano's encoding of natural numbers (Figure 3.1, p. 58) to track the depth of a tree. By using types to enforce the AVL balance invariant (the depths of the two sub-trees of an internal node may differ only by at most one level), the `AVL` data type ensures that a well-typed program cannot violate the balance invariant. The `Balance` data type represents the balance factor of a tree: −1 (`Less`), 0 (`Same`), or 1 (`More`).

problems must have pointwise unifiers:

$$\texttt{Split}\ u\ \ v \sim \texttt{Split Int Int}$$
$$\texttt{Split}\ u\ \ v \sim \texttt{Split (Int, a) (b, Bool)}$$

The existence of pointwise unifiers requires that u and v be different type variables, and there is no way to express the result type of the function. So `joint` is not well-typed in the Pointwise GADT type system. □

Third example I mentioned earlier that the AVL-tree example by Sheard [38, §4.1] is not well-typed in the Pointwise GADT type system. The data type, which I show in Figure 3.5 (p. 87), serves as my third example.

The AVL data type has only one type argument, which represents the depth of a tree. The data type uses this type argument to enforce the invariant that the balance factor of an internal node must be −1 (the right sub-tree is deeper than the left by one level), 0 (the two sub-trees have equal depth), or 1 (the left sub-tree is deeper than the right by one level). Instead of using three separate data constructors (each with a different type) to represent an internal node with each balance factor, the AVL data type employs a separate Balance data type to represent the balance factor of the node. The Balance value in Node associates three different types: the depth of the left sub-tree (l), the depth of the right sub-tree (r), and the maximum of the two depths (m).

Since only l and r are existentially quantified in Node (but not m), the Balance value naturally propagates type information from the type variable m to both l and r. This propagation constitutes non-pointwise type information flow, and it is the reason why the Pointwise GADT type system rejects the tree-rotation functions (excerpt here):

```
rotr :: forall h. AVL (S (S h)) → Int → AVL n →
        Either (AVL (S (S h))) (AVL (S (S (S h))))

rotr tree y c = case tree of
  Node bal a x b → case bal of
    Same → ...
```

The following diagram illustrates the non-pointwise type information flow in the inner Same pattern (dashed arrows between occurrences of the pattern type variable n indicates non-pointwise type information flow):

$$\begin{tikzcd}
\text{Pattern Type} & \texttt{Balance} \arrow[d, equal] & \texttt{n} \arrow[d] & \texttt{n} \arrow[d] & \texttt{n} \arrow[l, dashed] \arrow[ll, dashed, bend right] \\
\text{Scrutinee Type} & \texttt{Balance} & \texttt{l} & \texttt{r} & \texttt{(S h)} \arrow[u]
\end{tikzcd}$$

3.3.3 Workarounds

Do these three rejected program examples reflect any serious limitations in the expressiveness of the Pointwise GADT type system? My answer is *no* because all three examples can be rewritten as well-typed Pointwise GADT programs simply by following the principle of orthogonal design.

The `vary` function (Figure 3.2, p. 62) persuades the plain GADT type system into typing the `EquL` and the `EquR` pattern-matching branches under *arbitrarily different* type environments. This extreme level of flexibility exhibited by the plain GADT type system is arguably more of a curse than a blessing, and programmers are well advised to reimplement the `vary` function in the following way:

```
data W a where
  W1 :: Char → W Bool
  W2 :: Bool → W [Int]

vary :: forall r. W r → r
vary e = case e of
  W1 y → y == 'c'
  W2 y → [if y then 5 else 7]
```

This reimplemented function is clearer, arguably easier to understand, and well-typed in the Pointwise GADT type system. The second example `joint`, while not nearly as objectionable as `vary`, is also easily reimplemented as the following well-typed Pointwise GADT program:

```
data U a where
  U1 :: U Int
  U2 :: U (Int, Bool)

joint :: forall x. U x → x
joint e = case e of
```

```
  U1 → 7
  U2 → (3, True)
```

The third example is perhaps the most important because it arises out of a practical application. Recall that the problem with the `rotr` function is that the `AVL` data type forces the programmer to propagate type information in two steps: first to `m` using the `Node` pattern, then to the existential types `l` and `r` using the inner `Same` pattern. The following refactoring solves this problem:

```
data AVL n where
  Leaf :: AVL Z
  NodeL :: forall m. AVL m     → Int → AVL (S m) → AVL (S (S m))
  NodeS :: forall m. AVL m     → Int → AVL m     → AVL (S m)
  NodeM :: forall m. AVL (S m) → Int → AVL m     → AVL (S (S m))

rotr :: forall h. AVL (S (S h)) → Int → AVL n →
        Either (AVL (S (S h))) (AVL (S (S (S h))))

rotr tree y c = case tree of
  NodeS a x b → ...
```

By using three different data constructors to represent internal nodes with different balance factors, I eliminated the existential types `l`, `r` in the `Node` data constructor and the inner pattern `Same` in the original `rotr` function. Type information flow in the patterns are now pointwise, and the new `rotr` function is well-typed in the Pointwise GADT type system.

3.4 SUMMARY

In this chapter, I focused on the mechanism that the GADT type systems use to support parametric instantiation and type indexing. I argued that, although unification gets the job done, it is too powerful for this purpose, because it allows the plain GADT type system to accept some programs that programmers may not expect to be well-typed.

I proposed the Pointwise GADT type system, which separates practical plain GADT programs from such pathological programs. The Pointwise GADT type system works just like the plain GADT type system, except that it uses pointwise unifiers to support parametric instantiation and type indexing. Since a pointwise unifier propagates information only between the pointwise counterparts of the unified types, I believe that programmers should find it easier to see why a program is well-typed in the Pointwise GADT type system.

The Pointwise GADT type system is less expressive than the plain GADT type system, but the case studies I conducted indicate that most extant plain GADT programs are also well-typed in the Pointwise GADT type system. I attribute this phenomenon to the long-standing principle of orthogonal design, and I have shown that this principle is also useful for refactoring plain GADT programs into well-typed Pointwise GADT programs.

In the end, separating practical plain GADT programs from pathological ones is only a means to an end. The real purpose of the Pointwise GADT type system is to simplify the design of GADT type *inference* algorithms. I will describe how the Pointwise GADT type system fails to fulfill this promise in Chapter 5 — but only after showing you a further simplification of the plain GADT type system in the next chapter.

Chapter 4

THE NON-DEPENDENT GADT TYPE SYSTEM

The Pointwise GADT type system, which I introduced in Chapter 3, simplifies the GADT type inference problem by excluding certain pathological plain GADT programs that are not obviously well-typed (§3.1). This simplified type system has the advantage of accepting a wide variety of practical plain GADT programs. However, like the plain GADT type system, it is also extremely complicated, and it offers few clues about how one should attack the type inference problem.

This chapter introduces the *Non-Dependent* GADT type system, which further simplifies the type inference problem by eliminating GADT type refinements from the Pointwise GADT type system. Unlike the Pointwise GADT type system, the Non-Dependent GADT type system is no longer expressive enough to accept most practical plain GADT programs. However, by eliminating one major source of technical complexity (*i.e.,* GADT type refinements), the Non-Dependent GADT type system brings forward the relationship between existential types and GADT patterns, which leads to an important breakthrough in the type inference problem.

4.1 GADT WITHOUT TYPE REFINEMENTS

Recall that the plain GADT type system (§2.3) introduces two new features to the ADT type system. The first feature, GADT type arguments, supports non-uniform data constructor range types. The second feature, GADT type refinements, allows the types of pattern-matching branch bodies to vary with the type arguments of the branch pattern types.

Every program that uses GADT type refinements must also use GADT type arguments, because a uniform data constructor range type induces only a trivial GADT type refinement (which renames type variables). The converse, however, is not true: a program may use GADT type arguments alone without using GADT type refinements: it simply ignores all extra type information brought into scope by GADT patterns. I call a program that can be typed under this restriction a *Non-Dependent GADT* program, and I now present three examples.

4.1.1 Examples

My first example of a Non-Dependent GADT program is the function `inc1b` in Figure 4.1 (p. 94). The data types `PTa` and `PTb` both represent a pair of two types; they differ only in that `Pa` is an ADT constructor (for the `PTa` data type) with two distinct type variables `a`, `b` as its range type arguments, and that `Pb` is a GADT constructor (for the `PTb` data type) with a single type `(a, b)` as its range type argument. The functions `inc1a` and `inc1b` are, for all practical purposes, equivalent: they both return one plus the first component of a pair. However, due to differences between `PTa` and `PTb`, `inc1a` is an ADT function, and `inc1b` is a Non-Dependent GADT function because its `Pb` pattern induces only a trivial GADT type refinement.

My second example of a Non-Dependent GADT program is the function `tail` in Figure 4.1. This function, which uses the length-indexed list data type that I introduced in Figure 3.1 (p. 58), returns the tail of a non-empty list. In contrast with the typical `tail` function,[1] which diverges (*i.e.,* fails) when applied to the empty list, this function always succeeds: its argument type (`L (S k) a`) guarantees that a programmer cannot apply `tail` to the empty list (which has type `L Z a`). It is a Non-Dependent GADT function because the `Cons` pattern of its sole

[1] For example, the one implemented in the standard prelude of the Haskell language.

```
data PTa a b where
  Pa :: forall a b. a → b → PTa a b

data PTb a where
  Pb :: forall a b. a → b → PTb (a, b)

inc1a :: forall n. PTa Int n → Int
inc1a e = case e of
  Pa x y → x+1

inc1b :: forall n. PTb (Int, n) → Int
inc1b e = case e of
  Pb x y → x+1


tail :: forall k a. L (S k) a → L k a
tail xs = case xs of
  Cons y ys → ys


null :: forall m b. L m b → Bool
null xs = case xs of
  Nil → True
  Cons y ys → False
```

Figure 4.1: This figure shows three Non-Dependent GADT programs: `inc1b`, `tail`, and `null`. The latter two programs (`tail` and `null`) use the length-indexed list data type that I introduced in Figure 3.1 (p. 58).

branch induces only a trivial GADT type refinement.

My third example of a Non-Dependent GADT program is the function `null` in Figure 4.1. This function, which also uses the length-indexed list data type from Figure 3.1, checks if a list is empty. The type of the `case` scrutinee `xs` is `L m a`, and both patterns in the `case` expression induce nontrivial GADT type refinements: [`Z/m`] for the `Nil` pattern, and [`S n/m`] for the `Cons` pattern. However, since the body of neither branch uses the extra type information provided by the GADT type refinements, `null` remains a Non-Dependent GADT function.

These examples showcase the wide variety of Non-Dependent GADT programs. In particular, they include functions whose types constrain the shape of their input arguments (such as `tail`) and predicates on GADT values (such as `null`), both of which are common in GADT programs. Since a pattern-matching branch cannot take advantage of GADT type refinements in the Non-Dependent GADT type system, the type inference problem should, in principle, become much simpler. In the next subsection, I examine how existing type inference algorithms perform on these Non-Dependent GADT program.

4.1.2 Type inference testing

A paper on GADT type inference is incomplete without a paragraph or two explaining why the problem is so hard. And, without exception, the explanations and the accompanying code examples focus on GADT type refinements [17, 32, 36, 40, 43]. For example, Schrijvers et al. wrote [36, §9.2]:

> *Type inference for unannotated programs turns out to be extremely hard. The difficulty lies in the fact that GADT pattern matches bring into scope local type assumptions.*

Since type inference for the ADT type system (sans polymorphic recursion) has been completely solved [27], implicit in these explanations is the consensus that

GADT type refinements are the only significant feature that distinguishes GADT programs from ADT programs.

From the consensus, Non-Dependent GADT programs (which do not require GADT type refinements to type check) should be similar to ADT programs, and type inference for Non-Dependent GADT programs should not be too difficult. Alas, it turns out that all existing GADT type inference algorithms are *incomplete* for Non-Dependent GADT functions without type annotations. This result indicates that the current wisdom, which states that the difficulty of GADT type inference arises solely from GADT type refinements, is wrong, and there is more to the story than meets the eye. In this subsection, I will explain what happens behind the scenes. Existing GADT type inference algorithms roughly fall into two categories, and I will start by describing how algorithms in each category perform when applied to Non-Dependent GADT functions.

Scrutinee-major algorithms These type inference algorithms compute the scrutinee type of a pattern-matching branch differently depending on the shape of the pattern type. If the pattern type is uniform (*i.e.,* its type arguments are distinct type variables), these algorithms proceed using the standard type inference algorithm for ADT pattern-matching branches. If the pattern type is non-uniform, the algorithms consider the branch a GADT pattern-matching branch, and they compute the scrutinee type by inferring the type of the `case` scrutinee (possibly assisted by type annotations). Previous work in this category includes wobbly types [17], stratified type inference [32], the OutsideIn algorithm [36], and type inference via Herbrand constraint abduction [43].

Scrutinee-major algorithms require type annotations for the functions `inc1b` and `tail`. Inferring the scrutinee types for these two functions require using type information from the GADT pattern-matching branches, which scrutinee-major algorithms are inherently incapable of doing. Note that scrutinee-major algorithms can infer the type of `inc1a` but not the type of `inc1b`: since these algorithms use a syntactic criterion to distinguish GADT patterns from ADT patterns, they are unable to recognize (or to take advantage of) the ADT-like characteristic of the `inc1b` function.

Pattern-major algorithms Pattern-major algorithms infer the scrutinee type of a pattern-matching branch differently depending on whether a type annotation is available. If there is a type annotation, these algorithms use the annotation to compute the type of the `case` scrutinee as the scrutinee type of the branch. When there is no type annotation, they compute the scrutinee type of a pattern-matching branch from the pattern-type of the branch. If a `case` expression has multiple pattern-matching branches, these algorithms require that the type of the `case` scrutinee must be a common instance of all pattern types. The type inference algorithm in the latest Omega language [38] interpreter (Version 1.4.3) appears to be the only previous work in this category.

Pattern-major algorithms require a type annotation for the `null` function because, without it, they would infer both `L Z a` (the type of the pattern `Nil`) and `L (S n) a` (the type of the pattern `Cons y ys`) as the type of the `case` scrutinee `xs`, which then leads to type inference failure because these two types are not unifiable. Type inference for `inc1b` and `tail` succeeds without type annotations.

Discussion Existing GADT type inference algorithms perform poorly at inferring the types of Non-Dependent GADT functions. Different algorithms fail in different ways, and none of the algorithms can infer the types of all three Non-Dependent GADT functions (`inc1b`, `tail`, and `null`) from Figure 4.1 (p. 94). These failures indicate that Non-Dependent GADT programs are significantly different from ADT programs. Contrary to the consensus, GADT type refinements are not the only significant feature that distinguishes GADT programs from ADT programs. What, then, is the unacknowledged feature that distinguishes Non-Dependent GADT programs from ADT programs?

The answer is the use of type indices: Non-Dependent GADT programs allow programmers to choose between parametric instantiation and type indexing; ADT programs do not have this feature because they do not support type indexing. Recall from §3.1 that *parametric instantiation* refers to pointwise type information flow from the scrutinee type to the pattern type, and *type indexing* refers to pointwise type information flow from the pattern type to the scrutinee type. Type information flows from the specific to the generic, so parametric instantiation occurs when the scrutinee type is more specific than the pattern type, and type indexing occurs when the converse is true.

Deciding the specificity of a scrutinee type is a nontrivial task because, even for Non-Dependent GADT programs, there is no single best answer. Making the scrutinee type more specific (pattern-major algorithms take this idea to the extreme) facilitates parametric instantiation, which is essential for the `inc1b` function. In contrast, making the scrutinee type more general (scrutinee-major algorithms take this idea to the extreme) allows it to be consistent (*i.e.,* unifiable) with different pattern types, which is essential for the `null` function. To support Non-Dependent GADT programs, a type inference algorithm must make sensible choices on the appropriate specificity of each scrutinee type in the program. I will continue this discussion in §4.2 after introducing the Non-Dependent GADT type system.

4.1.3 Type system

The Non-Dependent GADT type system is nearly identical to the Pointwise GADT type system (§3.2): they are built on the same programming language and place identical restrictions on data constructor types. Figure 4.2 (p. 99) shows the type rules of the Non-Dependent GADT type system, which are identical to the Pointwise GADT type system (Figure 3.3, p. 80) except that here the ALT-NONDEP

VAR

$$\dfrac{\mathrm{x} : \forall\overline{\alpha}.\, t \in \Gamma \qquad s = \mathrm{inst}[\overline{\alpha}](t)}{\Gamma \vdash \mathrm{x} : s}$$

LAM

$$\dfrac{\Gamma\{\mathrm{u} : s\} \vdash \mathrm{e} : t}{\Gamma \vdash \lambda \mathrm{u}\,.\,\mathrm{e} : s \rightarrow t}$$

CONS

$$\dfrac{\mathrm{C} : \forall\overline{\alpha}.\, t \qquad s = \mathrm{inst}[\overline{\alpha}](t)}{\Gamma \vdash \mathrm{C} : s}$$

APP

$$\dfrac{\Gamma \vdash \mathrm{f} : t_1 \rightarrow t_2 \qquad \Gamma \vdash \mathrm{e} : t_1}{\Gamma \vdash \mathrm{f}\ \mathrm{e} : t_2}$$

LETREC-P

$$\dfrac{\Gamma\{\mathrm{u} : \forall\overline{\alpha}.\, s\} \vdash \mathrm{e} : s \qquad \overline{\alpha} \# \mathit{tyvar}(\Gamma) \qquad \Gamma\{\mathrm{u} : \forall\overline{\alpha}.\, s\} \vdash \mathrm{d} : t}{\Gamma \vdash \texttt{let}\ \mathrm{u} = \mathrm{e}\ \texttt{in}\ \mathrm{d} : t}$$

CASE

$$\dfrac{\Gamma \vdash \mathrm{e} : s \qquad \Gamma \vdash_p \mathrm{p}_i \rightarrow \mathrm{c}_i : s \rightarrow t}{\Gamma \vdash \texttt{case}\ \mathrm{e}\ \texttt{of}\ \{\, \overline{\mathrm{p}_i \rightarrow \mathrm{c}_i}\, \} : t}$$

ALT-NONDEP

$$\dfrac{\begin{array}{c}\mathrm{C} : \forall\overline{\alpha}.\, \overline{w} \rightarrow T\ \overline{s} \qquad \overline{\alpha} \# \mathit{tyvar}(\Gamma, \overline{u}, t) \\ \theta = \mathit{pwu}(T\ \overline{u} \sim T\ \overline{s}) \qquad \rho = \mathit{SK}_{\mathrm{dom}(\theta)} \qquad \rho(\Gamma)\{\overline{\mathrm{x} : \theta(w)}\} \vdash \mathrm{c} : \rho(t)\end{array}}{\Gamma \vdash_p \mathrm{C}\ \overline{\mathrm{x}} \rightarrow \mathrm{c} : T\ \overline{u} \rightarrow t}$$

$$\mathit{SK}_S(\alpha) = [\overline{\mathbb{X}_\alpha/\alpha}] \quad \text{for all } \alpha \in S$$

$$\mathrm{inst}[\overline{\alpha}](t) = \theta(t), \text{ where } \theta = [\overline{s/\alpha}], \text{ and } \overline{s} \text{ are arbitrary types}$$

Figure 4.2: The Non-Dependent GADT type system.

type rule replaces the ALT-PTWISE type rule. The Non-Dependent GADT type system is a minimal adaptation of the Pointwise GADT type system that removes GADT type refinements from GADT pattern-matching branches.

In contrast with the Pointwise GADT type system, the Non-Dependent GADT type system applies the pointwise unifier (θ) of the scrutinee type ($T\ \overline{u}$) and the pattern type ($T\ \overline{s}$) only to the argument types ($\overline{w}$) of the matched constructor (C). This mechanism is necessary to compute the types ($\theta(\overline{w})$) of the variables ($\overline{\mathrm{x}}$) bound in the pattern (C $\overline{\mathrm{x}}$). Instead of applying the same unifier to the outer type environment (Γ) and the type (t) of the `case` expression, the ALT-NONDEP applies a Skolemization substitution (ρ) that replaces each type variable α in the domain of the unifier with a unique type constant $\mathbb{X}_\alpha$. Each type constant $\mathbb{X}_\alpha$ is uninhabited and not equal to any other type, so the Skolemization mechanism ensures that the branch body (c) cannot take advantage of any local type information that would have been introduced by the pointwise unifier θ.

The functions `inc1b`, `tail`, and `null` (Figure 4.1, p. 94) are well-typed in the Non-Dependent GADT type system. The function `eval` (Figure 2.6, p. 34) is *not* well-typed in the Non-Dependent GADT type system because it uses GADT type refinements to type its return value (which has type `a`). The `repId` function (Figure 2.8, p. 41) is well-typed in both the Pointwise and the Non-Dependent GADT type systems. However, while `repId` has infinitely many maximal types in the Pointwise GADT type system, it has only one maximal type (which is also its principal type) in the Non-Dependent GADT type system:

```
repId :: forall a b. Term a → [b] → [b]
```

All other maximal types of `repId`, including those shown in Figure 2.8 (p. 41), are invalid because the ALT-NONDEP type rule instantiates the type variable `a` to $\mathbb{X}_{\mathtt{a}}$ (the Skolem type constant that corresponds to `a`) instead of to `Int`.

Since Skolemization affects type indexing but not parametric instantiation, the

ALT-NONDEP type rule accepts all type judgments that the ALT-ADT type rule does. Also, since Skolem type constants carry no type information whatsoever, it is always safe to replace a Skolem type constant with any type, so the ALT-PTWISE type rule accepts all type judgments that the ALT-NONDEP type rule does. In other words, every expression well-typed in the ADT type system is also well-typed in the Non-Dependent GADT type system, and every expression well-typed in the Non-Dependent GADT type system is also well-typed (and in fact may have more types) in the Pointwise GADT type system.

4.2 GENERALIZING EXISTENTIAL TYPES

Recall from §2.5 that every GADT pattern-matching branch has six associated type roles (Figure 2.9): Pattern, Declared, Instantiated, Refined, Scrutinee, and Outer. Given the first four type roles, it is the responsibility of a GADT type inference algorithm to infer types in the remaining two roles: Scrutinee, and Outer. In the plain and the Pointwise GADT type system, Refined and Outer are related by the GADT type refinement of the pattern-matching branch (§2.5). Since the Non-Dependent GADT type system does not support GADT type refinements, here Refined and Outer become, for all practical purposes, equivalent. In other words, type inference for the Outer type role is trivial. I therefore turn my attention to the problem of inferring the scrutinee type.

Existing type inference algorithms use one of two strategies for inferring the scrutinee type of a pattern-matching branch, and the main point of contention is the specificity of the scrutinee type. Scrutinee-major algorithms say that patterns should never contribute type information to the scrutinee type. Pattern-major algorithms, in contrast, say that patterns should contribute all their type information to the scrutinee type (unless the programmer says otherwise). Neither approach is ideal; the discussion in §4.1 shows how one fails exactly where the other succeeds. I therefore propose the following compromise: *a pattern should contribute only as* *much type information as it must to ensure that the branch is well-typed.*

How much type information must a pattern contribute to the scrutinee type? The answer to this question turns out to have deep connections with existentially quantified type variables in GADT patterns. In this section, I will introduce the notion of *generalized existential types* and use this new idea to develop an algorithm for inferring the scrutinee types of Non-Dependent GADT pattern-matching branches.

4.2.1 Existential types

In this subsection, I present background information on existential types in the ADT type system. The ideas in this subsection were first proposed by Laüfer and Odersky [22], and they serve as the foundation for my proposal of generalized existential types (to be introduced in the next subsection).

An *existential type* is a locally-quantified type variable in the type of a data constructor. When a constructor is used as an expression (to *construct* a value), an existential type behaves like a regular type variable. When a constructor is used as a pattern (to *destruct* a value), the pattern-matching branch must treat an existential type as an unknown type. More concretely, the branch body must not make any assumptions about the existential type (*i.e.,* no instantiation), and types outside the pattern-matching branch must not depend on the existential type (*i.e.,* no escape). These restrictions are necessary because the type of a pattern-matching branch provides no type information about the existential types introduced by the pattern, so any assumption on existential types can potentially compromise type safety.

In the ADT type system, an existential type is a type variable that appears in only the argument types (but not in the range type) of a data constructor. By this characterization, existential types are an intrinsic property[2] of a data constructor.

[2] A property of a thing is *intrinsic* if it depends only on the thing itself, and it is *extrinsic*

Here is an example of a data constructor with an existential type:

```
data AppT a where
  App :: forall a b. (b → a) → b → AppT a

app :: forall a. AppT a → a
app e = case e of
  App f x → f x
```

The type variable `b` in the type of `App` is an existential type because it does not appear in the range type `AppT a` of the `App` data constructor. The `App` pattern in the `app` function brings the type `b` into scope, and the pattern-matching branch obeys the aforementioned “no instantiation” and “no escape” restrictions. Here are two *negative* examples, each of which violates one of the two restrictions:

```
appInt :: forall a. AppT a → a
appInt e = case e of
  App f x → f 3       -- ill-typed due to instantiation

arg :: forall a b. AppT a → b
arg e = case e of
  App f x → x         -- ill-typed due to escape
```

The `appInt` function is not well-typed because its `App` branch body assumes that the existential type `b` is `Int`. The `arg` function is not well-typed because its result type depends on (*is*) the existential type `b` introduced by the `App` pattern.

4.2.2 Generalized existential types

The characterization of existential types for the ADT type system (reviewed in the previous subsection) is inadequate for the Non-Dependent GADT type system.

if it depends on a relationship between a thing and other things [49]. For example, mass is an intrinsic property of a physical object, but weight is an extrinsic property of a physical object.

The ADT characterization is certainly not wrong: a well-typed Non-Dependent GADT program should still adhere to the "no instantiation" and "no escape" restrictions for pattern-bound type variables that do not appear in the pattern type. The ADT characterization is, however, *inadequate*: there are sometimes other type variables that should also be subject to the same two restrictions. Existential types in the Non-Dependent GADT type system play an important role in deciding the appropriate specificity of scrutinee types, so in this subsection I will introduce a complete characterization of existential types in the Non-Dependent GADT type system. For the lack of a better name, I shall call it *generalized existential types*.

Definition 6. Consider the ALT-NONDEP rule in the Non-Dependent GADT type system (reproduced from Figure 4.2, p. 99):

ALT-NONDEP

$$\frac{\begin{array}{c} \mathrm{C} : \forall\overline{\alpha}.\, \overline{w} \to T\ \overline{s} \qquad \overline{\alpha} \,\#\, \mathit{tyvar}(\Gamma, \overline{u}, t) \\ \theta = \mathit{pwu}(T\ \overline{u} \sim T\ \overline{s}) \qquad \rho = \mathit{SK}_{\mathrm{dom}(\theta)} \qquad \rho(\Gamma)\{\overline{\mathrm{x} : \theta(w)}\} \vdash \mathrm{c} : \rho(t) \end{array}}{\Gamma \vdash_p \mathrm{C}\ \overline{\mathrm{x}} \to \mathrm{c} : T\ \overline{u} \to t}$$

Given a pattern-matching branch $\mathrm{C}\ \overline{\mathrm{x}} \to \mathrm{c}$ whose typing is described by the ALT-NONDEP rule, I define the *generalized existential types* of the pattern $\mathrm{C}\ \overline{\mathrm{x}}$ as the set of type variables $(\overline{\alpha} \setminus \bigcup_\theta \mathrm{dom}(\theta))$ where the set union ranges over all valid choices of θ in the ALT-NONDEP type rule. ✻

Let us start with an example on how to apply this definition. Here is the `null` function reproduced from Figure 4.1 (p. 94):

```
data L n a where
  Nil  :: forall a. L Z a
  Cons :: forall a n. a → L n a → L (S n) a

null :: forall m b. L m b → Bool
null xs = case xs of
```

```
Nil → True
Cons y ys → False
```

What are the generalized existential types in the `Cons` branch? Its scrutinee type (`L m b`) and its pattern type (`L (S n) a`) have two pointwise unifiers (they differ only in the orientation of renaming), and both are valid choices of θ:

$$\theta_1 = [\mathtt{S\ n/m,\ a/b}] \quad \text{and} \quad \theta_2 = [\mathtt{S\ n/m,\ b/a}]$$

Here $\overline{\alpha} = \{\mathtt{n}, \mathtt{a}\}$, $\mathrm{dom}(\theta_1) = \{\mathtt{m}, \mathtt{b}\}$, $\mathrm{dom}(\theta_2) = \{\mathtt{m}, \mathtt{a}\}$, so the type variable `n` is the only generalized existential type in the `Cons` branch. A similar derivation should show that the `Nil` branch has no generalized existential types.

Generalized existential types are well-defined for all GADT pattern-matching branches that are accepted by the ALT-NONDEP type rule. Pointwise unifiers, like most-general unifiers, are unique up to the orientation of variable renaming. Therefore, an algorithm can always enumerate all valid choices of θ because the set of valid choices is always finite.

Generalized existential types, like existential types, are not an addition to an existing type system. Instead, they describe a property, or, in other words, a logical consequence that follows from the definition of the type system. Definition 6 does not describe a new type system feature — the feature was put into place when I introduced the ALT-NONDEP type rule in Figure 4.2 (p. 99). Some may even say that generalized existential types came into existence when the plain GADT type system was first proposed in previous work — only that no one discovered them until now.

The generalized existential types of a pattern are all the pattern-bound type variables that should not escape or be instantiated. Here is an informal argument to support this statement:

- If a pattern-bound type variable γ is *not* a generalized existential type, then $\gamma \in \mathrm{dom}(\theta)$ for a valid θ. If $\theta(\gamma)$ is a type variable, then γ escapes without causing a type error. If $\theta(\gamma)$ is not a type variable, then γ is instantiated without causing a type error. In either case there is no need to restrict γ from escape or instantiation.

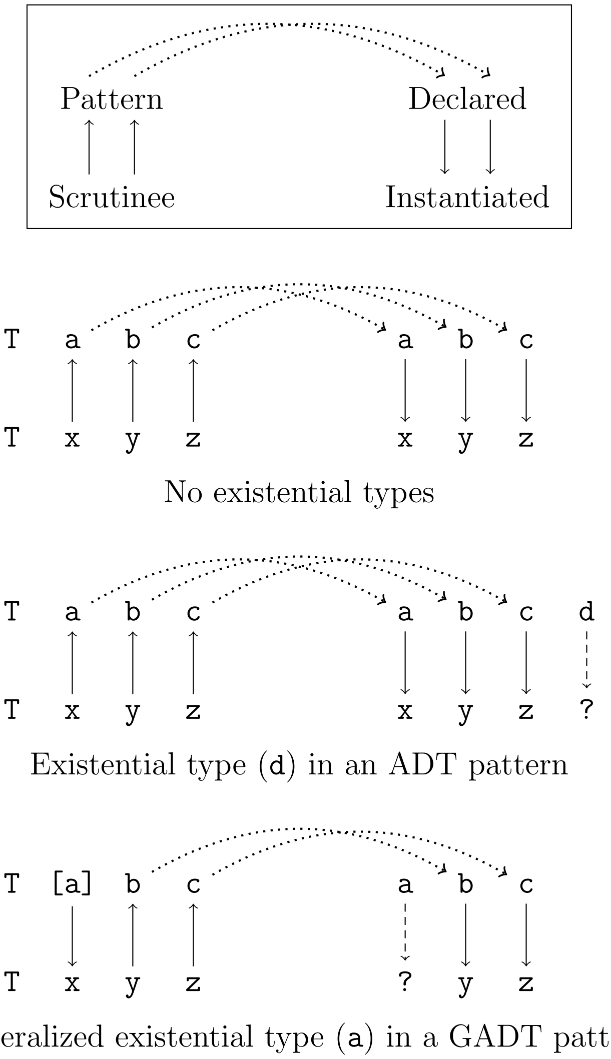
- If γ is a generalized existential type, then $\gamma \in \overline{\alpha}$ and $\gamma \notin \mathrm{dom}(\theta)$ for all valid θ. Since $\gamma \notin \mathrm{dom}(\theta)$, the pattern-matching branch body must not make any assumptions about γ (no instantiation). Since $\gamma \in \overline{\alpha}$ and $\overline{\alpha} \mathbin{\#} \mathit{tyvar}(\Gamma, \overline{u}, t)$, types outside the scope of the pattern-matching branch must not depend on γ (no escape).

Generalized existential types are a conservative extension to existential types. More specifically, the generalized existential types of an ADT pattern are identical to the existential types of the ADT data constructor. Let C in the ALT-NONDEP rule be an ADT constructor, then $\overline{s}$ must be a sequence of distinct type variables. Since $[\overline{u/s}]$ is a valid choice of θ, and $(\mathrm{dom}(\theta) \cap \overline{\alpha}) \subseteq \overline{s}$ for all valid choices of θ, the generalized existential types of the pattern C $\overline{\mathrm{x}}$ must be $(\overline{\alpha} \setminus \overline{s})$, which matches the existential types of the data constructor C.

Figure 4.3 (p. 107) shows why this conservative extension is necessary in the Non-Dependent GADT type system. The main part of this figure contains three diagrams, each of which illustrates one kind of interaction between the Pattern Matching and the Variable Binding type roles (§2.5) in a GADT pattern-matching branch. For example, the middle diagram is an abstract representation of type interactions in the following pattern-matching branch:

```
data T a b b where
  C :: forall a b c d. a → b → c → d → T a b c

f :: forall x y z. T x y z → ...
f e = case e of
        C p q r s → ...
```



No existential types

Existential type (d) in an ADT pattern

Generalized existential type (a) in a GADT pattern

Figure 4.3: This figure illustrates existential types and generalized existential types. Existential types appear when the unifier of the scrutinee type and the pattern type provides no information (represented by dotted arrows) about a type variable introduced by the pattern. In an ADT pattern, this situation happens only for type variables that do not appear in the pattern type (middle diagram). In a GADT pattern, it can happen even for type variables that do appear in the pattern type (bottom diagram). Thus the characterization of existential types is no longer adequate in GADT type systems.

The range type of the data constructor (`T a b c`) appears at top-left of the diagram, the argument types (`a b c d`) of the data constructor appear at top-right, the type (`T x y z`) of the scrutinee (`e`) appears at bottom-left, and the types (`x y z ?`) of pattern-bound variables (`p q r s`) appear at bottom-right. Please see §2.5 for a detailed description of type roles in a GADT pattern-matching branch. I use the following symbols in the diagrams:

- Solid vertical arrows in the left half of a diagram denote type information flow between the scrutinee type and the pattern type of a branch. Up arrows represent parametric instantiation; down arrows represent type indexing. Together the arrows represent a most-general unifier θ between the scrutinee type and the pattern type. All type variables that appear in the domain of θ have solid vertical arrows pointing to them.

- Dotted curved arrows that cross from the left half of a diagram to the right half indicate how type systems transfer parametric instantiation type information from the Pattern type role to the Declared type role. These dotted arrows link pattern type variables at the end of solid up arrows to the same type variables in the argument types of the matched data constructor.

- Solid down arrows in the right half of a diagram represent the instantiation from the Declared type role to the Instantiated type role. They link data constructor type variables in the domain of θ (which all have dotted arrows pointing to them) to the (instantiated) types of pattern-bound variables.

- Dashed down arrows in the right half of a diagram represent unknown types of pattern-bound variables that correspond to generalized existential types. They all originate from type variables that have *no* dotted arrows pointing to them, which reflects the intuition that generalized existential types are type variables introduced by a pattern that receive no parametric instantiation type information from the scrutinee type.

In the ADT type system, pattern types are always uniform (*i.e.,* the type arguments of a pattern type are always distinct type variables), so every type variable that appears in the pattern type can obtain type information from the scrutinee type (top diagram). The only pattern-introduced type variables that cannot obtain type information from the scrutinee type are those that do not appear in the pattern type (*e.g.,* `d` in the middle diagram), so the characterization of existential types makes sense in the ADT type system. In the Non-Dependent GADT type system, in contrast, even a type variable that appears in the pattern type (*e.g.,* `a` in the bottom diagram) may be unable to obtain type information from the scrutinee type (*e.g.,* due to the scrutinee type being less specific than the pattern type), so the traditional characterization of existential types is no longer adequate, and a generalization becomes necessary to capture these new kinds of existential types.

Generalized existential types differ from existential types in two ways:

1. Generalized existential types are a property of patterns, not a property of data constructors. This observation explains why existential types require special treatment only when a data constructor is used as a pattern, but not when the same constructor is used as an expression.

2. Generalized existential types are an extrinsic property instead of an intrinsic property. Unlike an intrinsic property, which is inherent to a thing itself, an extrinsic property also depends on the context of the thing. Existential types are intrinsic because they depend only on the type of the data constructor. Generalized existential types are extrinsic because they depend on the scrutinee type ($T\ \overline{u}$) from the context.

The second point is particularly noteworthy because it suggests that a type inference algorithm has some influence over the generalized existential types of a pattern. Existing type inference algorithms support existential types by checking against their instantiation and escape in each pattern-matching branch. This approach, however, is insufficient for generalized existential types because they are extrinsic and thus cannot be identified a priori. To support generalized existential types, a type inference algorithm must work backwards: it must first identify instantiated and escaped type variables in a pattern, and then adjust the specificity of the scrutinee type so that those instantiated or escaped type variables become regular (as opposed to generalized existential) types.

Given a Non-Dependent GADT pattern-matching branch whose typing is as described in Definition 6 (p. 104), a type variable γ introduced by the pattern (C $\overline{\mathrm{x}}$) is a regular type if and only if it satisfies the following conditions:

1. $\gamma \in \mathit{tyvar}(T\ \overline{s})$,
2. Every occurrence of the type variable γ in the pattern type $(T\ \overline{s})$ has a pointwise counterpart in the scrutinee type $(T\ \overline{u})$, and
3. All pointwise counterparts of γ in the scrutinee type $(T\ \overline{u})$ are identical.

These conditions come from the definition of pointwise unifiers (Definition 2). Intuitively, a type variable in a pattern type is a regular type if the scrutinee type provides complete and consistent type information at the corresponding positions. If a type variable introduced by a pattern does not appear in the pattern type, or if the scrutinee type provides only incomplete or inconsistent type information at positions corresponding to the occurrences of the type variable, then it must be considered a generalized existential type.

Examples In the Non-Dependent GADT type system, generalized existential types are preferable to existential types because they offer greater explanatory power. I illustrate this point with two examples. My first example is the `inc1b` function (reproduced from Figure 4.1, p. 94)

```
data PTb a where
  Pb :: forall a b. a → b → PTb (a, b)

inc1b :: forall n. PTb (Int, n) → Int
inc1b e = case e of
  Pb x y → x+1
```

Why is `PTb u → Int` *not* a valid type of `inc1b`? The notion of existential types provides no help because `Pb` has no existential types. Generalized existential types, in contrast, offer a simple explanation. The type variable `a` in the pattern type `PTb (a, b)` does not have a pointwise counterpart in the scrutinee type `PTb u`, so `a` must be a generalized existential type. Since the branch body requires instantiating the type `a` of `x` to `Int`, the type system must reject `inc1b` under the type `PTb u → Int`.

My second example is the `tail` function (reproduced from Figure 4.1, p. 94):

```
data L n a where
  Cons :: forall a n. a → L n a → L (S n) a

tail :: forall k a. L (S k) a → L k a
tail xs = case xs of
  Cons y ys → ys
```

Why is `L m a → L k a` *not* a valid type of `tail`? The notion of existential types (again) provides no help, and generalized existential types (again) offer an explanation. The type variable `n` in the pattern type `L (S n) a1` does not have a pointwise counterpart in the scrutinee type `L m a`, so `n` must be a generalized existential type. Since `n` in the type `L n a1` of `ys` escapes from the branch body as the type `k` in the context, the type system must reject `tail` under the type `L m a → L k a`.

There is a general principle at work here. Earlier I stated that the key to Non-Dependent GADT type inference is to infer a scrutinee type that is neither too specific nor too general. It is easy to see how a scrutinee type can be too specific: it is too specific when it makes the scrutinee type inconsistent (*i.e.,* not unifiable) with a pattern type. The examples I presented here provide the other side of the story: how a scrutinee type could be too general. A scrutinee type is too general when it fails to provide consistent type information about instantiated and escaped pattern type variables (and thus forces the instantiated/escaped type variable to become a generalized existential type). I will use this insight to develop a scrutinee type inference algorithm in the next subsection.

Other type systems So far, I have described generalized existential types only in the context of the Non-Dependent GADT type system, but they are not specific to this type system: Definition 6 is equally applicable to the Pointwise and the plain GADT type systems. Even though the characterization is identical, generalized existential types in different type systems may have different properties.

For example, in the Pointwise and the Non-Dependent GADT type systems, a pattern type variable that has no pointwise counterpart in the scrutinee type must be considered a generalized existential type. This property, however, does not hold in the plain GADT type system. Consider the `joint` function (reproduced from Figure 3.4, p. 86) in the plain GADT type system:

```
data Split a b where
  Whole :: Split Int Int
  Parts :: forall a b. Split (Int, a) (b, Bool)

joint :: forall x. Split x x → x
joint e = case e of
  Whole → 7
  Parts → (3, True)
```

The type variable `a` in the `Parts` pattern type `Split (Int, a) (b, Bool)` does not have a pointwise counterpart in the scrutinee type `Split x x`, but `a` is *not* a generalized existential type because θ, which unifies the pattern type with the scrutinee type, maps `a` to `Bool`. It is more difficult to characterize generalized existential types in the plain GADT type system (than it is in the Pointwise or the Non-Dependent GADT type system) because it does not regulate the type information flow between the pattern type and the scrutinee type.

4.2.3 Inferring scrutinee types

In the Non-Dependent GADT type system, generalized existential types arise in a pattern-matching branch when the scrutinee type lacks sufficient information to instantiate all type variables that are introduced by the pattern. In other words, the more detailed the scrutinee type is, the fewer generalized existential types there will be in a pattern-matching branch. Turning this statement backwards suggests a strategy for inferring the scrutinee type from a pattern-matching branch: if a pattern-introduced type variable violates the no-instantiation/no-escape rule for generalized existential types, then a type inference algorithm must add sufficient detail to the scrutinee type so that the instantiated/escaped type variable is not considered a generalized existential type. In this subsection, I describe an algorithm that uses this strategy.

Figure 4.4 (p. 114) shows a type inference algorithm for Non-Dependent GADT pattern-matching branches. The algorithm infers the type of a pattern-matching branch by first inferring the type of the branch body (c) under a suitable type environment, and then deciding an appropriate level of specificity for the scrutinee type based on the type inference result for the branch body. (Due to the lack of GADT type refinements in the Non-Dependent GADT type system, inferring the branch body type is trivial — the body type of the branch is the same as the type of the branch body.) The algorithm computes a scrutinee type for the branch in three steps:

1. The first step is to compute the set $\overline{\gamma}$ of pattern-bound type variables that

```
$\mathit{infer}(\Gamma,\ \mathrm{C}\ \overline{\mathrm{x}} \rightarrow \mathrm{c}) =$
  let  $(\forall \overline{\alpha}.\, \overline{w} \rightarrow T\ \overline{s}) = \mathit{lookup}(\mathrm{C})$ where $\overline{\alpha}$ fresh
       $(\eta,\ t) = \mathit{infer}(\Gamma\{\overline{\mathrm{x} : w}\},\ \mathrm{c})$
       $\overline{\gamma} = \overline{\alpha} \cap (\mathrm{dom}(\eta) \cup \mathit{tyvar}(\mathrm{rng}(\eta)) \cup \mathit{tyvar}(t))$
  in  if $\overline{\gamma} \not\subseteq \mathit{tyvar}(\overline{s})$  then $\bot$
      else  if $\overline{\gamma} = \emptyset$
            then  $(\eta,\ T\ \overline{\mu} \rightarrow t)$ where $\overline{\mu}$ fresh
            else  $(\eta,\ \eta(\mathit{transcb}(\overline{\gamma},\ T\ \overline{s})) \rightarrow t)$

$\mathit{transcb}(\overline{\gamma},\ t) =$
  if $\overline{\gamma} \mathrel{\#} \mathit{tyvar}(t)$
  then  $\mu$ where $\mu$ fresh
  else  case t of
           $U\ \overline{w} \rightarrow U\ \langle \mathit{transcb}(\overline{\gamma},\ z) \mid z \in \overline{w} \rangle$
           $\nu \quad\ \rightarrow \nu$
```

Figure 4.4: This figure describes a type inference algorithm for Non-Dependent GADT pattern-matching branches. The notation $\langle f(x) \mid x \in \overline{w} \rangle$ means applying f to a sequence of types $\overline{w}$. The algorithm follows the basic structure of Milner's Algorithm $\mathcal{W}$ [27]: the result of type inference is an idempotent type substitution for the type environment and a type of the expression. Due to the lack of GADT type refinements, the body type of the branch is identical to the type of the branch body (t). The type variables in $\overline{\gamma}$ either escaped or were instantiated in the branch body, so the algorithm first checks that $\overline{\gamma}$ are not existential types, and it then transcribes $\overline{\gamma}$ from the pattern type ($T\ \overline{s}$) to the scrutinee type using *transcb* so that $\overline{\gamma}$ are not considered generalized existential types. The algorithm returns a failure symbol $\bot$ if the existential type checking fails.

either escaped or have been instantiated by the branch body. The algorithm computes $\overline{\gamma}$ as the intersection of pattern-bound type variables ($\overline{\alpha}$) with the union of instantiated ($\mathrm{dom}(\eta)$) and escaped ($\mathit{tyvar}(\mathrm{rng}(\eta)) \cup \mathit{tyvar}(t)$) type variables. This distinction of instantiated and escaped type variables is imprecise, but confusion between the two does not matter here because it does not affect their union.

2. The second step is to check that the set $\overline{\gamma}$ contains no existential types (or, equivalently, that every type variable in $\overline{\gamma}$ appears in the pattern type $T\ \overline{s}$). Type inference for the pattern-matching branch fails if the check does not succeed. This step enforces the first condition (p. 110) for $\overline{\gamma}$ to be regular (as opposed to generalized existential) types.

3. The third step is to transcribe all occurrences of $\overline{\gamma}$ in the pattern type ($T\ \overline{s}$) to the scrutinee type. The algorithm invokes *transcb* to compute a minimal (*i.e.,* least specific) copy of the pattern type that keeps all occurrences of $\overline{\gamma}$ in their original positions. In the special case where $\overline{\gamma}$ is empty (which makes *transcb* return a fresh type variable as result), the algorithm returns a fresh type $T\ \overline{\mu}$ as the scrutinee type. This step enforces the last two conditions (p. 110) for $\overline{\gamma}$ to be regular (as opposed to generalized existential) types.

Effective functioning of the algorithm requires that the type substitution η, returned from the type inference of the branch body (c), should not contain any superfluous type variable renaming. Superfluous type variable renaming in η may mislead the algorithm into (falsely) believing that a type variable escaped or has been instantiated, which may either cause existential type check failure, or make the inferred scrutinee type more specific than necessary.

The main part of the type inference algorithm, which infers the scrutinee type of the pattern-matching branch, is relatively isolated from the rest of the algorithm. It does not require access to the type environment (Γ), the argument types ($\overline{w}$) of the matched constructor, or the internal structure of the branch body (c). All it needs are the pattern type ($T\ \overline{s}$), the type variables ($\overline{\alpha}$) introduced by the pattern, the type variables in the inferred branch body type ($\mathit{tyvar}(t)$), and the inferred type substitution (η). The dependency on the type inference result of the branch body is particularly significant: it echoes my earlier observation that generalized existential types are an extrinsic property rather than an intrinsic one, and that type inference algorithms should play an active role in shaping the generalized existential types of a pattern matching branch by adjusting the specificity of the scrutinee type.

Examples I now conclude this subsection by running the algorithm through the three Non-Dependent GADT program examples in Figure 4.1 (p. 94). The first example is the `inc1b` function (reproduced here):

```
data PTb a where
  Pb :: forall a b. a → b → PTb (a, b)
inc1b e = case e of
  Pb x y → x+1
```

Inferring the type of the branch body (`x+1`) returns $t = \texttt{Int}$ and $\eta = [\texttt{Int/a}]$ as result. The algorithm computes $\overline{\gamma} = \{\texttt{a}\}$ as the only instantiated pattern type variable, the *transcb* invocation returns `PTb (a, u)`, and the algorithm computes $\eta(\texttt{PTb (a, u)}) = \texttt{PTb (Int, u)}$ as the scrutinee type of the branch. The second example is the `tail` function (reproduced here):

```
data L n a where
  Nil  :: forall a. L Z a
  Cons :: forall a n. a → L n a → L (S n) a
tail xs = case xs of
  Cons y ys → ys
```

Inferring the type of the branch body (`ys`) returns $t =$ `L n a` and $\eta = [\,]$ (*i.e.,* the identity substitution) as result. The algorithm computes $\overline{\gamma} = \{$`n`$\}$ as the only escaped pattern type variable, the *transcb* invocation returns `L (S n) b`, which the algorithm uses as the scrutinee type of the branch. The third example is the `null` function (reproduced here):

```
null xs = case xs of
  Nil → True
  Cons y ys → False
```

The two pattern-matching branches are structurally similar, so I will use only the `Nil` branch in my demonstration. Inferring the type of the branch body (`True`) returns $t =$ `Bool` and $\eta = [\,]$ (*i.e.,* the identity substitution) as result. Since there are no escaped or instantiated pattern type variables, the algorithm computes $\overline{\gamma} = \emptyset$ and returns `L m b` as the scrutinee type of the branch.

4.3 SUMMARY

In this chapter, I studied the nature of Non-Dependent GADT programs, which are GADT programs that do not require GADT type refinements. Even though most previous work on GADT type systems invariably focused on GADT type refinements, Non-Dependent GADT programs remain substantially different from ADT programs. I demonstrated that type-indexing plays an important role in Non-Dependent GADT programs, and that programmers must decide the appropriate specificity of each branch scrutinee type based on the context.

I proposed generalized existential types, which generalize existential types in the ADT type system. They have several uses: they relate scrutinee types to the types of pattern-bound variables, they describe the appropriate level of scrutinee type specificity in different contexts, and they suggest a strategy for inferring the types of Non-Dependent GADT pattern-matching branches.

The research I presented in this chapter began with an anomaly regarding existing GADT type inference algorithms (incompleteness for programs that do not require GADT type refinements). The investigation of this anomaly lead to the discovery of generalized existential types, which then culminated in a type inference algorithm for Non-Dependent GADT pattern-matching branches. Such cycles of research are a recurring theme in this dissertation: limitations of type inference algorithms motivates type system research, and deeper insight into the type system, in turn, leads to breakthroughs in the design of new type inference algorithms.

Encouraged by the nice properties of the Non-Dependent GADT type system (for example, even the `repId` function (Figure 2.8, p. 41) has a principal type), and armed with the type inference algorithm for Non-Dependent GADT pattern-matching branches (typically the most difficult subject for type inference), I set out to implement a Non-Dependent GADT type inference algorithm. I did end up with a prototype system that infers the types of all three examples in Figure 4.1 (p. 94), but the system fell (very) short of my expectations. The cycle of research renews, and I have the details in the next chapter.

Chapter 5

SIMPLIFICATIONS TO COMPLICATIONS

The focus of this chapter is on the GADT branch reachability requirement (§5.1), which is a fairly well-known feature that has some obscure consequences. I discuss how this feature evolves in the Pointwise GADT (§5.2) and in the Non-Dependent GADT (§5.3) type systems, and, where applicable, I also discuss how the various incarnations of this feature affect the designs of GADT type inference algorithms in general. These discussions, in contrast with the earlier chapters, explore a new area in the design space of GADT type systems and should (I hope) serve as a cautionary tale on the intricacies of type system design.

GADT type inference is a difficult problem. To make the problem tractable, I have proposed two simplifications to the plain GADT type system: using pointwise unifiers to combine the scrutinee type and the pattern type in a pattern-matching branch (Chapter 3), and eliminating GADT type refinements in GADT pattern-matching branches (Chapter 4). Do these simplifications actually make the type inference problem more tractable?

The answer is both *yes* and *no*. These two simplifications are essential to the Non-Dependent GADT pattern-matching branch type inference algorithm (§4.2), and presenting them as restricted versions of the plain GADT type system really clarifies the nature of these simplifications. There is, however, a hidden drawback: the type rules that enforce these simplifications interact badly with some common type inference algorithm designs. In this chapter, I explain why type inference for these simplified GADT type systems remains difficult, even if one knows how to infer the types of GADT `case` expressions.

5.1 GADT BRANCH REACHABILITY REQUIREMENT

In the ADT type system, every pattern-matching branch in a `case` expression has the same type. To the type system, every branch looks the same, so the type system is oblivious to the reachability (or the lack thereof) of a pattern-matching branch. A GADT type system, in contrast, allows the pattern types of different pattern-matching branches to have different types. This type system feature allows a GADT type system to identify, statically, some pattern-matching branches that are unreachable at runtime. If the scrutinee type and the pattern type of a branch are inconsistent, then a value that matches the pattern can never flow to the `case` scrutinee, so the branch must be unreachable. This ability to identify unreachable branches presents a design choice to GADT type system designers: should a GADT type system accept programs with unreachable branches, or should it reject such programs? In this section, I weigh the pros and cons of each choice and explore how the choices affect the designs of GADT type systems and GADT type inference algorithms.

5.1.1 GADT and branch reachability

The reachability of a pattern-matching branch indicates whether the branch may affect the behavior of the program. If the answer is yes, then the branch is *reachable*; otherwise the branch is *unreachable*. Reachability is an extrinsic property; the same branch may be reachable in one context and unreachable in another. In a GADT type system, a pattern-matching branch with inconsistent scrutinee type and pattern type must be unreachable. This criterion is sound (it never mistakes a reachable branch as unreachable) but incomplete (it may consider an unreachable branch as reachable).

Figure 5.1 (p. 121) shows an unreachable pattern-matching branch with the `drop` function. Since the type of `Nil` (*i.e.,* `L Z a`) is inconsistent with the type

```
drop :: forall k a. L (S k) a → L k a
drop xs = case xs of
  Nil → Nil
  Cons y ys → ys
```

Figure 5.1: This figure shows a GADT function that has an unreachable branch. The `drop` function, which uses the length-indexed list type (Figure 3.1, p. 58), computes the tail of a non-empty list. The `Nil` branch, which makes `drop` return the empty list unchanged, is unreachable under the given type. The unreachable branch makes `drop` ill-typed in the plain GADT type system.

---

of `xs` (*i.e.,* `L (S k) a`), the `Nil` value will never flow to the `case` scrutinee (`xs`), therefore the `Nil` branch will never be taken and is thus unreachable. Echoing my earlier statement that reachability is an extrinsic property, here the reachability of `Nil` depends on the type of the `case` scrutinee. If I had instead given `drop` the following type:

```
drop :: forall a. L Z a → L Z a
```

Under this type, the `Nil` branch would become reachable, but the `Cons` branch would become unreachable (because the type of `xs` requires that only the empty list can flow to the `case` scrutinee).

There are a few ways to check the consistency between the scrutinee type and the pattern type of a branch. A type system may check if they are unifiable [17, §4.3] or if their equality is satisfiable [36, §3.2] (which is basically the same thing). Regardless of the underlying mechanism, the important idea is that a GADT type system can decide, statically, that some pattern-matching branches will never be taken at runtime.

The ability to identify unreachable GADT pattern-matching branches forces a type system designer to make a design decision. A GADT type system can accept programs with unreachable branches, or, alternately, it can reject programs with unreachable branches. The first option is acceptable because unreachable branches are harmless. The second option is also acceptable because unreachable branches are useless. Nonetheless, this is not a decision to be taken lightly — as I will show in this section, both options have far-reaching consequences.

It is straightforward to incorporate either option into a GADT type system. The plain GADT type system (§2.3) is an example that adopts the second option. By requiring the scrutinee type and the pattern type of a branch to be unifiable, the ALT-GADT type rule ensures that only reachable branches may appear in a well-typed program. To adopt the first option (accepting programs with unreachable branches), I can simply add the following type rule, which also types GADT pattern-matching branches, to the plain GADT type system:

$$\text{ALT-FAIL}\quad \frac{\mathrm{C} : \forall\overline{\alpha}.\,\overline{w} \to T\ \overline{s} \qquad \overline{\alpha} \,\#\, \mathit{tyvar}(\Gamma, \overline{u}, t) \qquad (T\ \overline{u}) \text{ is not unifiable with } (T\ \overline{s})}{\Gamma \vdash_p \mathrm{C}\ \overline{\mathrm{x}} \to \mathrm{c} : T\ \overline{u} \to t}$$

The ALT-FAIL type rule, which I adapted from Vytiniotis et al. [15, Figure 4], states that any pattern-matching branch whose scrutinee type is inconsistent (*i.e.,* not unifiable) with the pattern type is also well-typed. Note that the branch body (c) is never mentioned in the requirement of the type judgment (the part of the type rule that is above the horizontal line). This design is driven by the following two considerations:

1. Since the branch is unreachable, it does not really matter what the branch body is. Nothing in the branch body can compromise type safety, so there really is no reason to check the branch body.

2. Since the scrutinee type is inconsistent with the pattern type, there is no way to build a combined type environment that is needed to check the branch body. In other words, there is no clear criterion on what the body of an unreachable branch should (or should not) look like, so the very concept of checking the branch body in ALT-FAIL is ambiguous.

Even though it is both easy and safe to build GADT type systems that accept unreachable branches, such type systems are undesirable (compared to the ones that reject unreachable branches) from a software engineering perspective. Here are a few reasons why accepting unreachable branches is a bad idea:

- Doing so makes the behavior of a function dependent on its type. The `drop` function (Figure 5.1, p. 121) demonstrates the problem: depending on what its type is, it is either an identity function of the empty list or a tail function of non-empty lists (but never both at the same time). This feature is likely to confuse programmers.

- Doing so may cover up programming errors. The following `map` function on length-indexed lists (Figure 3.1, p. 58) demonstrates the problem:

  ```
  map f xs = case xs of
    Nil → Nil
    Cons y ys → Cons (f y) (mpa f ys)
  ```

  There is a typo in the `Cons` branch: the `map` function is misspelled as `mpa`, which is not a name bound in the scope of the branch body. However, in a type system that accepts unreachable branches, this incorrect `map` function remains well-typed under the following type:

  ```
  map :: forall a b c. a → L Z b → L Z c
  ```

This type does not fix the typo; it merely covers up the programming error by making the Cons branch unreachable. This ability to cover up errors makes a type system less useful as a debugging tool.

- Doing so may cover up type annotation errors. I demonstrate this problem using the isNat function (Figure 3.1, p. 58), which has the following type:

  ```
  isNat :: forall m. L m Int → L m Bool
  ```

  What if a confused programmer, who is unfamiliar with the type argument ordering of L, gives it the following type instead?

  ```
  isNat :: forall m. L Int m → L Bool m
  ```

  A type system that accepts unreachable branches would happily accept this incorrect type because it makes both pattern-matching branches in isNat unreachable. This ability to ignore type annotation errors also makes a type system less useful as a debugging tool.

Perhaps due to these downsides, the second option (*i.e.,* rejecting unreachable branches) has now become a de facto standard for GADT type systems. To the best of my knowledge, the target type system of wobbly types [15, §3] and guarded algebraic data types [40] are the only previous work on GADT-like type systems that accept programs with unreachable branches. All other previous work, as well as all GADT type systems that I presented in this dissertation, adopt the more practically useful option of enforcing the reachability requirement of GADT pattern-matching branches.

5.1.2 Restricting reachability enforcement

In the previous subsection, I argued that requiring pattern-matching branches to be reachable is sensible from a software engineering point of view because programmers typically avoid writing dead code. This very requirement, however, is also *absurd* from the language semantics point of view because the whole point of a `case` expression is that only one branch gets taken. Despite all the things I said in the last subsection, unreachable branches are an essential part of the programming language, and their complete elimination would make `case` expressions a much less useful tool for practical programming.

All GADT type systems in existence resolve this technical contradiction (that they must both accept and reject unreachable branches) by using `let` expressions to delimit the effects of the branch reachability requirement. More specifically, in a GADT type system, a local `let` definition defines a scope for the branch reachability requirement, and the type system requires only that a pattern-matching branch appears reachable in the limited context of the local `let` definition. As I mentioned earlier, branch reachability is extrinsic (*i.e.,* context-dependent), so restricting the scope of the context in which a branch must appear reachable can significantly affect the enforcement of the branch reachability requirement.

The expression `e1` in Figure 5.2 (p. 126) demonstrates this mechanism. Since the locally-defined `null` function is invoked only once with `Nil` as its argument, the `Cons` branch is clearly unreachable in the context of the entire `e1` expression. In the restricted context of the locally-defined `null` function, however, both the `Nil` and the `Cons` branches appear reachable because `null` could (hypothetically) be invoked with a list of any length as argument. Since the plain GADT type system enforces the branch reachability requirement only in the context of the local `let` definition, it accepts `e1` as well-typed. In contrast, consider the expression `e3`, which defines the same computation as `e1` but expresses `null` as an anonymous function instead of through a local `let` definition. Without the local `let` definition, the plain GADT type system enforces the branch reachability requirement over the entire `e3` expression, so it must reject `e3` due to branch reachability violation.

```
e1 = let null :: forall m a. L m a → Bool
         null xs = case xs of
           Nil → True
           Cons y ys → False
     in null Nil

-- e2 is not well typed under the following type of null
e2 = let null :: forall a. L Z a → Bool
         null xs = case xs of
           Nil → True
           Cons y ys → False
     in null Nil

-- e3 is not well typed and has no valid type
e3 = (λxs. case xs of
             Nil → True
             Cons y ys → False) Nil
```

Figure 5.2: This figure demonstrates the consequences of the GADT branch reachability requirement. Expression `e1` is well-typed in the plain GADT type system, but `e2` and `e3` are not because they violate the branch reachability requirement. Since `e2` is identical to `e1` except that the type variable `m` is replaced by the type constructor `Z`, these two expressions show that the plain GADT type system is not substitutive. Since taking one reduction step from `e1` results in `e3`, these two expressions show that the plain GADT type system does not satisfy type preservation. See Figure 3.1 (p. 58) for the definition of the data type `L`.

How does a local `let` definition limit the scope of the GADT branch reachability requirement? Recall that GADT type systems detect unreachable pattern-matching branches by inconsistency between scrutinee types and pattern types. When an expression flows to a `case` scrutinee directly without an intermediate `let`-bound variable, the type of expression must match the type of the scrutinee. For example, since `Nil` flows directly to `xs` in `e3`, `xs` must have type `L Z a`. This correspondence allows the plain GADT type system to detect that the `Cons` branch in `e3` is unreachable. In contrast, when an expression flows to a `case` scrutinee *indirectly* through a `let`-bound variable, the type of the expression is disentangled with the type of the scrutinee. There is no direct connection because the generalized type variables in the (polymorphic) type of the local definition are instantiated (*i.e.,* replaced by fresh types) on every reference. Since `Nil` flows to `xs` indirectly through the variable `null` in `e1`, the type of `xs` is disentangled with the type of `Nil`, and the `Cons` branch remains potentially reachable to the type system.

In the ADT type system, `let` expressions serve only one role: they provide a mechanism for introducing recursive local definitions with polymorphic types. In GADT type systems, they take on a new role: they restrict the enforcement of the branch reachability requirement. This new role makes `let` expressions an essential part of a GADT type system because they are the only way programmers can *use* GADT `case` expressions without risking branch reachability violations.

I want to make two more points before moving on. First, no one designed the LETREC-P type rule (which is used in the plain GADT type system to type `let` expressions) specifically to restrict the branch reachability requirement. This type rule, which also appears in the ADT type system (Figure 2.4, p. 27), was proposed by Mycroft [29] in 1984 based on earlier work by Milner [27] in 1978. This happy coincidence shows that seemingly unrelated type system features can interact in unexpected ways, so type system designers should exercise extreme caution when adding features to (or removing features from) a type system.

Second, there is no intrinsic connection between generalizing the type of a local definition and restricting the GADT branch reachability requirement. While the LETREC-P type rule uses the former to achieve the latter, there is no reason why it has to work this way. Indeed, it is possible to incorporate only one of these two type system features without the other. To restrict the GADT branch reachability requirement without generalizing the type of the local `let` definition, one can replace LETREC-P with the following LETREC-D type rule:

$$\text{LETREC-D}\quad \frac{\Gamma\{\mathrm{u} : \forall\overline{\alpha}.\, s\} \vdash \mathrm{e} : s \qquad \mathrm{dom}(\theta) \mathbin{\#} \mathit{tyvar}(\Gamma) \qquad \Gamma\{\mathrm{u} : \theta(s)\} \vdash \mathrm{d} : t}{\Gamma \vdash \texttt{let}\ \mathrm{u} = \mathrm{e}\ \texttt{in}\ \mathrm{d} : t}$$

Conversely, to generalize the types of local definitions without restricting the branch reachability requirement, one can use explicit type consistency constraints to represent the branch reachability requirement in the type system, and include the type constraints from a local `let` definition as part of its type. For example, such a GADT type system may give the `null` function in `e1` (Figure 5.2, p. 126) this qualified type in the `let` body (the ∼ symbol represents type consistency):

```
null :: forall m n a. (m ∼ Z) ∧ (m ∼ S n) ⇒ L m a → Bool
```

Using the type consistency constraints, the type system can now reject `null Nil` in the `let` body of `e1` because (`Z` ∼ `S n`) is not satisfiable.

5.1.3 Consequences of enforcing reachability

Though the GADT branch reachability requirement has seen tremendous uptake in the research community, this design decision is not without its quirks:

1. Enforcing the branch reachability requirement makes a GADT type system non-substitutive (so that $\Gamma \vdash \mathrm{e} : t$ no longer implies $\theta(\Gamma) \vdash \mathrm{e} : \theta(t)$ for an arbitrary type substitution θ), and

2. Using local `let` definitions to restrict the scope of the branch reachability requirement forces a GADT type system to violate type preservation under the standard small-step operational semantics.

Both consequences are significant. The first affects how researchers establish the safety property of a GADT type system, and the second affects the designs of GADT type inference algorithms. Figure 5.2 (p. 126) demonstrates these two consequences with examples:

1. The expressions `e1` and `e2` are identical, except that the `null` function has a more specific type in `e2` than in `e1`. The `null` function in `e1` is well-typed, but the `null` function in `e2` is not (because the `Cons` branch is unreachable). This pair of functions demonstrates that the plain GADT type system (or any GADT type system that adopts the branch reachability requirement) is not substitutive. In other words, an instance of a valid type of an expression may not be a valid type of the same expression.

2. The expression `e3` is the result of taking one reduction step from `e1` (the reduction replaces references of `null` in the `let` body with its definition). Since `e1` is well-typed, but `e3` is ill-typed (again, because the `Cons` branch is unreachable), this pair of expressions demonstrates that the plain GADT type system (or any GADT type system that uses local `let` definitions to restrict the branch reachability requirement) violates type preservation. In other words, a well-typed expression may reduce to an expression that is ill-typed.

This loss of type preservation deserves some elaboration. Even though type preservation and progress are a common strategy for proving type safety, type preservation is *not* a necessary condition for type safety. One can still prove the type safety of the plain GADT type system by embedding it into a more general

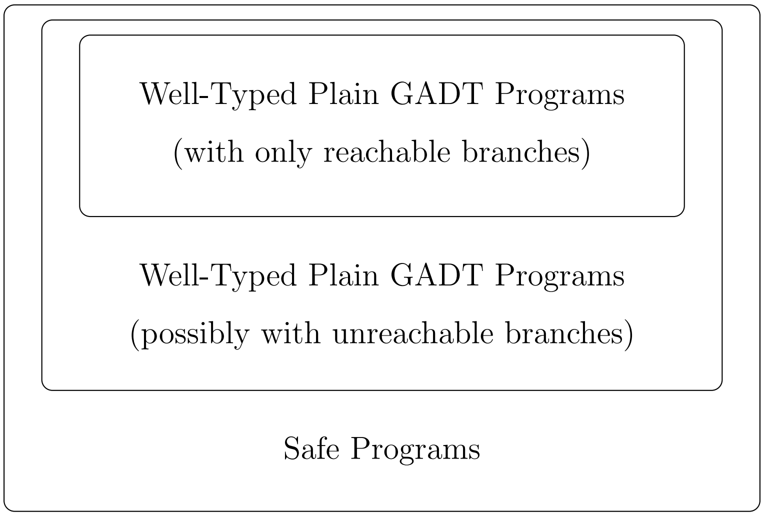

Well-Typed Plain GADT Programs
(with only reachable branches)
Well-Typed Plain GADT Programs
(possibly with unreachable branches)
Safe Programs

Figure 5.3: Embedding of GADT type systems.

type system that accepts unreachable branches (and thus has type preservation). Once the more general type system is proven safe, type safety of the plain GADT type system immediately follows (Figure 5.3, p. 130). Indeed, Peyton Jones et al. [17, §6.1] used this strategy to prove the soundness of their GADT type system.[1] Furthermore, since the plain GADT type system has a type-erasure semantics, the loss of type preservation has no effect on language implementations either. These discussions show that the loss of type preservation is less of a problem than it may first appear, and gaining the ability to restrict the branch reachability requirement at the cost of type preservation is, in my opinion, a very favorable trade.

5.1.4 Type inference

Earlier, I discussed the effects that the GADT branch reachability requirement has on the plain GADT type system. Since the design of a type inference algorithm relies heavily on the properties of the target type system, the branch reachability

[1] The published conference paper, however, does not say that the target type system differs from the source type system in that the former accepts unreachable branches but the latter does not. This difference is briefly discussed only in the companion technical appendix [15, §3], which suggests that the authors were perhaps not fully aware of its significance.

requirement must also have some influence on GADT type inference algorithms. In this subsection, I describe one such influence and explore ways that a GADT type inference algorithm can enforce the branch reachability requirement.

Since the plain GADT type system is an extension to the ADT type system, a natural way to design a GADT type inference algorithm is to extend an ADT type inference algorithm. An obvious candidate is the Algorithm $\mathcal{W}$ proposed by Milner [27], which is the earliest and the most well-known ADT type inference algorithm of all. Algorithm $\mathcal{W}$ is mostly compositional;[2] it collects type information from a program through depth-first traversal of the program's abstract-syntax tree. Given a type environment Γ and an expression e, a successful run of Algorithm $\mathcal{W}$ returns a type substitution θ and a type t such that $\theta(\Gamma) \vdash \mathrm{e} : t$ is a valid type judgment in the ADT type system.

Algorithm $\mathcal{W}$, which works by collecting and accumulating type information from a program, relies on the property that the ADT type system is substitutive (*i.e.,* applying a type substitution to a valid type judgment always yields a valid type judgment). Unfortunately, the plain GADT type system is not substitutive due to the branch reachability requirement, so a GADT type inference algorithm must either make substantial changes to Algorithm $\mathcal{W}$, or else adopt an entirely different structure.

Previous work on GADT type inference suggests three approaches to alleviate the non-substitutive nature of the plain GADT type system:

1. The first approach is to prohibit type substitutions on GADT scrutinee type variables in the type inference algorithm. The wobbly types algorithm [17], which adopts this approach, calls such type variables *rigid* (as opposed to *wobbly*). Since the type inference algorithm applies type substitutions only

[2] Algorithm $\mathcal{W}$ is only *mostly* compositional because it uses the inferred types of the local `let` definition to infer the type of the `let` body. Completely compositional type inference is possible [3, 4] but makes the treatment of `let` expressions considerably more complicated.

to the substitutive part of the GADT type system, the branch reachability requirement no longer causes any difficulties. The reliance on type annotations to distinguish type variables that can be substituted from those that cannot is the main limitation of this approach.

2. The second approach is to exclude GADT pattern-matching branches from type inference. Once type inference for the rest of the program completes, the algorithm enters a separate phase that checks the types and the reachability of GADT pattern-matching branches. The OutsideIn algorithm [36] adopts this approach. Since the algorithm checks GADT branch reachability last, there is no danger of a pattern-matching branch becoming unreachable due to subsequently-discovered type information. The inability to use type information from GADT pattern-matching branches for type inference is the main limitation of this approach.

3. The third approach is to represent GADT branch reachability as explicit type constraints in the type inference algorithm. These explicit type constraints capture the consistency requirement between the scrutinee and the pattern types of each pattern-matching branch, and the type inference algorithm checks these type constraints to ensure that any new type information discovered during type inference does not make a pattern-matching branch unreachable. Type inference via Herbrand constraint abduction [43] adopts this approach. Other than slightly complicating the type inference algorithm, this approach has no obvious limitations.

These approaches suggest that the GADT branch reachability requirement does not create any fundamental difficulty for GADT type inference algorithms. Still, there is a price in terms of the increased complexity of type inference algorithms, and as I will show, the price will only become steeper as I introduce more changes into the plain GADT type system.

5.2 POINTWISE BRANCH REACHABILITY

Like the plain GADT type system, the Pointwise GADT type system (§3.2) also enforces the GADT branch reachability requirement. The Pointwise GADT type system, however, has a stronger notion of consistency than the plain GADT type system: it requires the scrutinee type and the pattern type of a branch to be *pointwise unifiable* instead of merely unifiable. This stronger consistency requirement arises because the Pointwise GADT type system uses pointwise unifiers to combine scrutinee types and pattern types.

Using pointwise unifiers to type GADT pattern-matching branches has some clear benefits: it rejects certain pathological GADT programs by upholding the principle of orthogonal design (Chapter 3), and it leads to a set of conditions for identifying the generalized existential types in a GADT pattern (Chapter 4). In this section, I explore the other side of the story: the obligation to maintain *pointwise unifiability* between scrutinee types and pattern types during type inference. I will show that this seemingly trivial obligation is anything but: it significantly complicates type inference by interacting with standard type inference algorithm designs in obscure and convoluted ways.

5.2.1 Not pointwise, but unifiable

Ensuring that two types are unifiable, as required in a plain GADT type inference algorithm, is straightforward: the algorithm simply tries to unify the two types. If unification succeeds, the two types are unifiable. Otherwise, the two types are not unifiable, and type inference fails with an error. That is all there is to it.

A Pointwise GADT (or Non-Dependent GADT) type inference algorithm can easily adapt the same strategy to ensure that two types are pointwise unifiable: just perform the same steps but replace unification with pointwise unification. This adaptation is, however, inadequate for a complete type inference algorithm because

```
data P a b c where
  P1 :: forall a. P Int a [a]
  P2 :: forall a b c. a → b → c → P a b c

topple v w = case P2 v v w of
             P1 → True
             P2 a b c → False
```

Figure 5.4: This figure demonstrates a Non-Dependent GADT program whose type inference requires recovering from a pointwise unification failure. Note that the principal type (`P x x y`) of the `case` scrutinee (`P2 v v w`) is not pointwise unifiable with the pattern type (`P Int a [a]`) of the `P1` branch, but the `topple` function remains well-typed in the Non-Dependent GADT type system.

pointwise unification failures may be recoverable (thus they need not imply type inference failure). This curious phenomenon, which occurs when two types are unifiable but not pointwise unifiable, is best explained with an example.

Figure 5.4 (p. 134) shows the `topple` function, which is well-typed in both the Pointwise GADT and the Non-Dependent GADT type systems. Assuming that the variable `v` has type `x` and the variable `w` has type `y`, a Pointwise GADT type inference algorithm infers that the `case` scrutinee has type `P x x y`, and it checks that this type is pointwise unifiable with the pattern type `P Int a [a]` of the `P1` branch. The check fails because, as the following diagram shows, these two types are unifiable, but not pointwise unifiable:

$$\begin{tikzcd}
\text{Pattern Type} & \texttt{P} \arrow[d, equal] & \texttt{Int} \arrow[d] & \texttt{a} \arrow[r, dashed] & \texttt{[a]} \arrow[d] \\
\text{Scrutinee Type} & \texttt{P} & \texttt{x} \arrow[r, dashed] & \texttt{x} \arrow[u] & \texttt{y}
\end{tikzcd}$$

In the Pointwise GADT type system, the scrutinee type and the pattern type are inconsistent (because they are not pointwise unifiable). However, from the perspective of the type system, which tracks only types and not values, the `P1` branch remains reachable (the scrutinee type and the pattern type are unifiable, so the data constructor `P1` may still flow to the `case` scrutinee). This kind of minor type consistency violation does not exist in the plain GADT type system, and a Pointwise GADT type inference algorithm should deal with it gracefully instead of hastily reporting type inference failure.

The type inference algorithm should restore pointwise unifiability by *making the scrutinee type more specific.* In this example, if the algorithm infers `P Int Int [Int]` as the type of the `case` scrutinee (`P2 v v w`), this new scrutinee type would be consistent with the pattern type of the `P1` branch, and type inference for the `topple` function would succeed with the type `Int` → `[Int]` → `Bool`.

The `topple` function is not a special case, and the same fix also works for other similar examples. In the Pointwise GADT type system, if a scrutinee type and a pattern type are unifiable but not pointwise unifiable, a type inference algorithm can always recover pointwise unifiability by making the scrutinee type more specific. I show how in this subsection.

Figure 5.5 (p. 136) shows the *pointwise completion* algorithm, which I designed to restore pointwise unifiability between two types. Let s and t be two types such that $\mathit{tyvar}(s) \# \mathit{tyvar}(t)$. If s and t are unifiable, pointwise completion computes a most-general substitution σ such that $\sigma(s)$ is pointwise unifiable with t. If they are not unifiable, pointwise completion signals an error by returning the special symbol $\perp$.

The structure of the pointwise completion algorithm somewhat resembles that of the pointwise unification algorithm (§3.2). For example, the first three branches of *patch* correspond to the three failure rewrites in the conflict resolution phase of pointwise unification. The fourth branch of *patch* matches only when pointwise

```
$pwc(s,t) = patch(\text{id})$ where
  $patch(\sigma) = \text{case } (\sigma(s) \curlyvee t) \text{ of}$
    $\{(\alpha, T\,\overline{x})\} \uplus E$                  $\mid \alpha \in tyvar(E)$                          $\rightarrow patch([T\,\overline{\eta}/\alpha] \circ \sigma)$, $\overline{\eta}$ fresh
    $\{(\alpha, T\,\overline{x}), (y, \beta)\} \uplus E$ $\mid ndom(y, E), \beta \in tyvar(T\,\overline{x})$ $\rightarrow patch([T\,\overline{\eta}/\alpha] \circ \sigma)$, $\overline{\eta}$ fresh
    $\{(x, \beta), (y, \beta)\} \uplus E$     $\mid ndom(y, E), x \neq y \rightarrow$
        if $mgu(x \sim y) \neq \bot$ then $patch(mgu(x \sim y) \circ \sigma)$ else $\bot$
    $E \rightarrow$ if $E = \bot$ then $\bot$ else $\sigma$

$ndom(S\,\overline{y}, E) = \textsc{True}$
$ndom(\gamma, E)$    $= \gamma \in tyvar(E)$
```

Figure 5.5: This figure shows the pointwise completion algorithm, which computes a most-general type substitution σ from two types s, t such that $\sigma(s)$ is pointwise unifiable with t. The algorithm fails if s and t are not unifiable.

unification of $\sigma(s) \sim t$ succeeds ($E \neq \bot$) or fails in the counterpart collection phase ($E = \bot$). The resemblance is by design:

- When pointwise unification succeeds, pointwise completion succeeds immediately because $\sigma(s)$ and t are already pointwise unifiable.

- When pointwise unification fails in the counterpart collection phase, $\sigma(s)$ and t are not unifiable, so pointwise completion fails.

- When pointwise unification fails in the conflict resolution phase, a unifier θ of $\sigma(s) \sim t$, if it exists, requires non-pointwise type information flow. In this case, the pointwise completion algorithm applies a substitution to $\sigma(s)$ in an attempt to propagate local type information and thus eliminate the non-pointwise type information flow in θ.

The heart of the pointwise completion algorithm lies in the first three branches of *patch*, which deal with the non-trivial case of conflict resolution failure:

- The first branch matches when one occurrence of α in $\sigma(s)$ corresponds to $T\,\overline{x}$ in t, but another occurrence of α in $\sigma(s)$ does not. The algorithm deals with this disagreement by refining α to a fresh type $T\,\overline{\eta}$.

- The second branch matches when one occurrence of β in t corresponds to y in $\sigma(s)$, but another occurrence of β, which appears in $T\,\overline{x}$ in t, does not correspond to anything in $\sigma(s)$. The algorithm tries to generate a pointwise counterpart for the second occurrence of β in $T\,\overline{x}$ by refining α to a fresh type $T\,\overline{\eta}$.

- The third branch matches when two occurrences of β in t corresponds to different types (x and y) in $\sigma(s)$. The algorithm resolves this disagreement by unifying x and y so that these two occurrences of β correspond to the same type ($\mathit{mgu}(x \sim y)(x) = \mathit{mgu}(x \sim y)(y)$).

Pointwise completion is, in a way, an enhanced version of pointwise unification. Unlike the pointwise unification algorithm, which fails when the two input types require non-pointwise type information flow to unify, the pointwise completion algorithm eliminates the non-pointwise type information flow by identifying and propagating the non-local type information between the two types. Pointwise completion is sound (Theorem 27, p. 245), complete (Theorem 30, p. 250), and terminating (Theorem 22, p. 237). I present a proof of these three properties in Appendix A.

Example I demonstrate pointwise completion using the scrutinee type and the pattern type of the `P1` branch in the `topple` function (Figure 5.4, p. 134). To recover the consistency between these types, the Pointwise GADT type inference algorithm computes *pwc*(`P x x y`, `P Int a [a]`).

Figure 5.6 (p. 138) shows the successive calls to *patch* in the evaluation process; rows in the table represent calls to *patch* in chronological order. While the type

σ	$\sigma(s) \curlyvee t$	Branch Taken
id	$\{(\texttt{x}, \texttt{Int})\} \uplus \{(\texttt{x}, \texttt{a}), (\texttt{y}, \texttt{[a]})\}$	1st (recurses)
$[\texttt{Int/x}]$	$\{(\texttt{y}, \texttt{[a]})\} \uplus \{(\texttt{Int}, \texttt{a})\}$	2nd (recurses)
$[\texttt{Int/x}, \texttt{[d]/y}]$	$\{(\texttt{d}, \texttt{a}), (\texttt{Int}, \texttt{a})\} \uplus \{\}$	3rd (recurses)
$[\texttt{Int/x}, \texttt{[Int]/y}, \texttt{Int/d}]$	$\{(\texttt{Int}, \texttt{a})\}$	4th (returns σ)

Figure 5.6: Pointwise completion in action.

substitution σ grows with each recursive call, the set $\sigma(s) \curlyvee t$ shrinks (due to the propagation of local type information), and in the end the evaluation succeeds with the answer [`Int/x`, `[Int]/y`, `Int/d`]. It is simple to check that $\sigma(\texttt{P x x y}) =$ `P Int Int [Int]` is pointwise unifiable with `P Int a [a]`, so pointwise completion succeeds in computing a more specific scrutinee type that is consistent with the pattern type of the `P1` branch.

5.2.2 Interaction with let expressions

Adding pointwise completion to a Pointwise GADT type inference algorithm appears to solve the problem of maintaining the type consistency (*i.e.,* pointwise unifiability) between scrutinee types and pattern types. The solution is, however, incomplete because pointwise completion works by making scrutinee types more specific, and this mechanism breaks the standard type inference approach for `let` expressions. I give the details in this subsection.

Algorithm $\mathcal{W}$ (like other type inference algorithms based on it) infers a polymorphic type for a local `let` definition by inferring the type of the definien and then universally quantifying all type variables in the type of the definien that do

```
data P a b c where
  P1 :: forall a. P Int a [a]
  P2 :: forall a b c. a → b → c → P a b c
trip u v =
  let f w = case P2 u v w of
              P1 → True
              P2 a b c → False
  in v+3
```

Figure 5.7: This figure demonstrates a Non-Dependent GADT program where the type of a local `let` definition is affected by code in the `let` body. This interference prevents a type inference algorithm from generalizing the type of a local `let` definition using only type information that is locally available.

not appear free in the outer type environment [27]. This tried-and-true recipe is safe because, in the ADT and the plain GADT type systems, a type variable that does not appear free in the type environment is totally unconstrained. In other words, since the type inference algorithm will never find any additional information about such a type variable, there is no harm in making the type of the local `let` definition polymorphic in that type variable. Furthermore, this recipe is also convenient because it allows a type inference algorithm to make type generalization decisions using only type information that is available locally.

The `trip` function in Figure 5.7 (p. 139) demonstrates how pointwise completion interferes with this ability to make local type generalization decisions. This phenomenon, which shows that change in the GADT branch reachability requirement affects the typing of `let` expressions, echoes my earlier observation that `let` expressions are an integral part of a GADT type system. I will explain how a

Pointwise GADT type inference algorithm based on Algorithm $\mathcal{W}$ infers the type of `trip`, and then I will show that the type inferred for the local `let` definition `f` would be invalid in the Pointwise GADT type system.

Let us begin. Type inference for `trip` starts with type inference for `f`, and its definien has a most-general type `z → Bool` (where `z` is the type of `w`). Assuming that `u` has type `x` and `v` has type `y` in the environment, the `case` scrutinee `P2 u v w` has type `P x y z`. Since the scrutinee type is pointwise unifiable with the pattern types of both the `P1` and the `P2` branches, the type inference algorithm confirms that the definien of `f` makes no assumptions about the types of `u`, `v`, and `w`. The type variable `z` does not appear in the type environment of the `let` expression, so the type inference algorithm infers the type of `f` by binding `z` with a universal quantifier as follows:

```
f :: forall z. z → Bool
```

So far so good. Next, the type inference algorithm moves on to infer the type of the `let` body (`v+3`), which requires `v` to have type `Int`. With this newly discovered information, the type of the `case` scrutinee in `f` now becomes `P x Int z`, which is inconsistent with the pattern type `P Int a [a]` of the `P1` branch:

$$\begin{tikzcd}
\text{Pattern Type} & \texttt{P} \arrow[d, equal] & \texttt{Int} \arrow[d] & \texttt{a} \arrow[r, dashed] & \texttt{[a]} \arrow[d] \\
\text{Scrutinee Type} & \texttt{P} & \texttt{x} & \texttt{Int} \arrow[u] & \texttt{z}
\end{tikzcd}$$

In other words, `f` is now ill-typed in the Pointwise GADT type system due to type information discovered in the `let` body. This situation can never happen in the plain GADT type system, but it can in the Pointwise GADT type system.

Normally, pointwise completion can fix the problem by replacing `z` with `[Int]`. The type variable `z`, unfortunately, has already been generalized in the type of `f`, so it is no longer possible to apply the type substitution on `z`. The type inference algorithm is stuck and thus must report failure. In hindsight, the earlier decision to generalize z was a mistake. Type inference for trip would have succeeded had the algorithm inferred the following monomorphic type for f:

```
f :: z → Bool
```

Under this type, z is free in the type environment, so the type inference algorithm can apply the type substitution [[Int]/z] to restore the type consistency between the scrutinee type and the pattern type of the P1 branch.

What went wrong? Earlier, while playing the role of a type inference algorithm, I observed that the type consistency requirement between the scrutinee type and the pattern types places no restrictions on the type z of w. As it turns out, this observation is correct only when the type y of v is unconstrained. Once the type variable y is replaced by a more specific type (Int in this example), type consistency breaks down. To restore pointwise type information flow, the type inference algorithm must also make z more specific, which works only if z was not generalized. And therein lies the rub: what becomes of z depends on what becomes of y, but the type variable y appears free in the type environment of f. Should the type inference algorithm generalize z? Neither choice is ideal:

- Generalizing z in the type of f, as the trip function shows, leads to type inference failure when the type variable y becomes instantiated.

- Not generalizing z in the type of f makes the type of f unnecessarily restrictive if the type variable y is never instantiated.

To make the right call, the type inference algorithm must know what ultimately becomes of y, but this information is simply unavailable when the algorithm infers the type of the local let definition f. Once again, the strengthened GADT branch reachability requirement in the Pointwise GADT type system (that scrutinee types must be *pointwise* unifiable with pattern types) significantly complicates the type inference problem.

5.2.3 Type inference

In this section, I showed that the pointwise reachability requirement complicates the GADT type inference problem in two ways:

1. A complete type inference algorithm must (sometimes) restore pointwise unifiability by making the scrutinee types more specific, and
2. This mandate of making scrutinee types more specific as necessary breaks the tried-and-true recipe of inferring the types of local `let` definitions. More specifically, the set of type variables to generalize when inferring the type of a local `let` definition now depends on both the context and the body of the `let` expression.

There are at least three approaches that a Pointwise GADT (or Non-Dependent GADT) type inference algorithm can use to deal with these complications:

1. Instead of using pointwise completion to deal with the first complication, a type inference algorithm can simply report failure when a scrutinee type is not pointwise unifiable with the pattern type. This approach is easy to implement but makes the type inference algorithm incomplete. For example, type inference for `topple` (Figure 5.4, p. 134) would fail.
2. A type inference algorithm can use pointwise completion to restore pointwise unifiability and infer the types of `let` expressions as in Algorithm $\mathcal{W}$. The algorithm reports type inference failure when it needs to specialize a type variable that has already been generalized. This approaches also makes the type inference algorithm incomplete (but to a lesser degree than the first). For example, type inference for `topple` (Figure 5.4, p. 134) would succeed, but type inference for `trip` (Figure 5.7, p. 139) would fail.

3. A type inference algorithm can use explicit type constraints to track the instantiations of polymorphic types. The constraints associate each instantiated type with the original polymorphic type, so when the type inference algorithm retroactively specializes a type variable that has already been generalized, it can propagate the new type information (on the generalized type variable) uniformly to all instantiations of the type variable. This approach makes the type inference algorithm significantly more complicated than the previous two approaches, but it can allow successful type inference for both `topple` (Figure 5.4, p. 134) and `trip` (Figure 5.7, p. 139).

These approaches suggest that the complications I described in this section are solvable, if type inference algorithm designers are willing to cope with additional design complexity or to sacrifice completeness. The pointwise branch reachability requirement does not prohibit compositional type inference, but it may have made compositional type inference prohibitively expensive.

5.3 NESTED NON-DEPENDENT GADT BRANCHES

So far I have described type inference complications due to the GADT branch reachability requirement (§5.1) and additional complications due to the pointwise branch reachability requirement (§5.2). In this section, the trend continues, and I now show how the elimination of GADT type refinements in the Non-Dependent GADT type system complicates the type inference problem even further.

Let me start with how the branch reachability requirement affected the design of the Non-Dependent GADT type system. I designed the Non-Dependent GADT type system (§4.1) to meet the following two requirements. First, Non-Dependent GADT patterns should not introduce any local type assumptions. Second, every well-typed Non-Dependent GADT program should also be a well-typed Pointwise GADT program. These two requirements appear orthogonal: the first prohibits

```
divert :: forall n a. L n a → Bool
divert xs =
  case xs of
    Cons y ys → case xs of
      Nil → False

rematch :: forall n a. L n a → Bool
rematch xs =
  case xs of
    Nil → True
    Cons y ys → case xs of
      Cons w ws → False
```

Figure 5.8: This figure demonstrates the role that GADT type refinements play in enforcing the GADT branch reachability requirement. The `divert` function is ill-typed in all GADT type systems due to branch reachability violation, but `rematch` is ill-typed only in the Non-Dependent GADT type system (which, due to the elimination of GADT type refinements, is not expressive enough to detect that the inner `Cons` branch is reachable). See Figure 3.1 (p. 58) for the definition of the data type `L`.

GADT type refinements, and the second requires, among other things, that the type system enforces the branch reachability requirement.

These two requirements are, unfortunately, *not* orthogonal. The `divert` function (Figure 5.8, p. 144) demonstrates that GADT type refinements sometimes play a role in enforcing the GADT branch reachability requirement in the plain GADT type system. The `divert` function contains two nested `case` expressions that match the same scrutinee (`xs`) against different data constructors, so it is

	Plain GADT		Non-Dependent GADT
	w/ refinement	w/o refinement	Skolemization
`Cons` pattern	`L (S m) b`	`L (S m) b`	`L (S m) b`
`xs` (outer)	`L n a`	`L n a`	`L n a`
`xs` (inner)	`L (S m) a`	`L n a`	`L` $\mathbb{X}_{\mathtt{n}}$ `a`
`Nil` pattern	`L Z c`	`L Z c`	`L Z c`
`Nil` reachable?	No	Yes	No

Figure 5.9: This table illustrates how three GADT type systems (the plain GADT type system, the plain GADT type system without GADT type refinements, and the Non-Dependent GADT type system) check the reachability of the inner `Nil` branch in the `divert` function (Figure 5.8, p. 144).

clear that the inner (`Nil`) branch must be unreachable.

The left column of Figure 5.9 (p. 145) shows how the plain GADT type system catches the unreachable branch. The type system uses GADT type refinement to refine the type of the scrutinee (`xs`), and then unifying the refined scrutinee type (`L (S m) a`) with the pattern type (`L Z c`) of the `Nil` branch. The failure to unify `L (S m) a` with `L Z c` indicates that the `Nil` branch is unreachable, so the plain GADT type system rejects the `divert` function. The middle column of Figure 5.9 (p. 145) shows what happens without GADT type refinement: the scrutinee of the inner `case` expression would still have type `L n a` (despite that `xs` has already been successfully matched against the `Cons y ys` pattern), and the plain GADT type system would accept the `divert` function as well-typed.

This situation creates a technical contradiction: the Non-Dependent GADT type system must enforce the GADT branch reachability requirement, but the enforcement relies on GADT type refinement, which is the very feature I wish to eliminate in this type system. I ended up adopting a compromise in the design: instead of applying the GADT type refinement to the type environment and the type of the `case` expression, the ALT-NONDEP type rule (Figure 4.2, p. 99) substitutes Skolem type constants for type variables in the domain of the GADT type refinement. This compromise has the following effects:

- On the plus side, this design helps the Non-Dependent GADT type system reject the `divert` function. The right column of Figure 5.9 (p. 145) shows that the scrutinee of the inner `case` expression now has type `L` $\mathbb{X}_{\mathtt{n}}$ `a` (where $\mathbb{X}_{\mathtt{n}}$ is an uninhabited Skolem type constant). Since `L` $\mathbb{X}_{\mathtt{n}}$ `a` is not unifiable with `L Z c`, the Non-Dependent GADT type system agrees with the plain GADT type system and considers the `Nil` branch unreachable.

- On the minus side, substituting Skolem type constants for type variables could cause the Non-Dependent GADT type system to mistake a reachable branch for an unreachable one. The `rematch` function in Figure 5.8 (p. 144) demonstrates this problem. The inner `Cons` branch is definitely reachable because the scrutinee (`xs`) of the inner `case` expression has already been matched against the `Cons` data constructor. Unfortunately, without GADT type refinement, the Non-Dependent GADT type system is not sufficiently expressive to establish the reachability of this inner `Cons` branch, so it must consider this inner branch unreachable and reject the `rematch` program.

This inability to type certain nested `case` expressions with the same scrutinee is an unintended side effect of Non-Dependent GADT type system design. Since the Non-Dependent GADT type system was never intended for practical use, this reduction in expressiveness has little practical impact. The requirement to reject these nested `case` expressions, however, adds another layer of complexity on top of the already complicated Pointwise GADT type inference algorithms. The hope that I harbored at the end of Chapter 4 — that all the simplifications may lead to a simple and complete Non-Dependent GADT type inference algorithm — turned out to be unfounded. Incorporating the simplifications into the type rules did not make the type inference problem much simpler, instead it merely traded one set of difficulties for another.

5.4 SUMMARY

This chapter describes a few recurring themes that occurred throughout my dissertation research. These recurring themes represent principles of type system design that became apparent to me in the course of type inference research.

First, seemingly independent type system features may interact in unexpected ways. For example, local `let` definitions delimit the GADT branch reachability requirement (§5.1), and GADT type refinements play a role in enforcing the branch reachability requirement (§5.3). These interactions can affect type system properties (such as type preservation) and type inference algorithm design, so it is important to identify the feature interactions in a type system.

Due to these (sometimes obscure) interactions between type system features, making a type system more restrictive (*i.e.,* to accept fewer programs) does not necessarily make the type system simpler. More specifically, restricting a type system can make one aspect of the type system simpler, but it is also likely to make another aspect of the type system more complicated. Examples include:

- Enforcing branch reachability makes GADT type systems non-substitutive and breaks type preservation (§5.1),
- Requiring scrutinee types and pattern types to be pointwise unifiable interferes with the generalization of local `let` definition types (§5.2), and
- Eliminating GADT type refinements interferes with the enforcement of the GADT branch reachability requirement (§5.3).

In a type system where features are tightly connected (which is most of them), a local change can have far-reaching consequences. In principle, the consequences could work in your favor, but unfortunately that was not how they turned out in my case. Each successive type system simplification further complicates the type inference problem, and the complications then cause substantial increases in the design complexity of type inference algorithms. And in return for *what*? The mere ability to systematically reject certain classes of programs (which are all perfectly safe by virtue of being well-typed in an unrestricted type system).

In the beginning of this chapter, I asked whether these simplifications actually make the type inference problem more tractable. The answer remains both *yes* and *no*. Yes, the simplifications can guide a type inference algorithm in inferring a type for a pattern-matching branch. No, enforcing the simplifications where they serve no purpose (*i.e.,* everywhere except pattern-matching branches) is counterproductive because it makes the type inference algorithm more complicated than it needs to be. Since the simplifications do not contribute to type safety, their selective enforcement is a harmless way to make a type inference algorithm both simpler and more powerful.

To summarize, complete type inference for artificially restricted type systems can be a fool's errand because ensuring soundness (*i.e.,* that the algorithm fails for safe programs that lie outside the artificial restriction) tend to end up as the most complicated aspect of type inference algorithm design. Instead, it is better to develop incomplete type inference algorithms for an unrestricted type system. You may lose some bragging rights (there will be no completeness theorem), but the end result will be both simpler and more powerful. With this lesson at heart, I present a plain GADT type inference algorithm in the next chapter.

Chapter 6

GADT TYPE INFERENCE WITH ALGORITHM $\mathcal{P}$

In this chapter, I present Algorithm $\mathcal{P}$, which I designed for type inference in the plain GADT type system. Due to the difficulty of the GADT type inference problem (§2.4), I did not design Algorithm $\mathcal{P}$ in an attempt to achieve complete type inference. Instead, I designed it with the more practical goal of inferring types (without relying on type annotations) for as many well-typed GADT programs as possible. Even though Algorithm $\mathcal{P}$ is an incomplete type inference algorithm, its type inference power still represents a significant step forward compared to existing GADT type inference algorithms. The design of Algorithm $\mathcal{P}$ takes advantage of several plain GADT type system properties that I described earlier in this dissertation:

- Prevalence of pointwise type information flow in GADT patterns (§3.3),
- Generalized existential types in GADT branches (§4.2), and
- How GADT branch reachability interacts with local `let` definitions (§5.1).

In this chapter, I focus on the design of Algorithm $\mathcal{P}$ itself; readers who prefer a more example-oriented discussion on the capabilities of Algorithm $\mathcal{P}$ may skip ahead to Chapter 7.

6.1 OVERALL STRUCTURE

In this section, I describe the overall structure of Algorithm $\mathcal{P}$. Recall from §2.3 that the plain GADT type system extends the ADT type system by replacing the ALT-ADT type rule (for ADT pattern-matching branches) with the ALT-GADT type rule (for GADT pattern-matching branches). To take advantage of the commonality between these two type systems, I designed Algorithm $\mathcal{P}$ as an extension to Algorithm $\mathcal{W}$, which is a well-known type inference algorithm for the ADT type system. In this section, I focus on the parts of Algorithm $\mathcal{P}$ that are not directly related to type inference for `case` expressions, and I will leave details of type inference for GADT `case` expressions to the following two sections.

6.1.1 Foundation

I designed Algorithm $\mathcal{P}$ based on Milner's Algorithm $\mathcal{W}$ [27] and its extension proposed by Mycroft [29] that adds support for polymorphic recursion. In this subsection, I briefly review Algorithm $\mathcal{W}$ and Mycroft's extension.

Algorithm $\mathcal{W}$ is a complete type inference algorithm for the ADT type system without polymorphic recursion [8]. More specifically, it is complete for the set of type rules in Figure 2.4 (p. 27) but with the LETREC-P type rule (for polymorphic recursion) replaced by the LETREC-M type rule (for monomorphic recursion) in Figure 2.5 (p. 30). It works by collecting and accumulating type information from a program through depth-first traversal of the program abstract-syntax tree: given type environment Γ and expression e, a successful run of Algorithm $\mathcal{W}$ returns an idempotent type substitution θ and a type t such that $\theta(\Gamma) \vdash \mathrm{e} : t$ is a valid type judgment in the ADT type system.

To illustrate what Algorithm $\mathcal{W}$ looks like, Figure 6.1 (p. 151) shows how it infers the type of a function application (f e) under a type environment Γ. It infers the type of the function f under Γ and the type of the argument e under $\theta_1(\Gamma)$. It uses unification[1] to compute types for f and e that are consistent with respect

[1] Neither Algorithm $\mathcal{W}$ nor Algorithm $\mathcal{P}$ uses pointwise unifiers or pointwise unification. In the context of these two algorithms, the term unification always refers to the algorithm proposed by Robinson that computes most-general unifiers [35].

```
$infer(\Gamma, \text{f e}) =$
  let  $(\theta_1, t_1) = infer(\Gamma, \text{f})$
       $(\theta_2, t_2) = infer(\theta_1(\Gamma), \text{e})$
       $\theta_3 = \mathcal{U}(\theta_2(t_1) \sim t_2 \to \beta)$     $\beta$ fresh
  in  $(\theta_3 \circ \theta_2 \circ \theta_1, \theta_3(\beta))$
```

Figure 6.1: This figure shows the part of Algorithm $\mathcal{W}$ that infers the type of a function application. This algorithm is reproduced from Milner's 1978 paper on type polymorphism [27, §4.1].

to the APP type rule (for function application, Figure 2.4, p. 27), and it returns $\theta_3(\beta)$ as the type of the function application (f e) under the type environment $(\theta_3 \circ \theta_2 \circ \theta_1)(\Gamma)$.

I chose Algorithm $\mathcal{W}$ as the foundation for Algorithm $\mathcal{P}$ because it is simple, well understood, and widely known. There is, however, a slight problem with this choice: Algorithm $\mathcal{W}$ does not support type inference with polymorphic recursion. Polymorphic recursion, which allows a recursive definition to invoke itself on an instance of its own type, is prevalent in GADT programs. Due to this prevalence, a practical GADT type inference algorithm must support polymorphic recursion. Instead of abandoning Algorithm $\mathcal{W}$, I adopted the extension proposed by Mycroft that adds support for polymorphic recursion to Algorithm $\mathcal{W}$ [29].

Figure 6.2 (p. 152) shows Mycroft's extension to Algorithm $\mathcal{W}$ for type inference with polymorphic recursion. The expression snippet (u=e) in the first line represents the local definition in the `let` expression (`let` u = e `in` d), and here the equal sign works as an infix polymorphic recursion fixpoint operator. Mycroft's algorithm first assumes that the recursive reference u has the fully polymorphic type $\forall\alpha.\, \alpha$ in e, and it then iteratively specializes the (overly general) assumed type of

```
$\mathit{infer}(\Gamma,\ \mathrm{u{=}e}) = \mathit{polyrec}(\mathrm{id},\ \forall\alpha.\,\alpha)$ where
  $\mathit{polyrec}(\theta_1,\ x) =$
    let  $(\theta_2,\ t) = \mathit{infer}(\Gamma\{\mathrm{u} : x\},\ \mathrm{e})$
         $\overline{\gamma} = \mathit{tyvar}(t) \setminus \mathit{tyvar}(\Gamma\{\mathrm{u} : x\})$
         $y = \forall\overline{\gamma}.\, t$
    in  if  $\theta_2(x) = y$
        then  $(\theta_2 \circ \theta_1,\ y)$
        else  $\mathit{polyrec}(\theta_2 \circ \theta_1,\ y)$
```

Figure 6.2: This figure shows Mycroft's extension to Algorithm $\mathcal{W}$ for type inference with polymorphic recursion. The name "id" on the first line represents the identity type substitution. This algorithm is adapted from Mycroft's 1984 paper on polymorphic recursion [29, §6].

u until the iteration reaches a fixpoint. Even though this extension, which Mycroft proposed in 1984, predates the development of GADT type systems, my experience suggests that it also works well in Algorithm $\mathcal{P}$ for plain GADT programs that require polymorphic recursion.

Of course, since type inference with polymorphic recursion is computationally undecidable [12, 21], Mycroft's algorithm *will* loop forever on some programs. To ensure termination of Algorithm $\mathcal{P}$, I placed a limit on the number of iterations that *polyrec* can make. If *polyrec* finds a fixpoint within the iteration limit, type inference for the recursive definition succeeds. If, however, it reaches the iteration limit before finding a fixpoint, type inference for the recursive definition fails.

6.1.2 Compositionality

In this subsection, I describe a modification I made to Algorithm $\mathcal{W}$ to improve compositionality in Algorithm $\mathcal{P}$. This modification does not improve the power of Algorithm $\mathcal{P}$. Instead, it simplifies the development of Algorithm $\mathcal{P}$, and it makes the behavior of the algorithm more predictable to programmers.

Algorithm $\mathcal{W}$, which is the foundation of Algorithm $\mathcal{P}$, performs type inference in a mostly-compositional manner. One notable exception to this compositionality appears in the way Algorithm $\mathcal{W}$ infers a type for a function application (f e): type inference for the argument e depends on the type inference result for the function f (Figure 6.1, p. 151). In other words, Algorithm $\mathcal{W}$ exhibits a left-to-right bias. In addition to function applications, this bias also affects other language constructs that contain multiple sub-expressions (such as `case` expressions).

I want to eliminate this bias in Algorithm $\mathcal{P}$ for the following two reasons:

1. The bias introduces sideways type information flow (*e.g.,* from a function to its arguments), which makes it difficult for programmers to predict or to understand the propagation of type information during type inference. Since type inference for `case` expressions in Algorithm $\mathcal{P}$ can be sensitive to the amount of type information available from the context, any sideways type information flow could hinder programmers' ability to understand the behavior of Algorithm $\mathcal{P}$.

2. In general, compositional type inference algorithms are easier to design than algorithms that are not compositional. A non-compositional type inference algorithm (such as Algorithm $\mathcal{W}$) must ensure that the type environment used for type inference is always up-to-date (with respect to the type information discovered during type inference), and it must always compose type substitutions in the right order. Both requirements clutter the algorithm specification and make refactoring more difficult than it need to be.

```
$infer(\Gamma,\ \mathrm{f\ e}) =$
  let  $(\theta_1,\ t_1) = infer(\Gamma,\ \mathrm{f})$
       $(\theta_2,\ t_2) = infer(\Gamma,\ \mathrm{e})$
       $\eta = \mathcal{U}_S(\theta_1 \sim \theta_2)$
       $\theta_3 = \mathcal{U}(\eta(t_1) \sim \eta(t_2) \to \beta)$     $\beta$ fresh
  in  $(\theta_3 \circ \eta \circ \theta_1,\ \theta_3(\beta))$
```

Figure 6.3: This figure shows McAdam's proposal for compositional type inference for function application. Note that $\theta_3 \circ \eta \circ \theta_1 = \theta_3 \circ \eta \circ \theta_2$ because η unifies θ_1 with θ_2. This algorithm is reproduced from McAdam's 1999 paper on substitution-unification [26, Figure 6].

The modification that I made to Algorithm $\mathcal{W}$ in Algorithm $\mathcal{P}$ was inspired by McAdam's work on *substitution-unification* [26]. In his work, McAdam proposed the substitution-unification algorithm $\mathcal{U}_S$, which unifies type substitutions instead of types. Given idempotent type substitutions θ_1 and θ_2, $\mathcal{U}_S(\theta_1 \sim \theta_2)$ computes a most-general idempotent type substitution η such that $\eta \circ \theta_1 = \eta \circ \theta_2$. McAdam calls η a most-general unifier of θ_1 and θ_2.

McAdam proposed a modification to Algorithm $\mathcal{W}$ that eliminates the left-to-right bias by using substitution-unification. Figure 6.3 (p. 154) shows McAdam's proposal. Here, type inference for the argument (e) no longer depends on the type inference result for the function (f). Since θ_1 and θ_2 are independent type substitutions, McAdam combines them by substitution-unification instead of substitution composition.

This strategy for compositional type inference is a step in the right direction. Unfortunately, it is applicable only to expressions that contain two parts (such as function application) because the algorithm $\mathcal{U}_S$ cannot unify more than two type substitutions at once. McAdam did propose a naïve algorithm that unifies a set of type substitutions using $\mathcal{U}_S$ [25, §4.6]; the algorithm, however, does not help with the problem because it turns out to be correct only for sets that contain exactly two substitutions. Due to the large number of ways that the domains of a set of substitutions may overlap, directly extending $\mathcal{U}_S$ to unify an arbitrary number of substitutions appears to require extensive and non-trivial modifications.

After repeated failures to extend the algorithm $\mathcal{U}_S$, I developed the notion of *substitution-combination* to replace substitution-unification. Given a set of idempotent type substitutions $S = \{\theta_1, \ldots, \theta_n\}$, $\mathcal{C}(S)$ computes a most-general idempotent type substitution ρ such that the following condition holds:

$$\forall\, t_1,\, t_2.\, \forall\, \theta.\ (\theta \in S) \wedge (\theta(t_1) = \theta(t_2)) \supset \rho(t_1) = \rho(t_2)$$

This condition states that ρ unifies any pair of types t_1 and t_2 that are unified by any element θ of S (the $\supset$ symbol represents logical implication). Substitution-combination is closely related to substitution-unification: given two unifiable type substitutions θ_1 and θ_2, the following equations hold modulo variable renaming:

$$\mathcal{C}(\{\theta_1, \theta_2\}) = \mathcal{U}_S(\theta_1 \sim \theta_2) \circ \theta_1 = \mathcal{U}_S(\theta_1 \sim \theta_2) \circ \theta_2$$

Unlike substitution-unification, it is easy to compute $\mathcal{C}(S)$ for a set of idempotent type substitutions $S = \{\theta_1, \ldots, \theta_n\}$:

$$\mathcal{C}(S) = \mathcal{U}((\overline{\alpha_1}, \ldots, \overline{\alpha_n}) \sim (\theta_1(\overline{\alpha_1}), \ldots, \theta_n(\overline{\alpha_n}))) \quad \text{where } \overline{\alpha_i} = \mathrm{dom}(\theta_i)$$

For example,

$$\mathcal{C}(\{[\texttt{(b,c)/a}], [\texttt{Int/b}, \texttt{Bool/c}]\}) = \mathcal{U}((\texttt{a}, (\texttt{b}, \texttt{c})) \sim (\texttt{(b,c)}, (\texttt{Int}, \texttt{Bool})))$$

By unifying every type variable α in the domain $\overline{\alpha}$ of a type substitution θ_i with $\theta_i(\alpha)$, the substitution-combination algorithm ensures that $\mathcal{C}(S)$ contains all (and nothing but) the type equations implied by the type substitutions in the set S.

```
$\mathit{infer}(\Gamma, \mathrm{f\ e}) =$
  let  $(\theta_1, t_1) = \mathit{infer}(\Gamma, \mathrm{f})$
       $(\theta_2, t_2) = \mathit{infer}(\Gamma, \mathrm{e})$
       $\theta_3 = \mathcal{U}(t_1 \sim t_2 \to \beta)$     $\beta$ fresh
       $\rho = \mathcal{C}(\{\theta_1, \theta_2, \theta_3\})$
  in  $(\rho, \rho(\beta))$
```

Figure 6.4: Function application type inference in Algorithm $\mathcal{P}$.

As an example of how Algorithm $\mathcal{P}$ achieves compositional type inference using substitution-combination, Figure 6.4 (p. 156) shows how it infers the type of a function application. Note that Algorithm $\mathcal{P}$ computes θ_1, θ_2, and θ_3 independently and combines them all with substitution-combination. This compositional strategy addresses both of my concerns about the left-to-right bias in Algorithm $\mathcal{W}$: there is no sideways type information flow, and the type inference algorithm can combine type substitutions irrespective of ordering.

6.1.3 Branch reachability

The plain GADT type system requires every pattern-matching branch in a well-typed program to be potentially reachable (§5.1). Since reachability is an extrinsic property of a GADT pattern-matching branch (*i.e.,* it depends on both the branch and its context), enforcing GADT branch reachability in a compositional type inference algorithm tends to require non-local changes to the algorithm. In this subsection, I describe how Algorithm $\mathcal{P}$ enforces the GADT branch reachability requirement.

Algorithm $\mathcal{P}$ enforces the GADT branch reachability requirement by turning the consistency (*i.e.,* unifiability) between scrutinee types and pattern types into explicit type constraints. Whenever Algorithm $\mathcal{P}$ infers a new type substitution, it applies the substitution to the scrutinee types in the constraints and checks that the scrutinee types in the constraints remain consistent (*i.e.,* unifiable) with the pattern types. If the check fails, Algorithm $\mathcal{P}$ reports type inference failure due to violation of the GADT branch reachability requirement.

Instead of giving a full account of how Algorithm $\mathcal{P}$ enforces GADT branch reachability here, I will illustrate this feature through an example. Consider the expression `e1` (reproduced here from Figure 5.2, p. 126):

```
e1 = let null :: forall m a. L m a → Bool
         null xs = case xs of
           Nil → True
           Cons y ys → False
     in null Nil
```

Applying Algorithm $\mathcal{P}$ to the `case` expression in the local `let` definition of `null` produces the following type constraints:

$$(\texttt{L m a} \sim \texttt{L Z b}) \quad \text{and} \quad (\texttt{L m a} \sim \texttt{L (S n) c})$$

Here `L m a` is the type of the `case` scrutinee `xs`, `L Z b` is the pattern type of the `Nil` branch, and `L (S n) c` is the pattern type of the `Cons` branch. Algorithm $\mathcal{P}$ generates one constraint for each pattern-matching branch, and each constraint represents the reachability of the corresponding branch. In this example, since both constraints contain two consistent types, both branches are reachable.

Type constraints in Algorithm $\mathcal{P}$ have an unusual feature: they cannot appear in polymorphic types. In many type systems with type constraints, generalizing a constrained type (*i.e.,* a type that is subject to a type constraint) produces a polymorphic constrained type. For example, consider the following Haskell type that uses the type classes feature [14, §4.1]:

```
Eq a ⇒ a → a → Bool
```

The type constraint (`Eq a`) restricts `a` to types in the `Eq` class. Generalizing this type produces the following polymorphic type:

```
forall a. Eq a ⇒ a → a → Bool
```

Note that the type variable `a` in the constraint (`Eq a`) is universally quantified with the rest of the polymorphic type. Instantiating this polymorphic type also produces a fresh copy of the type constraint:

```
Eq a1 ⇒ a1 → a1 → Bool
```

This example shows that including type constraints in a polymorphic type helps type systems to keep track of type constraints through type generalization and instantiation. Algorithm $\mathcal{P}$ does *not* keep (branch reachability) type constraints in a polymorphic type because, unlike other type systems, it does *not* want to keep track of the type constraints through type generalization and instantiation. For example, Algorithm $\mathcal{P}$ infers the following polymorphic type for `null`:

```
forall m a. L m a → Bool
```

Instead of this constrained polymorphic type:

```
forall m a. (L m a ∼ L Z b) ∧ (L m a ∼ L (S n) c) ⇒ L m a → Bool
```

Dropping type constraints when generalizing the type of a local `let` definition allows Algorithm $\mathcal{P}$ to restrict the enforcement of GADT branch reachability as required by the plain GADT type system (§5.1).

6.2 BRANCH TYPE INFERENCE

Let us move on to the more interesting part of Algorithm $\mathcal{P}$: type inference for `case` expressions in plain GADT programs. This is the part of Algorithm $\mathcal{P}$ that is most difficult to design because it must confront the lack of principal types in the plain GADT type system and use GADT type refinements appropriately to resolve inconsistencies between the body types of pattern-matching branches in the `case` expression. Here, heuristics play an important role: in situations that require an arbitrary decision, instead of resorting to backtracking search, I try to make Algorithm $\mathcal{P}$ choose the option that, based on available type information, best matches (my understanding of) programmers' intuition.

Figure 6.5 (p. 160) outlines how Algorithm $\mathcal{P}$ performs type inference for `case` expressions. The algorithm in the figure collects type information from multiple sources, and most of its design difficulty comes from finding appropriate ways to combine the collected information in a useful manner. Figure 6.6 (p. 161) briefly describes each variable that appears in the algorithm, and it summarizes the input and output of each type inference step in a chart to facilitate navigation. Each column in the chart corresponds to a step (or a local definition) in the algorithm; the $\circ$ symbol marks its arguments, and the $\bullet$ symbol marks its results. Each row in the chart represents a variable in the algorithm; the $\bullet$ symbol marks its definition, and the $\circ$ symbol marks its uses.

The chart suggests that the algorithm works in two phases. The first phase (steps 0–9) consolidates all type information about the `case` expression into the placeholder type variable β, the unified scrutinee type s, and the type substitutions η_i (one for each pattern-matching branch). The second phase (steps A–F) uses the consolidated type information from the first phase to produce the type inference result for the `case` expression. In this section, I describe the first phase, and I will talk about the second phase in the next section.

```
0    $\mathit{infer}(\Gamma, \texttt{case}\ \mathrm{e}\ \texttt{of}\ \{\,\overline{\mathrm{p}_i \to \mathrm{c}_i}\,\}) = \mathit{res}$ where
1      $\beta$ fresh
2      $(\theta_0, u_0) = \mathit{infer}(\Gamma, \mathrm{e})$
3,4    $(\theta_i, u_i, x_i) = \mathit{infer}_{\mathrm{ALT}}(\Gamma, \beta, \mathrm{p}_i \to \mathrm{c}_i)$    for all $1 \le i \le n$        (§6.2.1)
5      $u \ \ = \mathit{comm}(x_1, \ldots, x_n)$                                                 (§6.2.2)
6      $\theta \ \ = \mathcal{U}((u, \ldots, u) \sim (u_0, u_1, \ldots, u_n))$
7      $s \ \ = \theta(u)$
8,9    $\eta_i \ \ = \mathcal{C}(\{\theta, \theta_0, \theta_i, \mathcal{U}(u_i \sim x_i)\})$    for all $1 \le i \le n$
A      $\overline{\kappa} \ \ = \mathit{tyvar}(s) \cap (\cup_i \operatorname{dom}(\eta_i))$
B      $\overline{\gamma} \ \ = \mathit{tyvar}(s) \setminus \overline{\kappa}$
C      $\mathit{trt} = \mathit{tabulate}(\overline{\kappa}, \eta_1, \ldots, \eta_n)$                              (§6.3.1)
D      $\mathit{btt} = \mathit{tabulate}(\mathit{tyvar}(\Gamma) \cup \{\beta\}, \eta_1, \ldots, \eta_n)$               (§6.3.1)
E      $\rho \ \ = \mathit{reconcile}(\overline{\gamma}, \mathit{trt}, \mathit{btt})$                              (§6.3.2)
F      $\mathit{res} = (\rho, \rho(\beta))$
```

Figure 6.5: This figure shows how Algorithm $\mathcal{P}$ infers a type for a GADT `case` expression. The numbers preceding each step in the algorithm correspond to the column labels in Figure 6.6 (p. 161), which also provides a short description of each variable that appears in this algorithm.

$btt = \mathit{tabulate}(\mathit{tyvar}(\Gamma) \cup \{\beta\},\, \eta_1,\, \eta_2)$

$\theta = \mathcal{U}((u,\, u,\, u) \sim (u_0,\, u_1,\, u_2))$

$(\theta_1,\, u_1,\, x_1) = \mathit{infer}_{\mathrm{ALT}}(\Gamma,\, \beta,\, \mathrm{p}_1 \rightarrow \mathrm{c}_1)$

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	
·	·	○	·	·	·	·	·	·	·	·	○	○	○	·	●	Γ — Input type environment
○	·	○	·	·	·	·	·	·	·	·	○	○	·	●		β — Branch body type placeholder
·	·	·	·	·	·	○	○	·	·	·	·	·	●			θ_0 — Type subst from `case` scrutinee
·	·	·	·	·	·	·	·	·	○	·	·	·	●			u_0 — Type from `case` scrutinee
·	·	·	·	·	·	·	○	·	·	·	·	●				θ_1 — Type subst from branch #1
·	·	·	·	·	·	·	○	·	○	·	·	●				u_1 — Scrutinee type from branch #1
·	·	·	·	·	·	·	○	·	·	○	·	●				x_1 — Pattern type from branch #1
·	·	·	·	·	·	○	·	·	·	·	●					θ_2 — Type subst from branch #2
·	·	·	·	·	·	○	·	·	○	·	●					u_2 — Scrutinee type from branch #2
·	·	·	·	·	·	○	·	·	·	○	●					x_2 — Pattern type from branch #2
·	·	·	·	·	·	·	·	○	○	●						u — Scrutinee-type specialization template
·	·	·	·	·	·	○	○	○	●							θ — Unifier of scrutinee types
·	·	·	·	○	○	·	·	●								s — Unified scrutinee type
·	·	○	○	·	○	·	●									η_1 — Complete type subst for branch #1
·	·	○	○	·	○	●										η_2 — Complete type subst for branch #2
·	·	·	○	○	●											$\overline{\kappa}$ — Scrutinee type indices
·	○	·	·	●												$\overline{\gamma}$ — Scrutinee type parameters
·	○	·	●													trt — Type refinement table
·	○	●														btt — Branch type table
○	●															ρ — Type subst for `case` expression
●																res — Result for `case` type inference

● Variable definition
○ Variable reference

Figure 6.6: This figure illustrates the steps and the intermediate results when Algorithm $\mathcal{P}$ (Figure 6.5, p. 160) infers a type for a `case` expression that has two branches. Each row shows the uses ○ and the definition ● of a variable, and each column corresponds to one step of type inference (see the algorithm line numbers and annotations above the chart).

6.2.1 Single-branch type inference

Figure 6.7 (p. 163) shows how Algorithm $\mathcal{P}$ infers the type of a GADT pattern-matching branch. This algorithm closely resembles the Non-Dependent GADT pattern-matching branch type inference algorithm (Figure 4.4, p. 114), which I already described in detail in §4.2. Therefore in this subsection I describe only the modifications that I made in the Algorithm $\mathcal{P}$ rendition.

The most obvious change in Figure 6.7 is that $\mathit{infer}_{\mathrm{ALT}}$ has a slightly different interface from *infer*, which is used in Algorithm $\mathcal{W}$:

$$(\theta,\, u \to t) = \mathit{infer}(\Gamma, p\ \to c)$$
$$(\theta \circ [t/\beta],\, u,\, x) = \mathit{infer}_{\mathrm{ALT}}(\Gamma, \beta, p\ \to c)$$

Here $(\theta,\, t)$ is the result of type inference for the branch body, u is the inferred branch scrutinee type, and x is the branch pattern type. The $\mathit{infer}_{\mathrm{ALT}}$ function uses its type variable argument β to embed the inferred branch body type t into the returned type substitution $\theta \circ [t/\beta]$, and it also returns the pattern type x as part of the result. I designed the new interface of $\mathit{infer}_{\mathrm{ALT}}$ to accommodate the needs of `case` expression type inference in Algorithm $\mathcal{P}$.

A more significant change lies in the treatment of pattern-bound type variables that find their way into the branch body type or the type environment. Recall that all GADT type systems I presented in this dissertation prohibit escape and instantiation of generalized existential type variables (§4.2). Both Non-Dependent GADT type inference (Figure 4.4, p. 114) and Algorithm $\mathcal{P}$ (Figure 6.7, p. 163) fail immediately if an existential type (*i.e.,* a type variable that is introduced by a pattern but does not appear in the pattern type of the branch, see §4.2) appears in the branch body type or the type environment. The algorithms, however, behave differently if the type variable in question *does* appear in the pattern type.

Non-Dependent GADT type inference takes a proactive approach: it specializes the inferred scrutinee type so that the pattern type variable is not considered a

```
$\mathit{infer}_{\mathrm{ALT}}(\Gamma,\, \beta,\, \mathrm{C}\,\overline{\mathrm{x}} \to \mathrm{c}) =$
  let  $(\forall\overline{\alpha}.\,\overline{w} \to T\ \overline{s}) = \mathit{lookup}(\mathrm{C})$ where $\overline{\alpha}$ fresh
       $(\eta,\, t) = \mathit{infer}(\Gamma\{\overline{\mathrm{x} : w}\},\, \mathrm{c})$
       $\overline{\tau} = \overline{\alpha} \cap (\mathrm{dom}(\eta) \cup \mathit{tyvar}(\mathrm{rng}(\eta)) \cup \mathit{tyvar}(t))$
       $\overline{\gamma} = \{\gamma \mid \gamma \in \overline{\alpha} \wedge \mathit{retain}(\overline{\alpha},\, \eta,\, \gamma)\}$
  in  if $\overline{\tau} \not\subseteq \mathit{tyvar}(\overline{s})$  then $\bot$
      else  if $\overline{\gamma} = \emptyset$
            then  $([t/\beta] \circ \eta,\, T\ \overline{\mu},\, T\ \overline{s})$ where $\overline{\mu}$ fresh
            else  $([t/\beta] \circ \eta,\, \eta(\mathit{transcb}(\overline{\gamma},\, T\ \overline{s})),\, T\ \overline{s})$

$\mathit{retain}(\overline{\alpha},\, \eta,\, \gamma) =$ case $\eta(\gamma)$ of
  $U\ \overline{w} \to \textsc{True}$
  $\nu \quad\ \to \exists\, \kappa \in \overline{\alpha}.\, \kappa \neq \gamma \wedge \nu \in \mathit{tyvar}(\eta(\kappa))$

$\mathit{transcb}(\overline{\gamma},\, t) =$
  if $\overline{\gamma} \# \mathit{tyvar}(t)$
  then  $\mu$ where $\mu$ fresh
  else  case t of
          $U\ \overline{w} \to U\ \langle \mathit{transcb}(\overline{\gamma},\, z) \mid z \in \overline{w} \rangle$
          $\nu \quad\ \to \nu$
```

Figure 6.7: This figure shows how Algorithm $\mathcal{P}$ infers a type for a GADT pattern-matching branch (*cf.* Figure 4.4, p. 114). This algorithm has a slightly different interface from the rest of Algorithm $\mathcal{P}$: it returns the branch body type through substitution on the placeholder type variable β, and it returns both the inferred scrutinee type and the pattern type of the branch.

```
listId xs = case xs of
  Nil → Nil
  Cons y ys → Cons y ys
```

Figure 6.8: Length-indexed list identity function.

generalized existential type in the Non-Dependent GADT type system (and thus can legally escape from the branch). Algorithm $\mathcal{P}$, in contrast, takes the wait-and-see approach. It does nothing during type inference for a pattern-matching branch and instead opts to deal with the problem later.

I adopted the wait-and-see attitude in Algorithm $\mathcal{P}$ because, in the presence of GADT type refinements, a single pattern-matching branch contains insufficient information to detect escaped type variables accurately. The `listId` function in Figure 6.8 (p. 164) illustrates the problem. Consider the following results from applying Algorithm $\mathcal{P}$ to the `Cons` branch:

- Branch scrutinee type: `L m b`
- Branch pattern type: `L (S n) a`
- Branch body type: `L (S n) a`

By Definition 6 (p. 104), the type variable `n` is a generalized existential type. Since `n` appears in the branch body type, the proactive approach requires specializing the scrutinee type to `L (S k) b`. In the Non-Dependent GADT type system, which does not support GADT type refinements, the proactive approach is appropriate because a generalized existential type that appears in the branch body type will definitely escape from the branch (and make the branch ill-typed).

However, in the plain GADT type system, which does support GADT type refinements, the proactive response may be premature: a generalized existential type that appears in the branch body type need not necessarily escape. In the `listId` example, if Algorithm $\mathcal{P}$ infers the type `L m b` for the `case` expression (which it will), then there would be no escape because the generalized existential type `n` is "recaptured" by the GADT type refinement [`S n`/`m`]. That is not all: the proactive response (of specializing the scrutinee type) is not only premature but also counterproductive. Since the specialized scrutinee type `L (S k) b` is not unifiable with the pattern type `L Z a` of the `Nil` branch, the proactive approach will cause type inference failure for `listId` due to branch reachability violation.

To avoid these problems with the proactive response to the potential escape of generalized existential types, when inferring the type of a single pattern-matching branch, Algorithm $\mathcal{P}$ considers only the instantiated pattern type variables for specializing the branch scrutinee type.

6.2.2 Scrutinee-type specialization

The reference chart for Algorithm $\mathcal{P}$ (Figure 6.6, p. 161) states that the scrutinee type unifier θ, which the algorithm uses to compute the unified scrutinee type s, combines type information from four sources (for a two-branch `case` expression):

- The type inferred from the `case` scrutinee (u_0),
- The scrutinee type inferred from branch #1 (u_1),
- The scrutinee type inferred from branch #2 (u_2), and
- The scrutinee-type specialization template (u).

The first three types are pretty self-explanatory, and in this subsection I explain what the fourth type — the scrutinee-type specialization template — is and why I included it in Algorithm $\mathcal{P}$.

```
head xs = case xs of
  Cons y ys → y
```

Figure 6.9: This figure shows the `head` function for length-indexed lists (*cf.* the `tail` function in Figure 4.1, p. 94). See Figure 3.1 (p. 58) for the definition of the `Cons` data constructor.

Earlier I described how the idea of generalized existential types relates to the appropriate specificity of a branch scrutinee type (§4.2). For a GADT pattern-matching branch to be well-typed, its scrutinee type must be specific enough to support escaped and instantiated pattern type variables (*i.e.,* type variables that are bound by the pattern of the branch). At the same time, the GADT branch reachability requirement (§5.1) dictates that the scrutinee type of a branch must remain general enough to retain the reachability of all pattern-matching branches in the `case` expression. These two opposing demands work in tandem to define the appropriate specificity of a `case` scrutinee type.

There are, however, some situations where the two opposing demands do not completely fix the specificity of a `case` scrutinee type. Figure 6.9 (p. 166) shows the `head` function, which demonstrates one such situation. The function has the following two types (plus many more that are not listed here):

```
head :: forall m a. L m a → a
head :: forall n a. L (S n) a → a
```

The first type is valid: since the type variable `n`, which appears in the pattern type `L (S n) a` of the `Cons` branch, does not escape or become instantiated, there is no need to make the scrutinee type `L m a` more specific. The second type is also valid: since `head` does not have a `Nil` branch, the (more specific) scrutinee type `L (S n) a` cannot cause any branch reachability violations.

Both types are valid. But which one is *better*? The first type is more general — in fact, it is the most general type of `head`. The second type, however, is more informative: it clearly states that `head` is defined only for non-empty lists, and it makes the GADT type checker enforce this precondition at compile time. Most programmers should agree that the second type is *better* than the first.

Programs such as `head` add an interesting twist to GADT type inference. In GADT type systems, a more general type is not necessarily better than a more specific one, and the most-general type (even when one is available) may not be the "best" type in practice. As a result, a GADT type inference algorithm should sometimes infer types that are not the most-general. The design decisions then become *where*, *when*, and *how specific*.

The *where* part is easy: Algorithm $\mathcal{P}$ infers more specific types only for `case` scrutinees (and it tries to infer the most-general types elsewhere). The *when* part is also easy: Algorithm $\mathcal{P}$ makes the type of a `case` scrutinee more specific when all pattern types in the `case` expression have a common top-level structure that does not appear in the `case` scrutinee type. For example, in the `head` function, all pattern types (there is only one) are of the form `L (S` x`)` y, so Algorithm $\mathcal{P}$ decides that it should make the inferred scrutinee type `L m a` more specific. The last part, *how specific*, is closely related to the *when*. Algorithm $\mathcal{P}$ makes the `case` scrutinee type only specific enough to share the same top-level structure with all pattern types in the `case` expression.

Algorithm $\mathcal{P}$ uses the scrutinee-type specialization template u to represent the common top-level structure in pattern types (Figure 6.6, p. 161). It computes u with the *comm* function, which implements a simplified version of least-general generalization [31] by extracting common top-level type constructors from a set of types into a new type. Here are a few examples:

$comm($`L (S n) a`$) =$ `L (S k) b`

$comm($`Maybe Int`$,$ `Maybe a`$) =$ `Maybe b`

$$\mathit{comm}(\texttt{Maybe Int}, \texttt{Maybe Bool}) = \texttt{Maybe b}$$

$$\mathit{comm}(\texttt{(a, a)}, \texttt{(b, b)}) = \texttt{(c, d)}$$

Since u contains the common top-level structure that is shared by all pattern types, Algorithm $\mathcal{P}$ specializes the inferred `case` scrutinee type by unifying u with other sources of scrutinee type information (as I described in the beginning of this subsection).

The way Algorithm $\mathcal{P}$ specializes an inferred `case` scrutinee type is not just a clever type inference trick — it reflects the following fundamental principle about the quality of `case` scrutinee types:

> The best type for a `case` scrutinee is the one that best matches[2] the pattern types of the branches in the `case` expression.

In the ADT type system, all pattern types are uniform (§2.2), so the principle states that the most-general scrutinee types are the best ones in the ADT type system. In GADT type systems, where pattern types may be non-uniform, the best scrutinee types are not necessarily the most-general ones, but the ones that best approximate the common structure of the pattern types. These more-specific scrutinee types restrict the values that can flow to the `case` scrutinee, but that is actually a good thing: the whole point of using GADT type systems is to allow such finer distinctions.

I now end this subsection by demonstrating how scrutinee-type specialization makes Algorithm $\mathcal{P}$ more powerful than it would otherwise be. In addition to the aforementioned benefits, a more specific scrutinee type also induces GADT type refinements that are more discriminating (and thus have greater typing power). The `wrap` function in Figure 6.10 (p. 169) demonstrates this effect. To determine

[2] Here I use the phrase "best match" informally. For a formal measure of how well two types match, one can count the number of type constructors in their most-general unifier (the fewer the better). Inconsistent types, which have no unifier, are the worst matches of each other.

```
data T a where
  T1 :: T (Maybe Int)
  T2 :: T (Maybe Bool)

wrap e = case e of
  T1 → [3]
  T2 → [True]
```

Figure 6.10: An example for scrutinee-type specialization.

which one of `T a` and `T (Maybe b)` is the better `case` scrutinee type for the `wrap` function, let us consider the GADT type refinements induced by each choice. The scrutinee type `T a` induces the following pair of GADT type refinements:

- [`Maybe Int/a`] for the `T1` branch, and
- [`Maybe Bool/a`] for the `T2` branch.

In contrast, the scrutinee type `T (Maybe b)`, which Algorithm $\mathcal{P}$ infers for `wrap` using scrutinee-type specialization, induces this pair of GADT type refinements:

- [`Int/b`] for the `T1` branch, and
- [`Bool/b`] for the `T2` branch.

This second pair of GADT type refinements is more useful than the first. Unlike the first pair of type refinements, which cannot reconcile the branch body types `[Int]` and `[Bool]` in `wrap`, the second pair can do so with `[b]`. By specializing the inferred scrutinee types, Algorithm $\mathcal{P}$ strips off common top-level structure in GADT type refinements from different branches. As a result, the type refinements become more discriminating, which allows Algorithm $\mathcal{P}$ to infer types for more GADT programs.

```
data D x y z where
  D1 :: D Int Bool a
  D2 :: D [b] [c] b

refine e = case e of
  D1 → (3, True)
  D2 → ([True], [])
```

Figure 6.11: Running example for branch type reconciliation.

6.3 GADT TYPE REFINEMENTS

To infer a type for a case expression, a GADT type inference algorithm must combine type information from all pattern-matching branches in the case expression through judicious use of GADT type refinements. In this section, I explain how Algorithm $\mathcal{P}$ accomplishes this task.

The plain GADT type system lacks principal types (§2.4) because GADT type refinements allow a GADT pattern-matching branch to have multiple maximal types (which are not instances of one another). The lack of principal types for GADT pattern-matching branches, which leads to the lack of principal types for GADT programs, complicates the type inference problem because most existing algorithms infer only one type for each expression.

Having multiple GADT pattern-matching branches in a case expression adds an interesting twist to the problem because different branches can place different demands on the type of the case expression. On the plus side, even if individual branches have no principal types, competing demands from different branches may give the case expression a principal type. On the minus side, the principal type, if it exists, does not come for free: a type inference algorithm must sort through competing demands to infer a type for the case expression as a whole.

To demonstrate the second phase of `case` expression type inference in Algorithm $\mathcal{P}$, I will use the `refine` function in Figure 6.11 (p. 170) as the running example for this section. Using this example, I will show how Algorithm $\mathcal{P}$ organizes type information from pattern-matching branches and how it uses GADT type refinements to resolve inconsistent type information between branches.

6.3.1 Tabulating branch types

Let me start by summarizing the intermediate type inference result for the `case` expression in `refine` from the first phase (steps 0–9) of Algorithm $\mathcal{P}$. Under the type environment $\Gamma = \{\texttt{e} : \texttt{m}\}$, the branch body type placeholder $\beta = \texttt{n}$, and the pattern types $x_1 = \texttt{D Int Bool a}$ (for the `D1` branch) and $x_2 = \texttt{D [b] [c] b}$ (for the `D2` branch), the first phase produces the following three results:

- $s = \texttt{D x y z}$
- $\eta_1 = [\texttt{Int/x}, \texttt{Bool/y}, \texttt{z/a}, (\texttt{D Int Bool z})\texttt{/m}, \texttt{(Int,Bool)/n}]$
- $\eta_2 = [\texttt{[z]/x}, \texttt{[c]/y}, \texttt{z/b}, (\texttt{D [z] c z})\texttt{/m}, \texttt{([Bool],[d])/n}]$

The unified scrutinee type s contains all currently known information about the type of the `case` scrutinee. The type substitutions η_1 and η_2 contain all currently known type information about the `D1` and the `D2` branches, which includes:

- Parametric instantiation (on pattern type variables `a` and `b`). For example, in the `D1` branch, the pattern type variable `a` is instantiated to the type `z`.
- Type indexing (on scrutinee type variables `x` and `y`). For example, in the `D2` branch, the scrutinee type variable `x` refines to the type `[z]`.
- Environment types (substitution on the environment type variable `m`). For example, the body of the `D1` branch assumes `m` in the type environment Γ to be (`D Int Bool z`).

- Body types (substitution on the placeholder type variable `n`). For example, the body of the `D2` branch has type `([Bool], [d])`.

The second phase of `case` expression type inference in Algorithm $\mathcal{P}$ uses the following pieces of type information as input: $\mathit{tyvar}(\Gamma) \cup \{\beta\}$, $\mathit{tyvar}(s)$, η_1, and η_2. Even though `m` represents the type of the scrutinee `e` in the type environment Γ, Algorithm $\mathcal{P}$ does not treat `m` differently due to its special status. As far as the algorithm is concerned, `m` is just another type variable in $\mathit{tyvar}(\Gamma)$ and therefore does not warrant special treatment.

Since the type substitutions η_1 and η_2 contain all type information about the `D1` and the `D2` branches, Algorithm $\mathcal{P}$ no longer needs other pieces of branch type information such as x_1 or θ_1. Type substitutions are, however, not very easy to work with, so Algorithm $\mathcal{P}$ tabulates the type information contained in η_1 and η_2 into two tables:

- The type refinement table (*trt*) lists the GADT type refinements that are available in each branch. Each row in the table corresponds to a branch, and each column corresponds to a type index variable in the scrutinee type. Each cell in the table shows the GADT type refinement for a specific type index in a specific branch. Only the rows in the table are ordered.

- The branch type table (*btt*) lists the Refined type role (which consists of the branch body type and the types in the environment, see §2.5, Figure 2.9, p. 49) for each branch. Each row in the table corresponds to a branch, and each column corresponds to either the placeholder type variable β or a type variable in the type environment. Each cell in the table shows the type assumed for a specific type variable in the body of a specific branch. Only the rows in the table are ordered.

Algorithm $\mathcal{P}$ uses these two tables to combine type information from different branches. Basically, it uses the branch type table (*btt*) to identify inconsistent types between the branches, and it then searches the type refinement table (*trt*) in an attempt to reconcile the inconsistency by applying GADT type refinements. Algorithm $\mathcal{P}$ uses the *tabulate* function to build these two tables. Given a set of type variables $\{\alpha_1, \ldots, \alpha_m\}$ and a sequence of type substitutions $(\eta_1, \ldots, \eta_n)$, the *tabulate* function builds the following table:

	Type Variables			
Branch	α_1	α_2	$\cdots$	α_m
1	$\eta_1(\alpha_1)$	$\eta_1(\alpha_2)$	$\cdots$	$\eta_1(\alpha_m)$
2	$\eta_2(\alpha_1)$	$\eta_2(\alpha_2)$	$\cdots$	$\eta_2(\alpha_m)$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
n	$\eta_n(\alpha_1)$	$\eta_n(\alpha_2)$	$\cdots$	$\eta_n(\alpha_m)$

To build the branch type table (*btt*) for the `refine` example, Algorithm $\mathcal{P}$ invokes *tabulate*$(\{\texttt{m}, \texttt{n}\}, \eta_1, \eta_2)$, which builds the following table:

	Branch Types	
Branch	`m`	`n`
`D1`	`D Int Bool z`	`(Int,Bool)`
`D2`	`D [z] [c]  z`	`([Bool],[d])`

For reference, here are the type substitutions η_1 and η_2, which I reproduced from the beginning of this subsection:

$$\eta_1 = [\texttt{Int/x},\ \texttt{Bool/y},\ \texttt{z/a},\ (\texttt{D Int Bool z})\texttt{/m},\ \texttt{(Int,Bool)/n}]$$

$$\eta_2 = [\texttt{[z]/x},\ \texttt{[c]/y},\ \texttt{z/b},\ (\texttt{D [z] c z})\texttt{/m},\ \texttt{([Bool],[d])/n}]$$

Building the type refinement table (*trt*) is slightly more complicated because Algorithm $\mathcal{P}$ must first decide on the columns of the table (which correspond to the type indices in the unified scrutinee type). The algorithm computes the type indices $\overline{\kappa}$ as the scrutinee type variables that appear in the domain of any branch type substitution η_i (Figure 6.5, p. 160), and for the `refine` example it computes $\overline{\kappa} = \{\mathtt{x}, \mathtt{y}\}$. This approach works best if an algorithm implementation can orient the type substitutions η_i to minimize the size of $\overline{\kappa}$ (identifying a type parameter as a type index is harmless but could cause unnecessary type inference failures). Invoking *tabulate*$(\overline{\kappa}, \eta_1, \eta_2)$ builds the following type refinement table:

	Refinements	
Branch	`x`	`y`
`D1`	`Int`	`Bool`
`D2`	`[z]`	`[c]`

Note that there is no column for `z` because `z` is not a type index.

Since Algorithm $\mathcal{P}$ applies GADT type refinements by matching a column in the type refinement table to a column in the branch type table, it is sometimes easier to follow the algorithm if I set the two tables side-by-side as follows:

	Refinements		Branch Types	
Branch	`x`	`y`	`m`	`n`
`D1`	`Int`	`Bool`	`D Int Bool z`	`(Int,Bool)`
`D2`	`[z]`	`[c]`	`D [z] [c] z`	`([Bool],[d])`

The reader should keep in mind that there remain two separate tables whose contents have very different semantics.

6.3.2 Applying type refinements

In this subsection, I describe the *reconcile* function, which combines type information from the pattern-matching branches in the `case` expression. The function accepts the following three pieces of type information as input:

- The scrutinee type parameters $\overline{\gamma} = \mathit{tyvar}(s) \setminus \overline{\kappa}$,

- The type refinement table (*trt*), and
- The branch type table (*btt*).

In return, *reconcile* computes a type substitution ρ so that $(\rho,\ \rho(\beta))$ is the type inference result for the `case` expression. In other words, ρ contains the combined type information from all pattern-matching branches in the `case` expression.

Type variable classification In *reconcile*, Algorithm $\mathcal{P}$ classifies the type variables that appear in the two tables (example for `refine` shown here) into five groups. Type variables in each group serve a different function and thus requires a slightly different treatment:

	Refinements		Branch Types	
Branch	`x`	`y`	`m`	`n`
`D1`	`Int`	`Bool`	`D Int Bool z`	`(Int,Bool)`
`D2`	`[z]`	`[*c]`	`D [z] [*c] z`	`([Bool],[d])`

1. The first group is *type indices*; each type variable in this group corresponds to a column in the type refinement table. In the `refine` example, the type indices are `x` and `y`. A type index represents different type refinements in different branches.

2. The second group is *outer type variables*; each type variable in this group corresponds to a column in the branch type table. In the `refine` example, the outer type variables are `m` and `n`. These type variables are in the Outer type role (§2.5), and Algorithm $\mathcal{P}$ applies GADT type refinements across branches by substituting a type index for an outer type variable.

3. The third group is *type parameters*; they are the scrutinee type variables that do not provide GADT type refinements. In the `refine` example, the only

type parameter is `z`. In Algorithm $\mathcal{P}$, type parameters transfer type information directly across branch boundaries. They are the only type variables that are not subject to GADT type refinements and therefore can appear in multiple rows in the two tables.

For each row in the two tables that corresponds to a branch, Algorithm $\mathcal{P}$ replaces all type variables in the row (except type parameters) with fresh type variables. This renaming step prevents unintended type information propagation between branches. Since the two branches in `refine` share only the type variable `z` (which is a type parameter), the renaming step is not strictly necessary in this example, and I will skip it here to avoid confusion.

4. The fourth group is *pattern type variables*; they are the type variables that appear in the (renamed) type refinement table but are neither type indices nor type parameters. In the `refine` example, the only pattern type variable is `c`. These type variables are akin to generalized existential types (§4.2), and Algorithm $\mathcal{P}$ replaces them with Skolem type constants (shown in the table with the `*` prefix) to prevent their escape and instantiation.

5. The fifth group is the *body type variables*; they are the type variables in the (renamed) branch type table that do not belong to any of the previous four groups. In the `refine` example, the only body type variable is `d`. These type variables are not directly accessible from the context of the `case` expression because they exist only in the scope of a specific pattern-matching branch.

Column type consistency The goal of the *reconcile* function is to combine the types in the branch type table into a type substitution ρ on the outer type variables (which correspond to the columns in the branch type table). Every column in the branch type table falls into one of the following three categories:

1. If all types in a column are consistent and contain no Skolem type constants, then all branches place consistent demands on the outer type variable of the column. I call such a column a *consistent column.* It is easy to infer a type substitution on the outer type variable: unifying all types in the column will do the trick.

2. If a column contains two types that are inconsistent, then there must be two branches that place contradictory demands on the outer type variable of the column. I call such a column an *inconsistent column.* Reconciling the contradictory demands requires applying GADT type refinements.

3. If a Skolem type constant appears in a column, then the corresponding pattern type variable may escape through the outer type variable of the column. I also call such a column an *inconsistent column*, and preventing the pattern type variable from escaping also requires applying GADT type refinements.

Tactics for inconsistent columns Algorithm $\mathcal{P}$ has two tactics for extracting information from an inconsistent column in the branch type table.

1. If all types in the inconsistent column are built from the same type constructor, Algorithm $\mathcal{P}$ destructs the column by replacing the outer type variable of the column with a fresh type built from the same type constructor. Each type argument for the destructed type constructor forms a new column in the branch type table.

2. Alternatively, if the (*btt*) column is unifiable with exactly one column in the type refinement table (*trt*), Algorithm $\mathcal{P}$ applies GADT type refinements by replacing the outer type variable of the (*btt*) column with the type index of the (*trt*) column.

By design, Algorithm $\mathcal{P}$ never applies these tactics to a consistent column. In other words, it applies GADT type refinements only if there are no alternatives (*i.e.,* only in case of type inconsistencies that cannot be reconciled by other means). Similarly, Algorithm $\mathcal{P}$ applies GADT type refinements only if there is a unique matching column in the type refinement table. In other words, it applies GADT type refinements only when there is no ambiguity about which type index to use. Both designs help Algorithm $\mathcal{P}$ behave in a conservative and predictable manner.

Example I now demonstrate the two tactics using the `refine` example:

$$\rho = \text{id}$$

	Refinements		Branch Types	
Branch	`x`	`y`	`m`	`n`
`D1`	`Int`	`Bool`	`D Int Bool z`	`(Int,Bool)`
`D2`	`[z]`	`[*c]`	`D [z] [*c] z`	`([Bool],[d])`

The tables preceding this sentence (reproduced from p. 175) show the inputs of the *reconcile* function after Algorithm $\mathcal{P}$ Skolemizes the pattern type variables. The type substitution ρ (shown above the tables) starts as the identity substitution, and it will grow as Algorithm $\mathcal{P}$ extracts more information from the branch type table. Here, columns `m` and `n` are both inconsistent; since Algorithm $\mathcal{P}$ does not require a specific order when dealing with inconsistent columns, I (randomly) choose to work on the `m` column first. Both types in the `m` column are built from the type constructor `D`, so Algorithm $\mathcal{P}$ destructs the column and replaces `m` with the fresh type (`D o p q`):

$$\rho = [(\texttt{D o p q})/\texttt{m}]$$

	Refinements		Branch Types			
Branch	`x`	`y`	`o`	`p`	`q`	`n`
`D1`	`Int`	`Bool`	`Int`	`Bool`	`z`	`(Int,Bool)`
`D2`	`[z]`	`[*c]`	`[z]`	`[*c]`	`z`	`([Bool],[d])`

Note that ρ has recorded the type substitution on `m`, and the `m` column in the branch type table is replaced by three new columns (`o`, `p`, and `q`), which correspond to the three type arguments of the data constructor `D`.

Next, I choose to work on the `p` column. Column `p` illustrates what I call an *opposable thumb*: an outer type variable that needs to have different top-level type constructors in different branches. Opposable thumbs are useful to Algorithm $\mathcal{P}$ because they clearly indicate the need for GADT type refinements, and they often contain enough information to identify the exact type refinements to apply. Here, the `p` column is unifiable with exactly one column (`y`) in the type refinement table, which means that `y` is the only type index whose type refinements can reconcile the inconsistencies for the outer type `p`. Therefore, Algorithm $\mathcal{P}$ applies to `p` the GADT type refinements that are induced by `y`: it replaces `p` with `y` and removes the `p` column from the branch type table:

$$\rho = [(\texttt{D o y q})/\texttt{m}, \texttt{y}/\texttt{p}]$$

	Refinements		Branch Types		
Branch	x	y	o	q	n
D1	Int	Bool	Int	z	(Int,Bool)
D2	[z]	[*c]	[z]	z	([Bool],[d])

Note that ρ now also includes the type substitution on `p`. Algorithm $\mathcal{P}$ has applied each tactic once, and two inconsistent columns (`o` and `n`) remain. Since a branch type table may have many inconsistent columns, Algorithm $\mathcal{P}$ repeatedly applies the tactics until there are no applicable inconsistent columns left, and I will do the same in this demonstration. Next I destruct the `n` column:

$$\rho = [(\texttt{D o y q})/\texttt{m}, \texttt{y}/\texttt{p}, (\texttt{r,s})/\texttt{n}]$$

	Refinements		Branch Types			
Branch	x	y	o	q	r	s
D1	Int	Bool	Int	z	Int	Bool
D2	[z]	[*c]	[z]	z	[Bool]	[d]

Next I apply GADT type refinements from the x column to the o column:

$$\rho = [(\texttt{D x y q})/\texttt{m}, \texttt{y}/\texttt{p}, (\texttt{r,s})/\texttt{n}, \texttt{x}/\texttt{o}]$$

	Refinements		Branch Types		
Branch	x	y	q	r	s
D1	Int	Bool	z	Int	Bool
D2	[z]	[*c]	z	[Bool]	[d]

Next I apply GADT type refinement from the x column to the r column, but things are a little different this time around: the two columns have a nontrivial most-general unifier $\sigma = [\texttt{Bool}/\texttt{z}]$. To apply the GADT type refinement from x, Algorithm $\mathcal{P}$ checks that σ does not replace a type parameter with a type that contains Skolem type constants (it does not because $\sigma(\texttt{z}) = \texttt{Bool}$), and it applies σ to both tables as follows:

$$\rho = [(\texttt{D x y q})/\texttt{m}, \texttt{y}/\texttt{p}, (\texttt{x,s})/\texttt{n}, \texttt{x}/\texttt{o}, \texttt{x}/\texttt{r}]$$

	Refinements		Branch Types	
Branch	x	y	q	s
D1	Int	Bool	Bool	Bool
D2	[Bool]	[*c]	Bool	[d]

Note that all occurrences of z in the tables are replaced by Bool. There is only one inconsistent branch s left, which Algorithm $\mathcal{P}$ eliminates by applying the

GADT type refinement y. Unifying the columns s and y produces the most-general unifier $\sigma = [\texttt{*c/d}]$, which replaces d with a Skolem type constant. Since d is not a type parameter, the escape check does not apply, so Algorithm $\mathcal{P}$ does not report an error. Applying σ to the tables produces no observable effect because the only occurrence of d is removed along with the rest of the s column:

$$\rho = [\texttt{(D x y q)/m}, \texttt{y/p}, \texttt{(x,y)/n}, \texttt{x/o}, \texttt{x/r}, \texttt{y/s}]$$

	Refinements		Branch Types
Branch	x	y	q
D1	Int	Bool	Bool
D2	[Bool]	[*c]	Bool

At this point, neither tactic is applicable to the only remaining column in the branch type table, so the iterative process in the *reconcile* function ends.

Final steps Since Algorithm $\mathcal{P}$ has exhausted its tactics, it checks the final branch type table for consistency:

- Are there any Skolem type constants left in the branch type table? Type inference fails if the answer is yes, because the *yes* answer indicates that a pattern type variable escaped from a branch.
- Are all the rows in the branch type table unifiable? Type inference fails if the answer is no, because the *no* answer indicates type inconsistencies that are beyond the capabilities of the two tactics employed by Algorithm $\mathcal{P}$.

The final branch type table in the retain example, which contains a single consistent column q, passes both tests, so Algorithm $\mathcal{P}$ unifies the rows in the table with the outer type variable q to produce the type substitution [Bool/q]. Composing it with ρ from the previous step produces the final result for *reconcile*:

$$\rho = [\texttt{(D x y Bool)/m}, \texttt{y/p}, \texttt{(x,y)/n}, \texttt{x/o}, \texttt{x/r}, \texttt{y/s}, \texttt{Bool/q}]$$

This type inference result ρ, combined with the placeholder type variable β and the type environment Γ, states that the `case` expression in the `refine` function has type `(x,y)` under the assumption that the `case` scrutinee `e` has type `D x y Bool`. In other words, the `refine` function has the following type:

```
refine :: forall x y. D x y Bool → (x, y)
```

This result concludes my demonstration of how Algorithm $\mathcal{P}$ applies GADT type refinements to reconcile type inconsistencies between pattern-matching branches.

Discussion Algorithm $\mathcal{P}$ relies on opposable thumbs to determine inconsistent columns and to identify the appropriate GADT type refinements that it should apply to reconcile the inconsistencies (p. 179). In the `refine` example, opposable thumbs readily arise from the branches themselves. In other programs, however, their formation may rely on additional type information from the context. This potential dependency on contextual type information is why compositionality is important for the predictability of Algorithm $\mathcal{P}$ (§6.1).

Observant readers may have noticed that the type Algorithm $\mathcal{P}$ inferred for `refine` (which is also its only type) requires non-pointwise type information flow between the scrutinee type and the pattern type of the `D2` branch:

$$\begin{tikzcd}
\text{Pattern Type} & \mathtt{D} \arrow[d, equal] & \mathtt{[b]} \arrow[d] & \mathtt{[c]} \arrow[d] & \mathtt{b} \arrow[ll, dashed, bend right] \\
\text{Scrutinee Type} & \mathtt{D} & \mathtt{x} & \mathtt{y} & \mathtt{Bool} \arrow[u]
\end{tikzcd}$$

Algorithm $\mathcal{P}$ achieves this feat by classifying `b` (which appears in the tables as `z`) as a type parameter, so it can specialize it to `Bool` and rely on non-pointwise type information flow to propagate this type specialization.

6.4 SUMMARY

In this chapter, I presented Algorithm $\mathcal{P}$, which I designed to infer types for plain GADT programs. My goal for Algorithm $\mathcal{P}$ was not to achieve completeness, but to test (and to push) the boundaries of GADT type inference in the total absence of programmer type annotations.

Algorithm $\mathcal{P}$ follows a conservative design strategy; it relies heavily on ideas and previous work by Milner [27], Mycroft [29], and McAdam [26]. Its structure is largely conventional, and most of the new machinery is isolated in the parts of the algorithm that deal with GADT pattern-matching branches and `case` expressions. The relatively unobtrusive extension from Algorithm $\mathcal{W}$ to Algorithm $\mathcal{P}$ should make the new type inference machinery easier to understand and to implement in other type inference systems.

The behavior of Algorithm $\mathcal{P}$ is similarly conservative by design. Algorithm $\mathcal{P}$ is refinement-averse; it tries to avoid applying GADT type refinements whenever possible. Even when GADT type refinements are necessary, Algorithm $\mathcal{P}$ applies them only if it can identify a unique type index to use. This conservative design helps Algorithm $\mathcal{P}$ mimic the behavior of Algorithm $\mathcal{W}$ (which is already familiar to many programmers), and it should make the behavior of Algorithm $\mathcal{P}$ easier for programmers to predict and to understand.

The design of Algorithm $\mathcal{P}$ takes advantage of many properties of the plain GADT type system. For example, Algorithm $\mathcal{P}$ specializes the inferred type of a `case` scrutinee based on the principle that a scrutinee type should be a good match to the pattern types. In other situations, however, type system properties are not very useful for judging design decisions. For example, both the proactive and the wait-and-see approaches to dealing with the escape of generalized existential type variables will likely be wrong for infinitely many programs. Making good decisions in these situations may rely more on cognitive sciences than on mathematics, and

I try to make do with my intuition and experiences.

Of course, an important, outstanding question is “does it really work?” In the next chapter I will illustrate the power and the limitations of Algorithm $\mathcal{P}$ through 32 program examples.

Chapter 7

ALGORITHM $\mathcal{P}$ BY EXAMPLE

In the previous chapter, I described the design of Algorithm $\mathcal{P}$. Does it actually work in practice? In this chapter, I try to answer this question by demonstrating Algorithm $\mathcal{P}$ through 32 program examples. I selected these examples from my type inference algorithm test suite, which contains 145 test cases. I designed a few of the selected examples to illustrate specific features of Algorithm $\mathcal{P}$; the others came from the following applications domains:

- Dimensional types [37, §4],
- Length-indexed lists,
- Generic N-way zip [37, §5.3],
- Tagless term interpreters [38, §5.7],
- Functional reactive programming [30],
- Monad libraries,
- Type equality witnesses,
- Integer ordering witnesses [38, §3.7],
- Shape-indexed binary-tree paths [38, §3.3],
- Balance-indexed AVL trees [38, §4.1], and
- Color-indexed red-black trees [38, Appendix A].

The following table shows the distribution of program examples:

Well-typed examples for which type inference succeeds	25
Well-typed examples for which type inference fails	5
Ill-typed examples for which type inference fails	2
Total	32

Of particular interest are the five well-typed examples for which type inference fails. They illustrate limitations of Algorithm $\mathcal{P}$ and highlight possible starting points for future work, and I will discuss these examples in §7.5.

7.1 ALGORITHM IMPLEMENTATION

I developed Algorithm $\mathcal{P}$ as a Haskell program. This program is currently the only complete specification of the algorithm. The algorithm implementation consists of 854 lines of code (excluding comments and blank lines), which includes 154 lines of source language definition for the Frown parser generator [13]. The complete source code of the implementation is listed in Appendix B. To help me evaluate design decisions and prevent regression during development, I maintained a suite of test programs, which currently has 805 lines of code with 146 top-level definitions. Type inference for the entire test suite takes about three seconds on an IBM ThinkPad X30 laptop computer manufactured in 2003.

The algorithm implementation uses a source language syntax that is similar to the Haskell syntax I used throughout this dissertation, but with the requirement that constructs with a variable number of parts (such as local `let` definitions and pattern-matching branches in `case` expressions) must enclose those parts in curly brackets and separate them with semicolons. For example, here is what the `null` function in Figure 4.1 (p. 94) looks like in this modified syntax:

```
null xs = case xs of
```

```
  { Nil → True
  ; Cons y ys → False }
```

I also made three changes to data type declarations:

1. There is no need to declare empty data types (*i.e.,* data types that have no data constructors),
2. There is no need to specify the arity of the type constructor when declaring a new data type, and
3. There is no need to explicitly quantify type variables in a data constructor declaration.

For example, here is how I would declare length-indexed lists (Figure 3.1, p. 58) for the Algorithm $\mathcal{P}$ implementation:

```
data L where
  { Nil  :: L Z a
  ; Cons :: a → L n a → L (S n) a }
```

Note that declarations for the empty `S` and `Z` data types have disappeared, the declaration for `L` now says `data L` instead of `data L n a`, and the explicit `forall` quantification in the declared types of `Nil` and `Cons` is now implicit.

The algorithm implementation provides built-in support for natural numbers and pairs; programmers are responsible for declaring all other (generalized) algebraic data types as necessary. Figure 7.1 (p. 188) defines seven data types that appear throughout the examples in this chapter. Note that the Haskell list data type `[a]` is now named (`La a`) with data constructors `N` and `C`, and the Haskell `Either` type is now named `E` with data constructors `L` and `R`. All other types in the figure appear in either the Haskell 98 standard prelude [14, §8] or an earlier chapter of this dissertation, so I list them without further explanation.

```
data Bool where     -- Boolean data type
  { True :: Bool ; False :: Bool }

data Maybe where    -- Option data type
  { Nothing :: Maybe a
  ; Just    :: a → Maybe a }

data Ord where      -- Ordering relation data type
  { LT :: Ord ; EQ :: Ord ; GT :: Ord }

data E where        -- Tagged union (sum) data type
  { L :: a → E a b
  ; R :: b → E a b }

data La where       -- Standard homogeneous list (ADT)
  { N :: La a
  ; C :: a → La a → La a }

data L where        -- Length-indexed list (GADT)
  { Nil  :: L Z a
  ; Cons :: a → L n a → L (S n) a }

data Term where     -- Tagless term expressions
  { RepInt  :: Int → Term Int
  ; RepBool :: Bool → Term Bool
  ; RepCond :: Term Bool → Term a → Term a → Term a
  ; RepSnd  :: Term (a, b) → Term b
  ; RepPair :: Term a → Term b → Term (a, b) }
```

Figure 7.1: Basic data types in type inference examples.

```
-- forall a b. L (S a) b → b
head e = case e of
  { Cons x xs → x }

-- forall a b. L (S a) b → L a b
tail e = case e of
  { Cons x xs → xs }

-- forall a b. L a b → Bool
null e = case e of
  { Nil → True
  ; Cons x xs → False }
```

Figure 7.2: Non-Dependent GADT type inference examples.

7.2 NON-DEPENDENT PROGRAMS

In this section, I focus on Non-Dependent GADT programs, which are about the simplest GADT programs of all. Figure 7.2 (p. 189) shows three functions that operate on length-indexed lists (*cf.* Figure 4.1, p. 94). The Haskell-style comment above each example is not part of the example; I use the comments to show the results produced by the implementation. If type inference succeeds (which is the case for all three examples in Figure 7.2), the comment above each example lists the type inferred for the example. If type inference fails (as in Figure 7.6, p. 194), the comment shows the error message.

Algorithm $\mathcal{P}$ successfully infers types for the Non-Dependent GADT programs (`head`, `tail`, and `null`) in Figure 7.2. Due to scrutinee type specialization (§6.2), Algorithm $\mathcal{P}$ does not infer the most-general type for `head`. The following type of `head` is strictly more general:

```
head :: forall a b. L a b → b
```

```
-- forall a. Term a → Maybe Int
term3 e = case e of
  { RepInt i → Just 3
  ; RepBool b → Nothing }

-- forall a b. Term a → La b → La b
repId e x = case e of
  { RepInt i → x
  ; RepBool b → N }
```

Figure 7.3: Type inference without principal types.

Even though the inferred type of `head` is less general than the one I listed here, the inferred type is *better* because it prevents programmers from applying `head` to `Nil` and causing a runtime pattern-matching failure. The `head` example echoes the principle for deciding the specificity of a scrutinee type (§6.2): the scrutinee type should be specific where the pattern types are specific, and it should be general where the pattern types are general.

7.2.1 Type refinement aversion

Figure 7.3 (p. 190) shows two Non-Dependent GADT programs that also have additional types in the plain GADT type system. In other words, a type inference algorithm can choose to infer either a type that requires GADT type refinements or a type that does not. I already described `repId` in some detail in §2.4, and here are two types of `term3`:

```
term3 :: forall a. Term a → Maybe a
term3 :: forall a. Term a → Maybe Int
```

Note that the two types are not instances of each other, and that only the first of the two types requires GADT type refinements. Algorithm $\mathcal{P}$ is, by design, *refinement-averse* (§6.3): it avoids applying GADT type refinements unless there is a clear need for them. As Figure 7.3 shows, Algorithm $\mathcal{P}$ infers the second of the two types for `term3` and the most-general Non-Dependent GADT type for `repId` (whose other maximal types all require GADT type refinements; see §4.1). Refinement aversion helps Algorithm $\mathcal{P}$ choose between competing maximal types in a way that is easy for programmers to understand.

Figure 7.4 (p. 192) shows a Non-Dependent GADT program `rotate` that rotates a red-black tree by recombining its three subtrees differently according to the two `Dir` arguments. Refinement aversion helps Algorithm $\mathcal{P}$ infer a type for `rotate` that introduces no local type information in the `RNode` branch, which is what programmers expect for this function.

7.2.2 Type equality witnesses

There are, however, some programs for which refinement aversion does not work very well. Figure 7.5 (p. 193) shows the `Equ` data type, whose only value `Refl` witnesses that the two type arguments of `Equ` are identical (*cf.* the `Equal` data type in §2.3). Programs that manipulate type-equality witnesses, such as the functions `equ1` and `equ2` in Figure 7.5, are useful mostly because of their types:

```
equ1 :: forall a b. Equ a b → a → b
equ2 :: forall a b. Term a → Term b → Maybe (Equ a b)
```

Under these types, `equ1` converts between values of (witnessed) equal types, and `equ2` returns a type-equality witness if its two arguments represent object-level values with the same (object-level) type. In contrast, the types that Algorithm $\mathcal{P}$ infers for these functions are much less useful: `equ1` is an identity function with an extra argument, and `equ2` returns a trivial witness of reflexivity (*i.e.,* a type is equal to itself).

```
data RoB where
  { Leaf  :: RoB Black Z
  ; RNode :: RoB Black n → Int → RoB Black n → RoB Red n
  ; BNode :: RoB cL m → Int → RoB cR m → RoB Black (S m) }

data Dir where
  { LeftD :: Dir ; RightD :: Dir }

-- forall a. Dir → Int → RoB Black a → Dir → Int →
--           RoB Black a → RoB Red a → RoB Black (S a)
rotate dir1 pE sib dir2 gE uncle tree = case tree of
  { RNode x e y → case dir1 of
    { RightD → case dir2 of
      { RightD → BNode (RNode x e y) pE (RNode sib gE uncle)
      ; LeftD → BNode (RNode uncle gE x) e (RNode y pE sib) }
    ; LeftD → case dir2 of
      { RightD → BNode (RNode sib pE x) e (RNode y gE uncle)
      ; LeftD → BNode (RNode uncle gE sib) pE (RNode x e y) } } }
```

Figure 7.4: This figure shows a red-black-tree rotation function `rotate` and the type Algorithm $\mathcal{P}$ infers for the function. The `rotate` function is adapted from the 2007 summer school notes by Sheard and Linger [38, Appendix A].

```
data Equ where
  { Refl :: Equ a a }

-- forall a b. Equ a a → b → b
equ1 e x = case e of
  { Refl → x }

-- forall a b c. Term a → Term b → Maybe (Equ c c)
equ2 x y = case x of
  { RepInt i → case y of
    { RepInt j → Just Refl
    ; RepBool b → Nothing }
  ; RepBool c → case y of
    { RepInt j → Nothing
    ; RepBool b → Just Refl } }
```

Figure 7.5: Type inference for type equality witnesses.

It is hard to design an algorithm that infers useful types for `equ1` and `equ2` because there are so many choices. Here I list three more for `equ1`:

```
equ1 :: forall a b. Equ [a] [b] → a → b
equ1 :: forall a b c. Equ a b → (a → Int) → b → Int
equ1 :: forall a b c. Equ a b → Equ b c → Equ a c
```

This list of three types merely scratches the surface of all useful types that `equ1` may have. A type inference algorithm may infer one, but probably not three, and definitely not all of the useful types for `equ1`. Therefore, while Algorithm $\mathcal{P}$ can definitely use some improvement, the ultimate limitation lies not in the type inference algorithm, but in the language of types.

```
data K where
  { KInt  :: Int → K Int
  ; KPair :: a → b → K (a, b) }

-- ERROR: A pattern type escapes in equalize
gext7x e = case e of
  { KInt i → Nothing
  ; KPair a b → Just a }

-- ERROR: A branch is unreachable
gext8x e = case e of
  { KInt i → Nothing
  ; KPair a b → Just (a+3) }
```

Figure 7.6: Generalized existential types in type inference.

7.2.3 Generalized existential types

My next examples in Figure 7.6 (p. 194) demonstrate how Algorithm $\mathcal{P}$ deals with escape (`gext7x`) and instantiation (`gext8x`) of generalized existential type variables (§4.2). Algorithm $\mathcal{P}$ *correctly* rejects both (ill-typed) programs, and the different error messages reflect the different ways that it deals with generalized existential type escape and instantiation (§6.2):

Escape Algorithm $\mathcal{P}$ infers a scrutinee type for a branch without considering whether a generalized existential type might escape, and only at the end of `case` expression type inference does it check that no generalized existential type escapes. This wait-and-see approach allows Algorithm $\mathcal{P}$ to infer types for programs that use GADT type refinements to prevent escape, and it also allows Algorithm $\mathcal{P}$ to identify escaped generalized existential types as an individual failure mode. The `gext7x` function demonstrates escape.

Instantiation In contrast, Algorithm $\mathcal{P}$ deals with an instantiated generalized existential type by specializing the scrutinee type (§4.2). I adopted this proactive approach because an instantiated generalized existential type is unlikely to go away without specific remedial action by Algorithm $\mathcal{P}$. This approach always completely eliminates the instantiation problem, but it could cause a different problem later in the type inference process. The `gext8x` function demonstrates instantiation.

Algorithm $\mathcal{P}$ infers `K u` as the `case` scrutinee type for the `gext7x` function (Figure 7.6, p. 194). Since the (generalized existential) type of the variable `a` (introduced by the pattern `KPair a b`) escapes, Algorithm $\mathcal{P}$ rejects `gext7x` with a descriptive message. The `gext8x` function requires instantiating the type of `a` (bound by the pattern `KPair a b`) to `Int`, so Algorithm $\mathcal{P}$ infers `K (Int, v)` as its `case` scrutinee type. Since this scrutinee type is not unifiable with the pattern type `K Int` of the `KInt` branch, Algorithm $\mathcal{P}$ concludes that the `KInt` branch is unreachable and rejects `gext8x` for branch reachability violation.

The examples in this section indicate that Algorithm $\mathcal{P}$ is very effective at type inference for Non-Dependent GADT programs.

7.3 GADT TYPE REFINEMENTS

In addition to Non-Dependent GADT programs, Algorithm $\mathcal{P}$ can also infer types for programs that require GADT type refinements. Due to its refinement-averse nature, Algorithm $\mathcal{P}$ applies GADT type refinements only when it detects one of the following three situations during type inference (§6.3):

1. A generalized existential type variable is about to escape,
2. Contextual type information suggests GADT type refinements, or
3. The body types of two branches form an *opposable thumb*.

```
data Avl where
  { Tip   :: Avl Z
  ; LNode :: Avl n → Int → Avl (S n) → Avl (S (S n))
  ; SNode :: Avl n → Int → Avl n → Avl (S n)
  ; MNode :: Avl (S n) → Int → Avl n → Avl (S (S n)) }

-- forall a. Avl a → Int → Avl (S (S a)) →
--           E (Avl (S (S a))) (Avl (S (S (S a))))
rotl u v w = case w of
  { SNode a x b → R (MNode (LNode u v a) x b)
  ; LNode a x b → L (SNode (SNode u v a) x b)
  ; MNode k y c → case k of
    { SNode a x b → L (SNode (SNode u v a) x (SNode b y c))
    ; LNode a x b → L (SNode (MNode u v a) x (SNode b y c))
    ; MNode a x b → L (SNode (SNode u v a) x (LNode b y c)) } }
```

Figure 7.7: This figure shows the type Algorithm $\mathcal{P}$ infers for the `rotl` function, which performs a left-rotation on a balance-indexed AVL tree. The `rotl` function is adapted from the 2007 Summer School Notes by Sheard and Linger [38, §4.1].

In this section, I demonstrate how these situations help Algorithm $\mathcal{P}$ apply GADT type refinements during type inference.

7.3.1 Generalized existential type escape

Type inference for the `rotl` function in Figure 7.7 (p. 196) demonstrates how Algorithm $\mathcal{P}$ uses the first situation to help it apply GADT type refinements. GADT type refinements are important in this example because they allow the `rotl` function to have a more general type. Without GADT type refinements, the most-general type of `rotl` would be as follows:

```
rotl :: forall a. Avl (S a) → Int → Avl (S (S (S a))) →
                  E (Avl (S (S (S a)))) (Avl (S (S (S (S a)))))
```

This refinement-free type of `rotl` states that its first argument must be a tree of height one (or greater), and its second argument must be a tree of height three (or greater). In contrast, the type Algorithm $\mathcal{P}$ infers for `rotl` (Figure 7.7) is better because it allows `rotl` to accept shorter trees as arguments.

Algorithm $\mathcal{P}$ applies GADT type refinements while inferring a type for `rotl` to prevent a generalized existential type from escaping. Consider the inner-most `MNode` branch, which appears in the last line of the `rotl` function. Here are the types Algorithm $\mathcal{P}$ infers for the branch:

- Branch pattern type: `Avl (S (S m))`
- Branch scrutinee type: `Avl (S k)`
- Branch body type: `Avl (S (S (S m)))`

By Definition 6 (p. 104), the type variable `m` is a generalized existential type. Since `m` also appears in the branch body type, Algorithm $\mathcal{P}$ must take action to prevent it from escaping. As luck would have it, this branch induces the type refinement [`S m`/`k`]. So, to capture `m`, Algorithm $\mathcal{P}$ applies the type refinement and rewrites the branch body type into `Avl (S (S k))`.

This example shows how possible escape of generalized existential types helps Algorithm $\mathcal{P}$ apply GADT type refinements.

7.3.2 Contextual type information

Type inference for the `fdFun` function in Figure 7.8 (p. 198) demonstrates how Algorithm $\mathcal{P}$ uses the second situation to help it apply GADT type refinements. GADT type refinements are important in this example because they allow the

```
data FunDesc where
  { FDI :: FunDesc a a
  ; FDC :: b → FunDesc a b
  ; FDG :: (a → b) → FunDesc a b }

-- forall a b. FunDesc a b → a → b
fdFun e = case e of
  { FDI → λx → x
  ; FDC b → λx → b
  ; FDG f → f }
```

Figure 7.8: This figure shows the type Algorithm $\mathcal{P}$ infers for the `fdFun` function, which evaluates a function description arrow to a function. The `fdFun` function is adapted from the 2005 paper on functional reactive programming optimization by Nilsson [30, §4.1].

`fdFun` function to have a more general type. Without GADT type refinements, the most-general type of `fdFun` would be as follows:

```
fdFun :: forall a. FunDesc a a → a → a
```

This refinement-free type restricts `fdFun` to function descriptions with the same domain and range, so it is less general than the type inferred by Algorithm $\mathcal{P}$.

The `FDI` branch in `fdFun` is essentially the same as the `Refl` branch in `equ1` (Figure 7.5, p. 193), and, as I explained earlier, the right type for such a branch really depends on the intention of the programmer. As luck would have it, the `FDG` branch reveals the programmer's intention with its type:

```
FunDesc a b → a → b
```

Using this piece of type information, which originates completely outside of the `FDI` branch, helps Algorithm $\mathcal{P}$ to narrow down the type for the `FDI` branch and successfully infer the expected type for the `fdFun` function.[1]

This example shows how type information from the context of a branch helps Algorithm $\mathcal{P}$ to apply GADT type refinements.

7.3.3 Opposable thumbs

Type inference for the `gadt1` function in Figure 7.9 (p. 200) demonstrates how Algorithm $\mathcal{P}$ uses the third situation to help it apply GADT type refinements. GADT type refinements are important in this example because `gadt1` does not have a type in the Non-Dependent GADT type system.

Algorithm $\mathcal{P}$ applies GADT type refinements while inferring a type for `gadt1` because it detects an *opposable thumb*: an inconsistency between the body types of two branches. Consider the body types of the two branches in `gadt1`:

- Branch body type (`Nil`): `L Z a`
- Branch body type (`Cons`): `L (S n) b`

Since these two branch body types are inconsistent (*i.e.,* they are not unifiable), a type of `gadt1` must use GADT type refinements to reconcile the inconsistency. The opposable thumb makes Algorithm $\mathcal{P}$ overcome its refinement-aversion and apply GADT type refinements to infer a type for `gadt1`.

The functions `term1` and `term7` (Figure 7.9, p. 200) also demonstrate how opposable thumbs make Algorithm $\mathcal{P}$ apply GADT type refinements during type inference. The opposable thumb in `term1` is pretty obvious:

[1] The OutsideIn algorithm by Schrijvers et al. [36] uses this same approach to infer the types of GADT programs. It is no coincidence that `fdFun` is the only program (out of 30 well-typed programs) in this chapter whose type the OutsideIn algorithm can successfully infer without programmer type annotations.

```
-- forall a b. L a b → L a b
gadt1 e = case e of
  { Nil → Nil
  ; Cons x xs → Cons x xs }

-- forall a. Term a → a
term1 e = case e of
  { RepInt i → i
  ; RepBool b → b }

-- forall a b. Term a → (a → Maybe b) → Maybe b
term7 e f = case e of
  { RepInt i → f 3
  ; RepBool b → f True
  ; RepPair u v → Nothing }
```

Figure 7.9: Type inference with GADT type refinements.

- Branch body type (`RepInt`): `Int`
- Branch body type (`RepBool`): `Bool`

Just as in `gadt1`, this inconsistency forces Algorithm $\mathcal{P}$ to apply GADT type refinements to infer a type for `term1`. The situation for `term7` is slightly more interesting because it involves three branches and types in the environment. Consider the inferred type of `f` in each of the three branches in `term7`:

- Type of `f` (in `RepInt` branch): `Int → c`
- Type of `f` (in `RepBool` branch): `Bool → d`
- Type of `f` (in `RepPair` branch): `x`

Note that the type of `f` is totally unconstrained in the `RepPair` branch because `f` does not appear in that branch. Even though the type of `f` in the `RepPair` branch is consistent with any type, the types of `f` in the `RepInt` and `RepBool` branches are inconsistent, and this single inconsistency is enough to form an opposable thumb. As a result, Algorithm $\mathcal{P}$ applies GADT type refinements and successfully infers a type for the `term7` function.

In this section, I demonstrated that Algorithm $\mathcal{P}$ can indeed infer types for programs that require GADT type refinements and that it can do so without programmer type annotations. In the next section, I will present more program examples that require not only GADT type refinements but also polymorphic recursion.

7.4 POLYMORPHIC RECURSION

Polymorphic recursion is quite common in recursive GADT programs because a GADT data constructor may have a non-uniform range type (§2.3). Since GADT type arguments can reflect the structure of a value, even structural recursion over a generalized algebraic data type may require polymorphic recursion. Therefore a GADT type inference algorithm must provide support for polymorphic recursion to be practically useful.

Instead of developing a type inference algorithm specifically for polymorphic recursion over GADT data values, I adopted an iterative algorithm proposed by Mycroft to infer the types of general polymorphic recursive functions (§6.1). Type inference with polymorphic recursion is undecidable [12, 21], so I restrict this algorithm to only a small number of iterations[2] to ensure termination.

Figure 7.10 (p. 202) lists three recursive GADT programs (length-indexed list

[2] I used 20 iterations as the limit in my implementation. However, experiment shows that even a limit of 3 iterations is sufficient for Algorithm $\mathcal{P}$ to produce the same result for all programs in my test suite.

```
-- forall a b. L a b → Int
length l = case l of
  { Nil → 0
  ; Cons x xs → 1 + length xs }
-- forall a b c. (a → b) → L c a → L c b
map f l = case l of
  { Nil → Nil
  ; Cons x xs → Cons (f x) (map f xs) }
-- forall a. Term a → a
eval4 x = case x of
  { RepInt i → i
  ; RepBool b → b
  ; RepCond u a b → case eval4 u of
    { True → eval4 a
    ; False → eval4 b }
  ; RepSnd u → case eval4 u of { (x, y) → y }
  ; RepPair a b → (eval4 a, eval4 b) }
```

Figure 7.10: This figure shows the types Algorithm $\mathcal{P}$ infers for three functions that conduct polymorphic recursion on length-indexed lists (`length`, `map`) and on tagless term representations (`eval4`).

length, length-indexed list map, and a tagless term interpreter eval4). Although the type inference algorithm for polymorphic recursion is not specifically tailored for GADT programs, it still performs quite well and successfully infers the types that programmers expect of these programs.

The Mycroft algorithm performs well because these programs contain so much redundant type information that Algorithm $\mathcal{P}$ does not need to rely on recursive references for type information. For example, let me take the map function and replace its recursive reference with an undefined expression:

```
map f l = case l of
  { Nil → Nil
  ; Cons x xs → Cons (f x) (undefined f xs) }
```

Even though the recursive reference has been erased, the branch body types still exhibit an opposable thumb, which enables Algorithm $\mathcal{P}$ to apply GADT type refinements and infer a type for map. Since the recursive reference is not essential to type inference, the Mycroft algorithm can incrementally specialize the inferred type for map without causing type inference failure for the case expression.

Luckily, this phenomenon (the abundance of type information) appears to be common among recursive GADT programs. This abundance allows Algorithm $\mathcal{P}$ to infer types for many different programs:

- Generic N-way zip (Figure 7.11, p. 204),
- Shape-indexed tree search and extraction (Figure 7.12, p. 205),
- Reified state monad (Figure 7.13, p. 206), and
- AVL-tree node insertion (Figure 7.14, p. 207).

The examples in this section demonstrate that, although type inference with polymorphic recursion is generally undecidable, Algorithm $\mathcal{P}$ can still support

```
data Zip2 where
  { Zero2 ::  Zip2 u (La u)
  ; Succ2 ::  Zip2 u v → Zip2 (w → u) (La w → v) }

-- forall a b. Zip2 a b → a → b
zip2 n f =
  let { z2 = zipS n f z2
      ; zipZ e = case e of
          { Zero2 → N
          ; Succ2 n → λys → zipZ n }
      ; zipS e f r = case e of
          { Zero2 → C f r
          ; Succ2 n → λys → case ys of
            { N → zipZ n
            ; C z zs → zipS n (f z) (r zs) } } }
  in z2
```

Figure 7.11: This figure shows the type Algorithm $\mathcal{P}$ infers for the `zip2` function, which implements generic N-way zip for lists. The `zip2` function is adapted from the 2006 Spring School Notes by Sheard [37, §5.3].

```
data Tree where
  { End  :: a → Tree Nd a
  ; Fork :: Tree u a → Tree v a → Tree (Fk u v) a }

data Path where
  { Here  :: Path Nd
  ; ForkL :: Path x → Path (Fk x y)
  ; ForkR :: Path y → Path (Fk x y) }

-- forall a b. (a → Bool) → Tree b a → La (Path b)
find f t = case t of
  { End m → case f m of
    { True → C Here N
    ; False → N }
  ; Fork x y → append (map ForkL (find f x))
                       (map ForkR (find f y)) }

-- forall a b. Path a → Tree a b → b
extract p t = case p of
  { Here → case t of
    { End m → m }
  ; ForkL p1 → case t of
    { Fork x y → extract p1 x }
  ; ForkR p1 → case t of
    { Fork x y → extract p1 y } }
```

Figure 7.12: This figure shows the types Algorithm $\mathcal{P}$ infers for the `find` and the `extract` function. The former returns shape-indexed tree paths from a search, and the latter extracts a value at a specific path in a tree. These functions are adapted from the 2007 Summer School Notes by Sheard and Linger [38, §3.3].

```
data State where
  { Bind   :: State s a → (a → State s b) → State s b
  ; Return ::  a → State s a
  ; Get    ::  State s s
  ; Put    ::  s → State s () }

-- forall a b. State a b → a → (a, b)
runState e s = case e of
  { Return a → (s, a)
  ; Get → (s, s)
  ; Put u → (u, ())
  ; Bind m k → case runState m s of
    { (s1, a1) → runState (k a1) s1 } }
```

Figure 7.13: This figure shows the type Algorithm $\mathcal{P}$ infers for `runState`, which evaluates a computation in a reified state monad (*cf.* Figure 7.22, p. 217).

polymorphic recursion in a practical setting. This conclusion echoes Henglein's observation that theoretical intractability of type inference problems need not affect the practical utility of type inference algorithms [12, §6]. Now that I have demonstrated the capabilities of Algorithm $\mathcal{P}$, I will end this chapter by discussing some of its limitations through a few more examples in the next section.

7.5 ALGORITHM LIMITATIONS

Even though Algorithm $\mathcal{P}$ is quite powerful, it is still incomplete: there are some well-typed practical GADT programs whose types it cannot infer. In this section, I discuss four limitations of Algorithm $\mathcal{P}$ and demonstrate them with examples.

```
-- forall a. Int → Avl a → E (Avl a) (Avl (S a))
ins i t = case t of
  { Tip → R (SNode Tip i Tip)
  ; SNode a x b → case compare i x of
    { EQ → L t
    ; LT → case ins i a of
      { L a → L (SNode a x b)
      ; R a → R (MNode a x b) }
    ; GT → case ins i b of
      { L b → L (SNode a x b)
      ; R b → R (LNode a x b) } }
  ; LNode a x b → case compare i x of
    { EQ → L t
    ; LT → case ins i a of
      { L a → L (LNode a x b)
      ; R a → L (SNode a x b) }
    ; GT → case ins i b of
      { L b → L (LNode a x b)
      ; R b → rotl a x b } }
  ; MNode a x b → case compare i x of
    { EQ → L t
    ; LT → case ins i a of
      { L a → L (MNode a x b)
      ; R a → rotr a x b }
    ; GT → case ins i b of
      { L b → L (MNode a x b)
      ; R b → L (SNode a x b) } } }
```

Figure 7.14: This figure shows the type Algorithm $\mathcal{P}$ infers for the `ins` function, which adds a node to a balance-indexed AVL tree. This function uses the `Avl` data type and the `rotl` function from Figure 7.7 (p. 196). The `ins` function is adapted from the 2007 Summer School Notes by Sheard and Linger [38, §4.1].

```
data Deg where
  { Fd :: Int → Deg F
  ; Kd :: Int → Deg K
  ; Cd :: Int → Deg C }

-- forall a. Deg a → Deg a → Deg a
plus a b = case a of
  { Fd u → case b of { Fd v → Fd (u+v) }
  ; Kd u → case b of { Kd v → Kd (u+v) }
  ; Cd u → case b of { Cd v → Cd (u+v) } }
```

Figure 7.15: This figure shows the type Algorithm $\mathcal{P}$ infers for the `plus` function, which adds two temperature measures in the same unit. The `plus` function is adapted from the 2006 Spring School Notes by Sheard [37, §4].

7.5.1 Local type reconciliation

The first limitation of Algorithm $\mathcal{P}$ is the design decision to reconcile the branch body types in a `case` expression directly in the scope of the `case` expression. In other words, Algorithm $\mathcal{P}$ reconciles the branch body types without considering other branches that appear elsewhere in the program. This design appears to work quite well, and it is powerful enough to link GADT type indices in nested `case` expressions. For example, Algorithm $\mathcal{P}$ infers that `plus` (Figure 7.15, p. 208) adds only temperatures in the same unit, and that `zipWith` (Figure 7.16, p. 209) zips only lists of equal length (into a list also of equal length).

This decision to reconcile branch body types locally also helps Algorithm $\mathcal{P}$ reject some well-typed plain GADT programs that may trigger runtime pattern-matching failures due to missing branches. The `gadt7o` function in Figure 7.17 (p. 209) is one such example. It has this type in the plain GADT type system:

```
-- forall a b c d. (a → b → c) → L d a → L d b → L d c
zipWith f a b = case a of
  { Nil → case b of
    { Nil → Nil }
  ; Cons x xs → case b of
    { Cons y ys → Cons (f x y) (zipWith f xs ys) } }
```

Figure 7.16: Type inference for length-indexed `zipWith`.

```
-- ERROR: Cannot unify different type constructors
gadt7o e =
  ( case e of { Nil → True },
    case e of { Cons x xs → False } )
```

Figure 7.17: Type inference for clashing `case` expressions.

```
gadt7o :: forall a b. L a b → (Bool, Bool)
```

However, regardless of what argument one applies `gadt7o` to, the function always returns a pair with one diverging component. If the argument is `Nil`, the second component of the pair diverges due to pattern-matching failure; otherwise the first component diverges. Algorithm $\mathcal{P}$ cannot infer a type for `gadt7o` because it infers two inconsistent types for `e` (`L Z a` and `L (S n) b`) separately from the two `case` expressions in the program. The `gadt7o` function shows that Algorithm $\mathcal{P}$ is incomplete, but programmers may welcome this incompleteness because they want to avoid writing programs that trigger pattern-matching failures.

My next example on local type reconciliation, in contrast, comes from a practical application and does not diverge. Figure 7.18 (p. 210) shows the `delmin_o`

```
data Zero where
  { IsZ :: Zero Z ; NotZ :: Zero (S n) }

-- forall a. Avl a → Zero a
empty t = case t of
  { Tip → IsZ
  ; LNode a x b → NotZ
  ; SNode a x b → NotZ
  ; MNode a x b → NotZ }

-- ERROR: A pattern type escapes in equalize
delmin_o t = case t of
  { LNode a x b → case empty a of
    { IsZ → (x, L b)
    ; NotZ → case delmin_o a of
      { (y, k) → case k of
        { L a → (y, rotl a x b)
        ; R a → (y, R (LNode a x b)) } } }
  ; SNode a x b → case empty a of
    { IsZ → (x, L b)
    ; NotZ → case delmin_o a of
      { (y, k) → case k of
        { L a → (y, R (LNode a x b))
        ; R a → (y, R (SNode a x b)) } } }
  ; MNode a x b → case delmin_o a of
    { (y, k) → case k of
      { L a → (y, L (SNode a x b))
      ; R a → (y, R (MNode a x b)) } } }
```

Figure 7.18: This figure shows the Algorithm $\mathcal{P}$ type inference error message for the delmin_o function, which removes the left-most node from a balance-indexed AVL tree. The delmin_o and the empty functions use the Avl data type and the rotl function from Figure 7.7 (p. 196). They are adapted from the 2007 Summer School Notes by Sheard and Linger [38, §4.1].

function, which deletes the left-most internal node of a non-empty AVL tree. It has the following type in the plain GADT type system:

```
delmin_o :: forall a. Avl (S a) → (Int, E (Avl a) (Avl (S a)))
```

If removing the left-most node reduces the height of the tree by one level, the `delmin_o` function returns the new tree using the left injection `L`. If the height of the tree remains unchanged, it returns the result using the right injection `R`.

Type inference for `delmin_o` fails because Algorithm $\mathcal{P}$ took a wrong turn while inferring the type of the second `IsZ` branch (which is inside the `SNode` branch). Here, since `empty a` matches `IsZ`, `a` must be `Tip`, and the type of `SNode` (Figure 7.7, p. 196) requires `b` to be `Tip` as well. If Algorithm $\mathcal{P}$ tries to reconcile all pattern-matching branches in `delmin_o` at the same time, it would see that the type constructor `Z` in the type `Avl Z` of `b` requires GADT type refinement (because of opposable thumbs from left-injections in other branches). In this inner `case` expression, however, there are no other left injections (removing a node from a perfectly-balanced tree never decreases its height), so refinement-aversion prevents Algorithm $\mathcal{P}$ from applying a GADT type refinement to the type of `b`. This decision ultimately leads to type inference failure for `delmin_o`.

7.5.2 Compositionality

The second limitation of Algorithm $\mathcal{P}$ is its compositional structure (§6.1), which requires type inference for a pattern-matching branch to use only type information from the branch. The `fdComp1` function in Figure 7.19 (p. 212) illustrates the problem with compositional type inference. It has the following most-general type in the plain GADT type system:

```
fdComp1 :: forall a b c. FunDesc a b → FunDesc b c → FunDesc a c
```

My specific implementation of Algorithm $\mathcal{P}$ infers a less general type for `fdComp1` because it makes a wrong guess. The following situation arises when Algorithm $\mathcal{P}$

```
-- forall a b. FunDesc a b → FunDesc b b → FunDesc a b
fdComp1 fd1 fd2 =
  let { o f g x = f (g x) }
  in case fd1 of
    { FDI → fd2
    ; FDC b → case fd2 of
      { FDI → fd1
      ; FDC c → FDC ((fdFun fd2) b)
      ; FDG g → FDC ((fdFun fd2) b) }
    ; FDG f → case fd2 of
      { FDI → fd1
      ; FDC c → FDC c
      ; FDG g → FDG (o (fdFun fd2) f) } }
```

Figure 7.19: This figure shows the suboptimal type Algorithm $\mathcal{P}$ infers for the `fdComp` function, which composes two function description arrows. This function uses the `FunDesc` data type and the `fdFun` function from Figure 7.8 (p. 198). The `fdComp` function is adapted from the 2005 paper on functional reactive programming optimization by Nilsson [30, §4.1].

tries to infer a type for fd1 in the second inner case expression (in the last four lines of fdComp1):

- Branch pattern type (FDI): FunDesc x x
- Branch scrutinee type (FDI): FunDesc b c
- Type of fd1 (in FDI branch): FunDesc a x

Since the type variables b and c both refine to x in the FDI branch, there are two ways to type fd1 outside the scope of the FDI branch:

```
fd1 :: FunDesc a b
fd1 :: FunDesc a c
```

Choosing the first type would allow Algorithm $\mathcal{P}$ to infer the most-general type for fdComp1. Unfortunately, my implementation (arbitrarily) picks the second type, and this wrong guess unifies the type variables b and c, so my implementation ends up inferring the less general type shown in Figure 7.19 (p. 212).

The problem is not that my implementation made the wrong guess. Instead, the guess should not have been necessary in the first place: the variable f, which is bound in the FDG f pattern, has the unambiguous type a → b. Since fd1 matches the FDG f pattern in this part of the program, there should be no ambiguity that the type of fd1 is FunDesc a b. However, due to its compositional design, Algorithm $\mathcal{P}$ cannot access type information about the case scrutinee when inferring a type for a pattern-matching branch body. Thus it must guess the type of fd1, and the wrong guess ultimately results in an inferred type that is suboptimal.

7.5.3 Lack of backtracking search

I now move on to the third limitation: Algorithm $\mathcal{P}$ does not use backtracking search when reconciling the body types of different branches. In other words, it

```
data Zip1 where
  { Zero1 ::  Zip1 (La u) (La u)
  ; Succ1 ::  Zip1 (La u) v → Zip1 (La (w → u)) (La w → v) }

-- ERROR: Cannot unify different type constructors
zip1o n f =
  let { apply f x = f x
      ; z1 n1 fs = case n1 of
          { Zero1 → fs
          ; Succ1 n2 → λxs → z1 n2 (zipWith apply fs xs) } }
  in z1 n (repeat f)
```

Figure 7.20: This figure shows the Algorithm $\mathcal{P}$ type inference error message for the `zip1o` function, which implements generic N-way zip for lists (*cf.* `zip2`, Figure 7.11, p. 204). The `zip1o` function is adapted from the 2006 Spring School Notes by Sheard [37, §5.3].

relies on specific conditions (such as opposable thumbs, §7.3) to identify where GADT type refinements are necessary, and what are the appropriate type indices to use. This simplified design is sufficient for many programs because GADT `case` expressions typically contain enough structure to create these specific conditions. However, there is no guarantee that this expectation holds for every program, and Figure 7.20 (p. 214) shows an example where it fails.

The `zip1o` function, which also implements N-way zip (*cf.* `zip2`, Figure 7.11, p. 204), has the following type in the plain GADT type system:

```
zip1o :: forall a b. Zip1 (La a) b → a → b
```

Type inference for `zip1o` fails because Algorithm $\mathcal{P}$ cannot decide the type of `fs` in the local `let` definition `z1`. The problem arises because a single type (which I call `w`) in the `Zero1` branch corresponds to two inconsistent types in the `Succ1` branch. Here is how `w` appears in the `Zero1` branch:

- Body type of the `Zero1` branch: `w`
- Type of `fs` in the `Zero1` branch: `w`

And here are the corresponding types in the `Succ1` branch:

- Body type of the `Succ1` branch: `La x → y`
- Type of `fs` in the `Succ1` branch: `La (x → u)`

Note that the first type is a function, and the second type is a list. The type `w` in the `Zero1` branch is consistent with either of the two corresponding types in the `Succ1` branch, so there is no opposable thumb. However, since a function type is inconsistent with a list type, the type correspondence between the two branches entails an indirect inconsistency. Without backtracking search, Algorithm $\mathcal{P}$ cannot reconcile this indirect inconsistency, and therefore type inference for `zip1o` fails with a unification error.

7.5.4 Polymorphic recursion

The last limitation concerns functions that require polymorphic recursion. Since type inference with polymorphic recursion is undecidable, and Algorithm $\mathcal{P}$ supports polymorphic recursion with an iterative algorithm (§6.1, §7.4), there must be programs for which type inference fails due to reaching the iteration limit. The `leq_o` function in Figure 7.21 (p. 216), which computes a witness that one natural number is less than or equal to another, is one such program. It has the following infinite set of types in the plain GADT type system:

```
data Nat where
  { Zn :: Nat Z
  ; Sn :: Nat n → Nat (S n) }

data NatLeq where
  { LeZ :: NatLeq Z b
  ; LeS :: NatLeq a b → NatLeq (S a) (S b) }

-- ERROR: Mycroft iteration limit reached
leq_o k = case k of
  { Zn → LeZ
  ; Sn n → LeS (leq_o n) }
```

Figure 7.21: This figure shows the Algorithm $\mathcal{P}$ type inference error message for the `leq_o` function, which computes a witness for the ordering relation of two natural numbers. The `leq_o` function is adapted from the 2007 summer school notes by Sheard and Linger [38, §3.7].

```
leq_o :: forall a. Nat a → NatLeq a a
leq_o :: forall a. Nat a → NatLeq a (S a)
leq_o :: forall a. Nat a → NatLeq a (S (S a))
                ⋮
```

Algorithm $\mathcal{P}$ fails to infer a type for `leq_o` because the `NatLeq` data type and the `leq_o` program lack sufficient structure to show how the two type arguments of `NatLeq` should be related. As a result, polymorphic recursion type inference in Algorithm $\mathcal{P}$ generates the following sequence of types, none of them valid, before reaching the iteration limit:

```
leq_o :: forall a b. Nat a → NatLeq a (S b)
```

```
-- ERROR: A pattern type escapes in equalize
runState_o e s = case e of
  { Return a → (s, a)
  ; Get → (s, s)
  ; Put u → (u, ())
  ; Bind m k → case m of
    { Return a → runState_o (k a) s
    ; Get → runState_o (k s) s
    ; Put u → runState_o (k ()) u
    ; Bind n j → runState_o (Bind n (λx. Bind (j x) k)) s } }
```

Figure 7.22: This figure shows the Algorithm $\mathcal{P}$ type inference error message for the `runState_o` function, which evaluates a computation in a reified state monad. The `runState_o` function uses the `State` data type from Figure 7.13 (p. 206).

```
leq_o :: forall a b. Nat a → NatLeq a (S (S b))
leq_o :: forall a b. Nat a → NatLeq a (S (S (S b)))
                ⋮
```

This example suggests that type inference with polymorphic recursion works well only for programs with sufficient structure, so that Algorithm $\mathcal{P}$ knows how to apply GADT type refinements to the types that vary in recursive invocations.

My last example is the `runState_o` function from Figure 7.22 (p. 217), which expands the evaluation of `Bind` by analyzing its first argument with a nested `case` expression (*cf.* `runState`, Figure 7.13, p. 206). It has the same type as `runState` does in the plain GADT type system:

```
runState_o :: forall a b. State a b → a → (a, b)
```

Type inference for `runState_o` fails because Algorithm $\mathcal{P}$ cannot find a GADT type refinement for the variable `s` in the inner `case` expression against `m`. The variable `s` appears four times in this `case` expression:

- As the second argument of `runState_o` in the `Return` branch,
- As the second argument of `runState_o` in the `Get` branch,
- As the argument of `k` in the `Get` branch, and
- As the second argument of `runState_o` in the `Bind` branch.

To infer the type of `runState_o` using Mycroft's algorithm, Algorithm $\mathcal{P}$ starts by assuming that `runState_o` has a fully polymorphic type (*i.e.,* `forall a. a`) and then gradually specializes this assumed type. Due to this initial condition, all but the third occurrence of `s` provide no useful type information to Algorithm $\mathcal{P}$ in the first Mycroft iteration. Assuming that `m` has type `State a b`, the third occurrence of `m` states only that `s` has type `a` or `b` but not which. Without a way to break the tie, Algorithm $\mathcal{P}$ has no choice but to declare failure. This example suggests that Algorithm $\mathcal{P}$ may have problems with tail-recursive GADT programs that also require polymorphic recursion.

7.6 SUMMARY

In this chapter, I demonstrated Algorithm $\mathcal{P}$ through 32 program examples. The programs come from a wide range of application domains, and they show that Algorithm $\mathcal{P}$ represents a significant step toward practical type inference for plain GADT programs. They also exercise many features of Algorithm $\mathcal{P}$:

- Scrutinee type specialization (`head`, Figure 7.2, p. 189, also §6.2)
- Refinement aversion (Figure 7.3, p. 190, also §6.3)
- Generalized existential type property enforcement (Figure 7.6, p. 194)

- Type refinements through escape prevention (Figure 7.7, p. 196)
- Type refinements through contextual information (Figure 7.8, p. 198)
- Type refinements through opposable thumbs (Figure 7.9, p. 200)
- Polymorphic recursion support (Figure 7.10, p. 202)

Algorithm $\mathcal{P}$ is, of course, not without its faults, and some of the examples help demonstrate common causes of type inference failure for programs that are well-typed in the plain GADT type system:

1. Algorithm $\mathcal{P}$ does not work well with general type equality witness types (Figure 7.5, p. 193) because it lacks sufficient information to apply GADT type refinements. In some situations it resorts to guessing, and sometimes it makes the wrong guesses (Figure 7.19, p. 212).

2. Algorithm $\mathcal{P}$ does not work well when there are type inconsistencies that arise only between pattern-matching branches in different `case` expressions (Figure 7.18, p. 210).

3. Algorithm $\mathcal{P}$ does not work well with nested `case` expressions that refer to variables that have already been matched (Figure 7.19, p. 212) because its compositional design prohibits top-down type information propagation from the `case` scrutinee to the pattern-matching branches.

4. Algorithm $\mathcal{P}$ is not well-equipped to deal with indirect type inconsistencies between branches (Figure 7.20, p. 214) because it does not use backtracking search to reconcile the body types of different branches.

5. Programs that conduct polymorphic recursion in a relatively unstructured fashion (Figure 7.21, p. 216) could cause Algorithm $\mathcal{P}$ to loop (which gets cut off after a small number of iterations).

6. Algorithm $\mathcal{P}$ does not work well with programs that rely on recursive references to convey type information that is not otherwise available (Figure 7.22, p. 217) because Mycroft's algorithm makes the initial assumption that recursive references have fully-polymorphic types (§6.1).

These limitations represent obvious starting points for future work. That said, since Algorithm $\mathcal{P}$ successfully infers types for 25 out of the 30 well-typed plain GADT programs that I presented in this chapter, there is reason to be optimistic that many practical GADT programs will naturally avoid these pitfalls without special accommodation by the programmers.

Chapter 8

CONCLUSION

I started this dissertation research with the goal of designing a practical GADT type inference algorithm that does not require programmer type annotations. The development of this type inference algorithm prompted me to investigate various properties of the plain GADT type system, and this research effort advanced the state of the art in two research areas — GADT type system characterization and GADT type inference. This outcome suggests that the challenges of designing a practical GADT type inference algorithm may not lie in any inherent technical difficulty, but merely our lack of understanding of the GADT type system.

In this chapter, I conclude the dissertation by summarizing my contributions and discussing possible areas for future work.

8.1 CONTRIBUTIONS

This dissertation makes technical contributions in five areas of research that are related to programming languages and type systems:

Research methodology First, this dissertation demonstrates that the *development* of a type inference algorithm can be an effective approach to type system research. I speculate that this effectiveness is due to two reasons. First, the type inference problem, which encourages a systematic exploration of all well-typed programs, helps researchers come up with atypical program examples. These examples, such as the `vary` function in §3.1 (Figure 3.2, p. 62), are instrumental for uncovering unexpected type system properties. Second, a type inference algorithm codifies the intuition that a type is an abstract interpretation of a program, so type inference failures correspond directly to deficiencies in the abstract interpretation. This connection, which I exploited in §4.1, helps researchers identify well-typed programs for which our collective understanding remains incomplete (and are thus worthy of a careful study). It is time for more researchers to start seeing difficult type inference problems for what they are — research opportunities to gain a deeper understanding of type systems.

GADT type system properties Second, in this dissertation I describe three new major properties of the plain GADT type system:

- Generalized existential types provide an extrinsic characterization of how a GADT pattern-matching branch restricts the escape and instantiation of type variables (§4.2),
- The plain GADT type system violates type preservation due to the GADT branch reachability requirement (§5.1), and
- Local `let` definitions can restrict the enforcement of the branch reachability requirement in the plain GADT type system (§5.1).

In addition, I also explained why guarded algebraic data types are not equivalent to other GADT type systems (§2.3), refuted two undecidability claims for the GADT type inference problem (§2.4), described a symmetry in the ALT-GADT type rule for GADT pattern-matching branches (§2.5), showed a pathological plain GADT program whose two GADT pattern-matching branches can have arbitrarily different types (§3.1), and proposed a principle for judging the quality of `case` scrutinee types (§6.2). These discoveries should benefit researchers, educators, and practical programmers who work with GADT programs.

Type system design Third, this dissertation contains multiple examples and discussions on the intricacies of type system design. In addition to presenting two complete type systems (Pointwise GADT, §3.2, and Non-Dependent GADT, §4.1) that restrict the plain GADT type system, I also sketched type system changes that disentangle the two roles of local `let` definitions (type polymorphism and limiting branch reachability enforcement) in the plain GADT type system (§5.1). To illustrate the complications that could arise from a seemingly innocent design decision, I devoted Chapter 5 to the GADT branch reachability requirement. I presented arguments both for and against the requirement, described how the requirement interacts with other (seemingly independent) type system features, explained how the requirement evolves as GADT type systems become more restrictive, and discussed how all these consequences affect the GADT type inference problem. All the discussions combine into a comprehensive example on why type system designers must tread lightly when adding features to (or removing features from) a type system.

Programming principles and practice Fourth, I conjectured that, although the plain GADT type system permits arbitrary type information flow between the scrutinee type and the pattern type of a GADT pattern-matching branch, in practice programmers rarely take advantage of this generality. I supported the conjecture with case studies (§3.3), and I speculated why the conjecture holds in practice. I observed that pointwise type information flow between scrutinee types and pattern types is a consequence of the principle of orthogonal design, and I showed how this principle helps programmers turn a plain GADT program into a Pointwise GADT program that is easier to understand. This connection between type systems and programming principles suggests that cognitive sciences may have a rightful place in future type system research projects.

GADT type inference Finally, in this dissertation I presented Algorithm $\mathcal{P}$, which is a type inference algorithm that is capable of inferring the types of many practical plain GADT programs. I designed Algorithm $\mathcal{P}$ using all the discoveries I presented in this dissertation; the algorithm is, in some sense, a summary of the dissertation as a whole. Algorithm $\mathcal{P}$ is *significantly* more powerful than existing plain GADT type inference algorithms: it successfully infers the types for 25 out of the 30 well-typed programs in Chapter 7, which I selected from a wide range of application domains. In comparison, the OutsideIn algorithm by Schrijvers et al. [36] can infer the type of only one (`fdFun`, Figure 7.9, p. 200) in the absence of programmer type annotations. Algorithm $\mathcal{P}$ proves that, although GADT type inference is difficult in principle, it can also be practically viable by exploiting the internal structures that are common in practical GADT programs.

8.2 FUTURE WORK

Even though Algorithm $\mathcal{P}$ performs significantly better than existing GADT type inference algorithms such as OutsideIn, there are still many aspects of the algorithm that warrant further research. Here are five possible areas for future work on Algorithm $\mathcal{P}$:

Type inference power As the examples in §7.5 demonstrated, Algorithm $\mathcal{P}$ still has problems inferring the types for many practical plain GADT programs. For example, in some cases (such as `fdComp1` in Figure 7.19, p. 212) type inference fails because Algorithm $\mathcal{P}$ does not propagate type information from the context into a GADT `case` expression. In other cases (such as `runState_o` in Figure 7.22, p. 217) type inference fails because type inference for polymorphic recursion in Algorithm $\mathcal{P}$ interferes with the type inference for GADT `case` expression. There is no obvious reason why an algorithm cannot do better in these situations, and research opportunities abound.

Error reporting It is essential for a practical type inference algorithm to produce informative error messages because programmers often need help to find and to fix type errors. It is even more important for an *incomplete* practical type inference algorithm to produce informative error messages because type inference may fail even if the program is well-typed (in which case there is no error to fix). Algorithm $\mathcal{P}$, as presented in Chapter 6, does not include any facility for error reporting, and the Haskell implementation produces only generic error messages. More work is necessary to categorize the failure modes of Algorithm $\mathcal{P}$ and to identify the best ways to explain those errors to the programmer.

Algorithm structuring Following the example of Milner's Algorithm $\mathcal{W}$ [27], Algorithm $\mathcal{P}$ uses explicit type substitutions to represent type information that it discovers during type inference. This traditional design, however, is no longer in fashion: some type inference algorithms represent type substitutions implicitly using mutable variables [16] to achieve greater efficiency, and others forgo type substitutions altogether and instead reduce type inference to constraint solving [36, 40] to improve modularity. More work is necessary to investigate how one can adapt Algorithm $\mathcal{P}$ to these two new designs.

Formal verification In Chapter 6, I presented Algorithm $\mathcal{P}$ only informally by sketching the type inference algorithm for GADT `case` expressions. Due to the design complexity of the algorithm, I have been unable to come up with a simple description of Algorithm $\mathcal{P}$, and the lack of a simple description in turn prevents proof of type inference soundness and termination. As of now, the only complete description of Algorithm $\mathcal{P}$ is the Haskell implementation. More work is necessary to define an abstract formal description of Algorithm $\mathcal{P}$ and to provide a formal proof of correctness.

Feature enhancements The plain GADT type system, for which I designed Algorithm $\mathcal{P}$, captures the essence of generalized algebraic data types. This type system, however, lacks many features that are supported by modern functional programming language implementations such as the Glasgow Haskell Compiler. Here are a few examples:

- Type annotations,
- Nested patterns in pattern-matching branches,
- Qualified types (*e.g.,* type classes in Haskell),
- Higher-kinded type variables, and
- Rank-N polymorphism.

More work is necessary to extend Algorithm $\mathcal{P}$ to these programming language and type system features. And, tying back to the idea of using type inference as a research vehicle, I am certain that the efforts to extend Algorithm $\mathcal{P}$ will lead to some interesting discoveries about how generalized algebraic data types interact with these other type system features.

REFERENCES

[1] Lennart Augustsson and Kent Petersson. Silly type families. Online at `http://web.cecs.pdx.edu/~sheard/papers/silly.pdf` (accessed June 16, 2010), September 1994.

[2] Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, volume 37(9) of *ACM SIGPLAN Notices*, pages 157–166. ACM, October 2002.

[3] Karen L. Bernstein and Eugene W. Stark. Debugging type errors (full version). Technical report, State University of New York at Stony Brook, Stony Brook, NY, USA, November 1995.

[4] Carlos Camarão and Lucília Figueiredo. ML has principal typings. In *4th Brazilian Symposium on Programming Languages*, Recife, Brazil, May 2000.

[5] James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, pages 90–104. ACM Press, 2002.

[6] James Cheney and Ralf Hinze. First-class phantom types. Technical Report TR2003-1901, Cornell University, July 2003.

[7] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, pages 301–312, New York, NY, USA, September 1998. ACM Press.

[8] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 207–212, New York, NY, USA, January 1982. ACM Press.

[9] Anatoli Degtyarev and Andrei Voronkov. Simultaneous rigid e-unification is undecidable. In *Annual Conference of the European Association for Computer Science Logic, CSL'95*, volume 1092 of *LNCS*, pages 178–190. Springer, 1996.

[10] Peter Forrest. The identity of indiscernibles. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Center for the Study of Language and Information, Stanford University, Summer 2008 edition, 2010. Online at `http://plato.stanford.edu/archives/fall2008/entries/identity-indiscernible/`.

[11] Jean H. Gallier, Paliath Narendran, David Plaisted, and Wayne Snyder. Rigid e-unification: Np-completeness and applications to equational matings. Technical Report MS-CIS-88-14, University of Pennsylvania, March 1988.

[12] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[13] Ralf Hinze. *Frown: An LALR(k) Parser Generator for Haskell*, November 2005. Online at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.135.3574` (accessed June 16, 2010).

[14] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, UK, May 2003.

[15] Simon Peyton Jones, Dimitrios Vytiniotis, and Stephanie Weirich. Simple unification-based type inference for GADTs, technical appendix. Technical Report MS-CIS-05-22, University of Pennsylvania, May 2006.

[16] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(1):1–82, January 2007.

[17] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *ICFP'06: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, pages 50–61, New York, NY, USA, September 2006. ACM Press.

[18] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: type inference for generalized algebraic data types. Technical Report MS-CIS-05-26, University of Pennsylvania, July 2004.

[19] Chuan kai Lin and Tim Sheard. Pointwise generalized algebraic data types. In *TLDI'10: Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 51–62, New York, NY, USA, September 2010. ACM Press.

[20] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. In *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*, pages 468–476, New York, NY, USA, May 1990. ACM Press.

[21] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, April 1993.

[22] Konstantin Laüfer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1411–1430, September 1994.

[23] Chuan-kai Lin. Programming monads operationally with Unimo. In *ICFP'06: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, pages 274–285, New York, NY, USA, September 2006. ACM Press.

[24] Michael Maher. Herbrand constraint abduction. In *LICS'05: Proceedings of the 20th Annual Symposium on Logic in Computer Science*, pages 397–406, Los Alamitos, CA, USA, June 2005. IEEE Computer Society.

[25] Bruce J. McAdam. On the unification of substitutions in type inference. Technical Report ECS-LFCS-98-384, Department of Computer Science, University of Edinburgh, March 1998.

[26] Bruce J. McAdam. On the unification of substitutions in type inference. In *Implementation of Functional Languages: 10th International Workshop, IFL'98*, volume 1595 of *LNCS*, pages 137–152. Springer, 1999.

[27] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[28] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA, Revised edition, May 1997.

[29] Alan Mycroft. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*, volume 167 of *LNCS*, pages 217–228. Springer-Verlag, April 1984.

[30] Henrik Nilsson. Dynamic optimization for functional reactive programming using generalized algebraic data types. In *ICFP'05: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 54–65, New York, NY, USA, September 2005. ACM Press.

[31] G. D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5:153–163, 1970.

[32] François Pottier and Yann Régis-Gianas. Stratified type inference for generalized algebraic data types. In *POPL'06: Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–244, New York, NY, USA, January 2006. ACM Press.

[33] Eric S. Raymond. *The Art of UNIX Programming.* Addison-Wesley Professional Computing Series. Addison-Wesley Professional, First edition, September 2003.

[34] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, March 1953.

[35] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[36] Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. Complete and decidable type inference for GADTs. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 341–352, New York, NY, USA, September 2009. ACM Press.

[37] Tim Sheard. Generic programming in Omega. In Roland Backhouse, Jeremy

Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Datatype-Generic Programming*, volume 4719 of *LNCS*, pages 258–284. Springer, 2006.

[38] Tim Sheard and Nathan Linger. Programming in Omega. In Zoltán Horváth, Rinus Plasmeijer, Anna Soós, and Viktória Zsók, editors, *Central European Functional Programming School*, volume 5161 of *LNCS*, pages 158–227. Springer, 2007.

[39] Tim Sheard and Emir Pasalic. Meta-programming with built-in type equality. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages*, pages 106–124, July 2004.

[40] Vincent Simonet and François Pottier. A constraint-based approach to guarded algebraic data types. *ACM Transactions on Programming Languages and Systems*, 29(1):1–56, January 2007.

[41] Morten Heine Sørensen and Paweł Urzyczyn. *Lectures on the Curry-Howard Isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, September 2006.

[42] Peter J. Stuckey and Martin Sulzmann. Type inference for guarded recursive data types. *The Computing Research Repository (CoRR)*, abs/cs/0507037, July 2005.

[43] Martin Sulzmann, Tom Schrijvers, and Peter J. Stuckey. Type inference for GADTs via Herbrand constraint abduction. Technical Report CW507, Department of Computer Science, K. U. Leuven, Leuven, Belgium, January 2008.

[44] The GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.4*, March 2005.

[45] The Coq proof assistant. Online at http://coq.inria.fr/. Accessed June 16, 2010.

[46] The Darcs project. Online at http://www.darcs.net/. Accessed June 16, 2010.

[47] The Pugs project. Online at http://www.pugscode.org/. Accessed June 16, 2010.

[48] Cristiano Vasconcellos, Lucilia Figueiredo, and Carlos Camarão. Practical type inference for polymorphic recursion: an implementation in Haskell. *Journal of Universal Computer Science*, 9(8):873–890, August 2003.

[49] Brian Weatherson. Intrinsic vs. extrinsic properties. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Center for the Study of Language and Information, Stanford University, Fall 2008 edition, 2008. Online at http://plato.stanford.edu/archives/fall2008/entries/intrinsic-extrinsic/.

[50] Stephanie Weirich. Type-safe cast: Functional Pearl. In *ICFP'00: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, volume 35(9) of *ACM SIGPLAN Notices*, pages 58–67, New York, NY, USA, September 2000. ACM Press.

[51] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge University Press, second edition, January 1927.

[52] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 224–235, New York, NY, USA, January 2003. ACM Press.

Appendix A

CORRECTNESS OF POINTWISE COMPLETION

In this appendix, I prove the correctness of pointwise completion, which I introduced in §5.2 (Figure 5.5, p. 136) as a way to recover pointwise unifiability between scrutinee types and pattern types.

A.1 TERMINATION

In this section, I prove that $\mathit{pwc}(s, t)$ terminates.

Definition 7. Let types s, t be given. I define the function *size*, which takes the explicit error token $\perp$ or a set containing pairs of types as argument, as follows:

$$\mathit{size}(S) = \begin{cases} 0 & \text{if } S = \perp \\ \sum_{(x,y)\in S} \mathit{count}(y) & \text{otherwise} \end{cases}$$

$$\mathit{count}(x) = \begin{cases} 1 & \text{if } x = \alpha \\ 1 + \sum_{x \in \overline{s}} \mathit{count}(x) & \text{if } x = T\ \overline{s} \end{cases}$$

I designed *size* such that $\mathit{size}(s \curlyvee t)$ counts the number of symbols (*i.e.,* type constructors and type variables) in the set $(s \curlyvee t)$ that come from t. ❈

The following lemma establishes an upper bound on $\mathit{size}(s \curlyvee t)$.

Lemma 17. For all types s, t, $\mathit{count}(t) \geq \mathit{size}(s \curlyvee t)$.

Proof. Trivial by structural induction over s. □

Given a substitution σ, the following lemma describes a way to compute $\sigma(s) \curlyvee t$ from $s \curlyvee t$. To simplify the presentation and the usage of the lemma, in this proof I extend the set union operator $\cup$ to propagate the explicit failure token $\bot$. More precisely,

$$S \cup T = \begin{cases} \bot & \text{if } S = \bot \text{ or } T = \bot \\ \text{Set union of } S \text{ and } T & \text{if } S \text{ and } T \text{ are both sets} \end{cases}$$

I also extend the aggregate union operator $\bigcup$ in a similar fashion.

Lemma 18. Let s, t be types such that $s \curlyvee t \neq \bot$. For any substitution σ,

$$\sigma(s) \curlyvee t = \textstyle\bigcup_{(x,y)\in(s \curlyvee t)} \sigma(x) \curlyvee y$$

Proof. Trivial by structural induction over s. □

The following two lemmas state basic properties of summation.

Lemma 19. Let S and T be finite disjoint sets, and f be a function that maps elements in S and T to natural numbers. Then

$$\textstyle\sum_{x\in S} f(x) + \sum_{x\in T} f(x) = \sum_{x\in(S\cup T)} f(x)$$

Proof. Trivial by set induction over S. □

Lemma 20. Let S and T be finite sets, and f be a function that maps elements in S and T to natural numbers. Then

$$\textstyle\sum_{x\in S} f(x) + \sum_{x\in T} f(x) \geq \sum_{x\in(S\cup T)} f(x)$$

Proof. Informally, the relation holds because every element in $S \cup T$ must appear either in S or in T.

Let $J = S \cap T$ and $K = S \setminus J$ (set S minus J), so $S = J \uplus K$. Then

$$\begin{aligned}
\textstyle\sum_{x \in S} f(x) + \sum_{x \in T} f(x) &= \textstyle\sum_{x \in J} f(x) + \sum_{x \in K} f(x) + \sum_{x \in T} f(x) \\
&\geq \textstyle\sum_{x \in K} f(x) + \sum_{x \in T} f(x) \\
&= \textstyle\sum_{x \in (K \cup T)} f(x) \\
&= \textstyle\sum_{x \in (K \cup (J \cup T))} f(x) \\
&= \textstyle\sum_{x \in (S \cup T)} f(x)
\end{aligned}$$

The derivation completes the proof. □

Lemma 21. Let S be a finite set of pairs of types, then

$$\mathit{size}(S) \geq \mathit{size}(\textstyle\bigcup_{(x,y) \in S} x \curlyvee y)$$

Proof. Le me begin by partitioning S into two disjoint subsets J and K.

$$J = \{(x, y) \mid (x, y) \in S, x \curlyvee y \neq \bot\}$$
$$K = \{(x, y) \mid (x, y) \in S, x \curlyvee y = \bot\}$$

The set D contains pointwise counterparts collected from elements of J.

$$D = \textstyle\bigcup_{(x,y) \in J} x \curlyvee y$$

The following derivation shows the desired inequality; comments in parentheses describe the derivation step right above each comment.

$$\mathit{size}(S) = \mathit{size}(J) + \mathit{size}(K)$$

(Equality holds by Definition 7 and Lemma 19 because $S = J \uplus K$.)

$$\geq \mathit{size}(J)$$

(Inequality holds because $\mathit{size}(K) \geq 0$.)

$$= \textstyle\sum_{(x,y)\in J} \mathit{count}(y)$$

$$\geq \textstyle\sum_{(x,y)\in J} \mathit{size}(x \curlyvee y)$$

(Inequality holds by Lemma 17.)

$$= \textstyle\sum_{(x,y)\in J} \sum_{(x',y')\in(x\curlyvee y)} \mathit{count}(y')$$

(Equality holds by Definition 7.)

$$= \textstyle\sum_{(x,y)\in J,\,(x',y')\in(x\curlyvee y)} \mathit{count}(y')$$

$$\geq \textstyle\sum_{(x',y')\in D} \mathit{count}(y')$$

(Inequality holds because for every pair of types $(x', y') \in D$, there exists a pair of types $(x, y) \in J$ such that $(x', y') \in (x \curlyvee y)$.)

$$= \mathit{size}(D)$$

$$= \mathit{size}(\textstyle\bigcup_{(x,y)\in J} x \curlyvee y)$$

$$\geq \mathit{size}(\textstyle\bigcup_{(x,y)\in (J\cup K)} x \curlyvee y)$$

(Inequality holds by definition of K and because $\mathit{size}(\bot) = 0$.)

$$= \mathit{size}(\textstyle\bigcup_{(x,y)\in S} x \curlyvee y)$$

The derivation completes the proof. □

Theorem 22. For all types s, t, $\mathit{pwc}(s, t)$ terminates.

Proof. I conduct the proof by showing that under fixed types s and t, $\mathit{patch}(\sigma)$ terminates for all substitutions σ. To show that $\mathit{patch}(\sigma)$ terminates, I show that there exists a size measure $m(\sigma)$ that decreases in a well-founded domain for every recursive call to *patch*. More specifically, m satisfies the following inequalities, each derived from a recursive call in *patch*:

1. $m(\sigma) > m([T\ \overline{\eta}/\alpha] \circ \sigma)$ where $\sigma(s) \curlyvee t = \{(\alpha, T\ \overline{x})\} \uplus E$, and $\overline{\eta}$ are fresh type variables,

2. $m(\sigma) > m([T\ \overline{\eta}/\alpha] \circ \sigma)$ where $\sigma(s) \curlyvee t = \{(\alpha, T\ \overline{x}), (y, \beta)\} \uplus E$, and $\overline{\eta}$ are fresh type variables, and

3. $m(\sigma) > m(mgu(x \sim y) \circ \sigma)$ where $\sigma(s) \curlyvee t = \{(x, \beta), (y, \beta)\} \uplus E$, and the types x and y are unifiable.

I define $m(\sigma) = \mathit{size}(\sigma(s) \curlyvee t)$. The function m returns natural numbers, which are a well-founded domain with respect to the relation $>$. Now I verify that m satisfies the three aforementioned inequalities.

1. Let me start by defining the set K as follows:

$$K = \{([T\ \overline{\eta}/\alpha](u), w) \mid (u, w) \in E\}$$

For every $(u', w') \in K$, there exists a type u'' such that $(u'', w') \in E$ and $u' = [T\ \overline{\eta}/\alpha](u'')$, so $\mathit{size}(E) \geq \mathit{size}(K)$. Now,

$$\begin{aligned} m(\sigma) &= \mathit{size}(\sigma(s) \curlyvee t) \\ &= \mathit{size}(\{(\alpha, T\ \overline{x})\} \uplus E) \\ &= \mathit{size}(\{(\alpha, T\ \overline{x})\}) + \mathit{size}(E) \end{aligned}$$

(Equality holds by Definition 7 and Lemma 19.)

$$\begin{aligned} &= \mathit{count}(T\ \overline{x}) + \mathit{size}(E) \\ &= 1 + \textstyle\sum_{u \in \overline{x}} \mathit{count}(u) + \mathit{size}(E) \\ &> \textstyle\sum_{u \in \overline{x}} \mathit{count}(u) + \mathit{size}(E) \\ &= \mathit{size}(\{(\eta_1, x_1), \ldots, (\eta_n, x_n)\}) + \mathit{size}(E) \quad \text{where } \{x_1, \ldots, x_n\} = \overline{x} \end{aligned}$$

(Equality holds by expanding summation.)

$$= \mathit{size}(T\ \overline{\eta} \curlyvee T\ \overline{x}) + \mathit{size}(E)$$

$$= \mathit{size}([T\ \overline{\eta}/\alpha](\alpha) \curlyvee T\ \overline{x}) + \mathit{size}(E)$$

$$\geq \mathit{size}([T\ \overline{\eta}/\alpha](\alpha) \curlyvee T\ \overline{x}) + \mathit{size}(K)$$

$$\geq \mathit{size}([T\ \overline{\eta}/\alpha](\alpha) \curlyvee T\ \overline{x}) + \mathit{size}(\textstyle\bigcup_{(x',y')\in K} x' \curlyvee y')$$

(Inequality holds by Lemma 21.)

$$= \mathit{size}([T\ \overline{\eta}/\alpha](\alpha) \curlyvee T\ \overline{x}) +$$
$$\mathit{size}(\textstyle\bigcup_{(x',y')\in\{([T\ \overline{\eta}/\alpha](u),w)\ \mid\ (u,w)\in E\}} x' \curlyvee y')$$

(Equality holds by expanding K using its definition.)

$$= \mathit{size}([T\ \overline{\eta}/\alpha](\alpha) \curlyvee T\ \overline{x}) +$$
$$\mathit{size}(\textstyle\bigcup_{(x',y')\in\{(u,w)\ \mid\ (u,w)\in E\}} [T\ \overline{\eta}/\alpha](x') \curlyvee y')$$

(Equality holds by moving $[T\ \overline{\eta}/\alpha]$ to the summand.)

$$= \mathit{size}([T\ \overline{\eta}/\alpha](\alpha) \curlyvee T\ \overline{x}) + \mathit{size}(\textstyle\bigcup_{(x',y')\in E} [T\ \overline{\eta}/\alpha](x') \curlyvee y')$$

$$\geq \mathit{size}(([T\ \overline{\eta}/\alpha](\alpha) \curlyvee T\ \overline{x}) \cup (\textstyle\bigcup_{(x',y')\in E} [T\ \overline{\eta}/\alpha](x') \curlyvee y'))$$

(Inequality holds by Lemma 20.)

$$= \mathit{size}((\textstyle\bigcup_{(x',y')\in\{(\alpha,T\ \overline{x})\}}[T\ \overline{\eta}/\alpha](x') \curlyvee y') \cup$$
$$(\textstyle\bigcup_{(x',y')\in E} [T\ \overline{\eta}/\alpha](x') \curlyvee y'))$$

(Equality holds because $F(x) = \bigcup_{x'\in\{x\}} F(x')$.)

$$= \mathit{size}(\textstyle\bigcup_{(x',y')\in(\{(\alpha,T\ \overline{x})\}\cup E)} [T\ \overline{\eta}/\alpha](x') \curlyvee y')$$

$$= \mathit{size}(\textstyle\bigcup_{(x',y')\in(\sigma(s)\curlyvee t)} [T\ \overline{\eta}/\alpha](x') \curlyvee y')$$

$$= \mathit{size}(([T\ \overline{\eta}/\alpha] \circ \sigma)(s) \curlyvee t)$$

(Equality holds by Lemma 18.)

$$= m([T\ \overline{\eta}/\alpha] \circ \sigma)$$

2. If I define $E' = \{(y, \beta)\} \uplus E$, then $\sigma(s) \curlyvee t = \{(\alpha, T\ \overline{x})\} \uplus E'$, and the same argument as in the previous point shows that $m(\sigma) > m([T\ \overline{\eta}/\alpha] \circ \sigma)$.

3. Let me start by defining the set K as follows:

$$K = \{(\mathit{mgu}(x \sim y)(u), w) \mid (u, w) \in E\}$$

For every $(u', w') \in K$, there exists a type u'' such that $(u'', w') \in E$ and $u' = \mathit{mgu}(x \sim y)(u'')$, so $\mathit{size}(E) \geq \mathit{size}(K)$. Now,

$$\begin{aligned}
m(\sigma) &= \mathit{size}(\sigma(s) \curlyvee t) \\
&= \mathit{size}(\{(x, \beta), (y, \beta)\} \uplus E) \\
&= \mathit{size}(\{(x, \beta), (y, \beta)\}) + \mathit{size}(E) \\
&= 2 + \mathit{size}(E) \\
&> 1 + \mathit{size}(E) \\
&= \mathit{size}(\{(\mathit{mgu}(x \sim y)(x), \beta)\}) + \mathit{size}(E) \\
&= \mathit{size}(\{(\mathit{mgu}(x \sim y)(x), \beta)\} \cup \{(\mathit{mgu}(x \sim y)(y), \beta)\}) + \mathit{size}(E)
\end{aligned}$$

(Equality holds because $\mathit{mgu}(x \sim y)(x) = \mathit{mgu}(x \sim y)(y)$.)

$$= \mathit{size}(\mathit{mgu}(x \sim y)(x) \curlyvee \beta \cup \mathit{mgu}(x \sim y)(y) \curlyvee \beta) + \mathit{size}(E)$$

(Equality holds because for all types u, $u \curlyvee \beta = \{(u, \beta)\}$.)

$$\begin{aligned}
&= \mathit{size}(\textstyle\bigcup_{(x',y') \in \{(x,\beta),(y,\beta)\}} \mathit{mgu}(x \sim y)(x') \curlyvee y') + \mathit{size}(E) \\
&\geq \mathit{size}(\textstyle\bigcup_{(x',y') \in \{(x,\beta),(y,\beta)\}} \mathit{mgu}(x \sim y)(x') \curlyvee y') + \mathit{size}(K) \\
&\geq \mathit{size}(\textstyle\bigcup_{(x',y') \in \{(x,\beta),(y,\beta)\}} \mathit{mgu}(x \sim y)(x') \curlyvee y') + \\
&\qquad \mathit{size}(\textstyle\bigcup_{(x',y') \in K} x' \curlyvee y')
\end{aligned}$$

(Inequality holds by Lemma 21.)

$$\begin{aligned}&= \mathit{size}(\textstyle\bigcup_{(x',y')\in\{(x,\beta),(y,\beta)\}} \mathit{mgu}(x \sim y)(x') \curlyvee y') + \\ &\qquad \mathit{size}(\textstyle\bigcup_{(x',y')\in\{(\mathit{mgu}(x\sim y)(u),w)\ \mid\ (u,w)\in E\}} x' \curlyvee y')\end{aligned}$$

(Equality holds by expanding K using its definition.)

$$\begin{aligned}&= \mathit{size}(\textstyle\bigcup_{(x',y')\in\{(x,\beta),(y,\beta)\}} \mathit{mgu}(x \sim y)(x') \curlyvee y') + \\ &\qquad \mathit{size}(\textstyle\bigcup_{(x',y')\in\{(u,w)\ \mid\ (u,w)\in E\}} \mathit{mgu}(x \sim y)(x') \curlyvee y')\end{aligned}$$

(Equality holds by moving $\mathit{mgu}(x \sim y)$ to the summand.)

$$\begin{aligned}&= \mathit{size}(\textstyle\bigcup_{(x',y')\in\{(x,\beta),(y,\beta)\}} \mathit{mgu}(x \sim y)(x') \curlyvee y') + \\ &\qquad \mathit{size}(\textstyle\bigcup_{(x',y')\in E} \mathit{mgu}(x \sim y)(x') \curlyvee y') \\ &\geq \mathit{size}((\textstyle\bigcup_{(x',y')\in\{(x,\beta),(y,\beta)\}} \mathit{mgu}(x \sim y)(x') \curlyvee y') \cup \\ &\qquad\qquad (\textstyle\bigcup_{(x',y')\in E} \mathit{mgu}(x \sim y)(x') \curlyvee y'))\end{aligned}$$

(Inequality holds by Lemma 20.)

$$\begin{aligned}&= \mathit{size}(\textstyle\bigcup_{(x',y')\in(\{(x,\beta),(y,\beta)\}\cup E)} \mathit{mgu}(x \sim y)(x') \curlyvee y') \\ &= \mathit{size}(\textstyle\bigcup_{(x',y')\in(\sigma(s)\curlyvee t)} \mathit{mgu}(x \sim y)(x') \curlyvee y') \\ &= \mathit{size}((\mathit{mgu}(x \sim y) \circ \sigma)(s) \curlyvee t)\end{aligned}$$

(Equality holds by Lemma 18.)

$$= m(\mathit{mgu}(x \sim y) \circ \sigma)$$

The size measure $m(\sigma)$ satisfies all the conditions, so $\mathit{patch}(\sigma)$ terminates, and $\mathit{pwc}(s,t)$ terminates. □

A.2 SOUNDNESS

Lemma 23. If $\mathit{patch}(\sigma) = \sigma'$ and $\sigma' \neq \bot$, then there exists a substitution θ such that $\sigma' = \theta \circ \sigma$.

Proof. Trivial by induction over the recursive calls of *patch*. □

Lemma 24. Let θ and σ be idempotent substitutions. The substitution $(\theta \circ \sigma)$ is idempotent if $\mathrm{dom}(\sigma) \mathrel{\#} \mathit{tyvar}(\mathrm{rng}(\theta))$.

Proof. It is trivial to show that the following relations hold.

$$\mathrm{dom}(\theta \circ \sigma) \subseteq \mathrm{dom}(\theta) \cup \mathrm{dom}(\sigma)$$
$$\mathit{tyvar}(\mathrm{rng}(\theta \circ \sigma)) = \mathit{tyvar}(\mathrm{rng}(\theta)) \cup (\mathit{tyvar}(\mathrm{rng}(\sigma)) \setminus \mathrm{dom}(\theta))$$

The binary operator $\setminus$ subtracts the second set from the first. To show that $(\theta \circ \sigma)$ is idempotent, I show $\mathrm{dom}(\theta \circ \sigma) \mathrel{\#} \mathit{tyvar}(\mathrm{rng}(\theta \circ \sigma))$ as follows:

1. $\mathrm{dom}(\theta) \mathrel{\#} \mathit{tyvar}(\mathrm{rng}(\theta))$ because θ is idempotent,

2. $\mathrm{dom}(\sigma) \mathrel{\#} (\mathit{tyvar}(\mathrm{rng}(\sigma)) \setminus \mathrm{dom}(\theta))$ because σ is idempotent,

3. $\mathrm{dom}(\theta) \mathrel{\#} (\mathit{tyvar}(\mathrm{rng}(\sigma)) \setminus \mathrm{dom}(\theta))$ because of the set subtraction, and

4. $\mathrm{dom}(\sigma) \mathrel{\#} \mathit{tyvar}(\mathrm{rng}(\theta))$ is given.

Since $\mathrm{dom}(\theta \circ \sigma) \mathrel{\#} \mathit{tyvar}(\mathrm{rng}(\theta \circ \sigma))$, $(\theta \circ \sigma)$ is idempotent. □

Lemma 25. If $\mathit{tyvar}(s) \mathrel{\#} \mathit{tyvar}(t)$, then $\mathit{pwc}(s, t)$ maintains the invariant that, for every call to $\mathit{patch}(\sigma)$, σ is an idempotent substitution, $\mathit{tyvar}(\sigma(s)) \mathrel{\#} \mathit{tyvar}(t)$, and $\sigma(t) = t$.

Proof. I conduct the proof by structural induction over the call graph of $\mathit{pwc}(s, t)$, which, except for the initial call to $\mathit{pwc}(s, t)$, consists entirely of calls to *patch*. This lemma is equivalent to the following three properties, which are derived from the calls to *patch*:

1. The identity substitution id is idempotent, $\mathrm{id}(t) = t$, and $\mathit{tyvar}(\mathrm{id}(s)) \# \mathit{tyvar}(t)$.

2. $[T\ \overline{\eta}/\alpha] \circ \sigma$ is idempotent, $([T\ \overline{\eta}/\alpha] \circ \sigma)(t) = t$, and $\mathit{tyvar}(([T\ \overline{\eta}/\alpha] \circ \sigma)(s)) \# \mathit{tyvar}(t)$, where σ is idempotent, $\sigma(t) = t$, and $\mathit{tyvar}(\sigma(s)) \# \mathit{tyvar}(t)$.

3. $\mathit{mgu}(x \sim y) \circ \sigma$ is idempotent, $(\mathit{mgu}(x \sim y) \circ \sigma)(t) = t$, and $\mathit{tyvar}((\mathit{mgu}(x \sim y) \circ \sigma)(s)) \# \mathit{tyvar}(t)$, where $\{(x, \beta), (y, \beta)\} \in \sigma(s) \curlyvee t$, σ is idempotent, $\sigma(t) = t$, and $\mathit{tyvar}(\sigma(s)) \# \mathit{tyvar}(t)$.

I shall prove each property in turn.

1. Trivial.

2. Since $\overline{\eta}$ are fresh and $[T\ \overline{\eta}/\alpha]$ is idempotent, by Lemma 24 I know that $([T\ \overline{\eta}/\alpha] \circ \sigma)$ is idempotent.

   Since $\alpha \notin \mathit{tyvar}(t)$ and $\sigma(t) = t$, I know $([T\ \overline{\eta}/\alpha] \circ \sigma)(t) = t$. Since $\mathit{tyvar}(([T\ \overline{\eta}/\alpha] \circ \sigma)(s)) \subseteq \mathit{tyvar}(\sigma(s), T\ \overline{\eta})$, $\overline{\eta}$ are fresh, and $\mathit{tyvar}(\sigma(s)) \# \mathit{tyvar}(t)$, I know $\mathit{tyvar}(([T\ \overline{\eta}/\alpha] \circ \sigma)(s)) \# \mathit{tyvar}(t)$.

3. The substitution $\mathit{mgu}(x \sim y)$ is idempotent, and $\mathit{tyvar}(\mathrm{rng}(\mathit{mgu}(x \sim y))) \subseteq \mathit{tyvar}(x, y) \subseteq \mathit{tyvar}(\sigma(s))$. Since σ is idempotent, $\mathit{tyvar}(\sigma(s)) \# \mathrm{dom}(\sigma)$, so $\mathit{tyvar}(\mathrm{rng}(\mathit{mgu}(x \sim y))) \# \mathrm{dom}(\sigma)$. By Lemma 24 I know $\mathit{mgu}(x \sim y) \circ \sigma$ is idempotent.

   Since $\mathrm{dom}(\mathit{mgu}(x \sim y)) \subseteq \mathit{tyvar}(x, y) \subseteq \mathit{tyvar}(\sigma(s))$, I know $(\mathit{mgu}(x \sim y) \circ \sigma)(t) = t$. Since $\mathit{tyvar}(\mathrm{rng}(\mathit{mgu}(x \sim y))) \subseteq \mathit{tyvar}(\sigma(s))$,

$\mathit{tyvar}((\mathit{mgu}(x \sim y) \circ \sigma)(s)) \subseteq \mathit{tyvar}(\sigma(s))$. Since $\mathit{tyvar}(\sigma(s)) \mathrel{\#} \mathit{tyvar}(t)$, $\mathit{tyvar}((\mathit{mgu}(x \sim y) \circ \sigma)(s)) \mathrel{\#} \mathit{tyvar}(t)$.

The analysis completes the proof. □

Lemma 26. Let types s, t and substitution σ be given. If $\mathit{patch}(\sigma) = \sigma$, then $\mathit{patch}(\sigma)$ must take the fourth pattern-matching branch.

Proof. I will conduct this proof by contradiction.
Assume to the contrary that $\mathit{patch}(\sigma)$ takes one of the first three branches. Let us look at each branch in turn.

1. In the first branch, $\sigma(s) \curlyvee t = \{(\alpha, T\ \overline{x})\} \uplus E$, and $\mathit{patch}(\sigma) = \mathit{patch}([T\ \overline{\eta}/\alpha] \circ \sigma)$. If $\mathit{patch}([T\ \overline{\eta}/\alpha] \circ \sigma) = \bot$, then $\mathit{patch}(\sigma) = \bot \neq \sigma$, contradicting my assumption.

   If $\mathit{patch}([T\ \overline{\eta}/\alpha] \circ \sigma) = \sigma' \neq \bot$, by Lemma 23, there exists a substitution θ such that $\sigma' = \theta \circ [T\ \overline{\eta}/\alpha] \circ \sigma$. By Lemma 25 I know that σ is idempotent, so $(\alpha, T\ \overline{x}) \in \sigma(s) \curlyvee t$ implies $\alpha \notin \mathrm{dom}(\sigma)$. However, $\alpha \in \mathrm{dom}(\sigma')$, so $\mathit{patch}(\sigma) = \sigma' \neq \sigma$, contradicting my assumption.

   Therefore $\mathit{patch}(\sigma)$ does not take the first branch.

2. In the second branch, if I define $E' = \{(y, \beta)\} \uplus E$, then $\sigma(s) \curlyvee t = \{(\alpha, T\ \overline{x})\} \uplus E'$, and $\mathit{patch}(\sigma) = \mathit{patch}([T\ \overline{\eta}/\alpha] \circ \sigma)$.

   Applying the same arguments as in the first branch shows that $\mathit{patch}(\sigma)$ does not take the second branch.

3. In the third branch, $\sigma(s) \curlyvee t = \{(x, \beta), (y, \beta)\} \uplus E$ and $x \neq y$. If $\mathit{mgu}(x \sim y) = \bot$, then $\mathit{patch}(\sigma) = \bot \neq \sigma$, contradicting my assumption. If $\mathit{mgu}(x \sim y) \neq \bot$, then $\mathit{patch}(\sigma) = \mathit{patch}(\mathit{mgu}(x \sim y) \circ \sigma)$, and there are two possible cases:

(a) $\mathit{patch}(\mathit{mgu}(x \sim y) \circ \sigma) = \bot$, then $\mathit{patch}(\sigma) = \bot \neq \sigma$, contradicting my assumption.

(b) $\mathit{patch}(\mathit{mgu}(x \sim y) \circ \sigma) = \sigma' \neq \bot$, then there exists a substitution θ such that $\sigma' = \theta \circ \mathit{mgu}(x \sim y) \circ \sigma$. Since $\sigma(x) \neq \sigma(y)$ but $\sigma'(x) = \sigma'(y)$, $\mathit{patch}(\sigma) = \sigma' \neq \sigma$, contradicting my assumption.

Therefore $\mathit{patch}(\sigma)$ does not take the third branch.

The analysis proves that $\mathit{patch}(\sigma)$ must take the fourth branch. □

Theorem 27. If $\mathit{tyvar}(s) \# \mathit{tyvar}(t)$, $\mathit{pwc}(s, t) = \sigma$, and $\sigma \neq \bot$, then $\mathit{tyvar}(\sigma(s)) \# \mathit{tyvar}(t)$, and $\sigma(s)$ and t are pointwise unifiable.

Proof. It is easy to show $\mathit{tyvar}(\sigma(s)) \# \mathit{tyvar}(t)$ by Lemma 25, so I focus on proving that $\sigma(s)$ and t are pointwise unifiable. I will conduct this proof by contradiction.

Assume to the contrary that $\sigma(s)$ and t are *not* pointwise unifiable. Since the pointwise unification algorithm is sound, pointwise unification of $\sigma(s)$ and t must fail. There are four ways pointwise unification can fail, and I show that, in each case, pointwise unification failure leads to contradiction. The four possible pointwise unification failures are:

1. Pointwise unification fails if $\sigma(s) \curlyvee t = \bot$,
2. Pointwise unification fails if $\sigma(s) \curlyvee t = \{(\alpha, \beta)\} \uplus E$, where $\alpha \neq \beta$ and $\{\alpha, \beta\} \subseteq \mathit{tyvar}(E)$,
3. Pointwise unification fails if $\sigma(s) \curlyvee t = \{(T\,\overline{x}, \alpha)\} \uplus E$, where $\alpha \in \mathit{tyvar}(E) \cup \mathit{tyvar}(T\,\overline{x})$, and
4. Pointwise unification fails if $\sigma(s) \curlyvee t = \{(\alpha, T\,\overline{x})\} \uplus E$, where $\alpha \in \mathit{tyvar}(E) \cup \mathit{tyvar}(T\,\overline{x})$.

Since $\mathit{pwc}(s,t) = \sigma$, the evaluation of $\mathit{pwc}(s,t)$ must end in a call to $\mathit{patch}(\sigma)$, which returns its argument σ unchanged. Let us look at how each pointwise unification failure affects the evaluation of $\mathit{patch}(\sigma)$:

1. Let $\sigma(s) \curlyvee t = \bot$, then $\mathit{patch}(\sigma) = \bot$, which contradicts with my assumption $\mathit{patch}(\sigma) = \sigma \neq \bot$.

2. Let $\sigma(s) \curlyvee t = \{(\alpha,\beta)\} \uplus E$ such that $\alpha \neq \beta$ and $\{\alpha,\beta\} \subseteq \mathit{tyvar}(E)$. By Lemma 25, I can further refine this case into three sub-cases:

    (a) $\sigma(s) \curlyvee t = \{(\alpha,\beta),(\alpha, T\,\overline{x})\} \uplus E'$, where $\beta \in \mathit{tyvar}(T\,\overline{x})$. Then, $\mathit{patch}(\sigma)$ takes the first branch, and, by Lemma 26, $\mathit{patch}(\sigma) \neq \sigma$, which contradicts with my assumption $\mathit{patch}(\sigma) = \sigma$.

    (b) $\sigma(s) \curlyvee t = \{(\alpha,\beta),(S\,\overline{y}, \beta)\} \uplus E'$, where $\alpha \in \mathit{tyvar}(S\,\overline{y})$. Then, $\mathit{patch}(\sigma)$ takes the third branch, and, by Lemma 26, $\mathit{patch}(\sigma) \neq \sigma$, which contradicts with my assumption $\mathit{patch}(\sigma) = \sigma$.

    (c) $\sigma(s) \curlyvee t = \{(\alpha,\beta),(\delta, T\,\overline{x}),(S\,\overline{y},\gamma)\} \uplus E'$, where $\alpha \in \mathit{tyvar}(S\,\overline{y})$, $\beta \in \mathit{tyvar}(T\,\overline{x})$, $\alpha \neq \delta$, and $\beta \neq \gamma$. Then, $\mathit{patch}(\sigma)$ takes the second branch, and, by Lemma 26, $\mathit{patch}(\sigma) \neq \sigma$, which contradicts with my assumption $\mathit{patch}(\sigma) = \sigma$.

    Since all three sub-cases contradict my assumption that $\mathit{patch}(\sigma) = \sigma$, this case contradicts my assumption that $\mathit{patch}(\sigma) = \sigma$.

3. Let $\sigma(x) \curlyvee t = \{(T\,\overline{x}, \alpha)\} \uplus E$ such that $\alpha \in \mathit{tyvar}(E) \cup \mathit{tyvar}(T\,\overline{x})$. By Lemma 25, I know $\alpha \notin T\,\overline{x}$, so $\alpha \in \mathit{tyvar}(E)$, and I can further refine this case into two sub-cases:

    (a) $\sigma(x) \curlyvee t = \{(T\,\overline{x}, \alpha),(\beta, S\,\overline{y})\} \uplus E'$, where $\alpha \in \mathit{tyvar}(S\,\overline{y})$. Then, $\mathit{patch}(\sigma)$ takes the second branch, and, by Lemma 26, $\mathit{patch}(\sigma) \neq \sigma$, which contradicts with my assumption $\mathit{patch}(\sigma) = \sigma$.

   (b) $\sigma(x) \curlyvee t = \{(T\ \overline{x}, \alpha), (S\ \overline{y}, \alpha)\} \uplus E'$. Then, $\mathit{patch}(\sigma)$ takes the third branch, and, by Lemma 26, $\mathit{patch}(\sigma) \neq \sigma$, which contradicts with my assumption $\mathit{patch}(\sigma) = \sigma$.

   Since both sub-cases contradict my assumption that $\mathit{patch}(\sigma) = \sigma$, this case contradicts my assumption that $\mathit{patch}(\sigma) = \sigma$.

4. Let $\sigma(s) \curlyvee t = \{(\alpha, T\ \overline{x})\} \uplus E$ such that $\alpha \in \mathit{tyvar}(E) \cup \mathit{tyvar}(T\ \overline{x})$. By Lemma 25, I know $\alpha \notin T\ \overline{x}$, so $\alpha \in \mathit{tyvar}(E)$, and I can further refine this case into two sub-cases:

   (a) $\sigma(x) \curlyvee t = \{(\alpha, T\ \overline{x}), (S\ \overline{y}, \beta)\} \uplus E'$, where $\alpha \in \mathit{tyvar}(S\ \overline{y})$. Then, $\mathit{patch}(\sigma)$ takes the first branch, and, by Lemma 26, $\mathit{patch}(\sigma) \neq \sigma$, which contradicts with my assumption $\mathit{patch}(\sigma) = \sigma$.

   (b) $\sigma(x) \curlyvee t = \{(\alpha, T\ \overline{x}), (\alpha, S\ \overline{y})\} \uplus E'$. Then, $\mathit{patch}(\sigma)$ takes the third branch, and, by Lemma 26, $\mathit{patch}(\sigma) \neq \sigma$, which contradicts with my assumption $\mathit{patch}(\sigma) = \sigma$.

   Since both sub-cases contradict my assumption that $\mathit{patch}(\sigma) = \sigma$, this case contradicts my assumption that $\mathit{patch}(\sigma) = \sigma$.

The analysis shows that pointwise unification for $\sigma(s) \sim t$ must succeed, because any failure contradicts with my assumptions. Since pointwise unification is sound, $\sigma(s)$ and t must be pointwise unifiable. □

A.3 COMPLETENESS

Lemma 28. Let types s, t and substitutions μ, θ be given such that $\mathit{tyvar}(s) \# \mathit{tyvar}(t)$, $\mathit{tyvar}(\theta(s)) \# \mathit{tyvar}(t)$, and μ is a pointwise unifier of $\theta(s) \sim t$. If $s \curlyvee t = \{(y, \beta)\} \uplus E$ and $\mathit{ndom}(y, E)$ is true, then $\beta \in \mathrm{dom}(\mu)$.

Proof. Consider $\theta(y)$.
If $\theta(y) = T\ \overline{x}$, then $\mu(\theta(y)) = \mu(\beta)$ implies that $\beta \in \mathrm{dom}(\mu)$.
If $\theta(y) = \gamma$, then y must be a type variable α, and $\alpha \neq \beta$. I show that $\alpha \notin \mathrm{dom}(\mu)$, therefore $\beta \in \mathrm{dom}(\mu)$. From $\mathit{ndom}(\alpha, E)$ I know that one of the following conditions must hold:

1. $(\alpha, r) \in E$ where $s \neq \beta$. Since β in t and r in t are both pointwise counterparts of γ in $\theta(s)$, $\gamma \notin \mathrm{dom}(\mu)$, and therefore $\beta \in \mathrm{dom}(\mu)$.

2. $(T\ \overline{x}, \eta) \in E$ such that $\alpha \in \mathit{tyvar}(T\ \overline{x})$. In this case, $\eta \in \mathrm{dom}(\mu)$, $\mu(\eta) = T\ \overline{x} = \mu(T\ \overline{x})$, therefore $\gamma \notin \mathrm{dom}(\mu)$, and $\beta \in \mathrm{dom}(\mu)$.

The analysis completes the proof. □

Lemma 29. Let types s, t and substitution θ be given so that $\mathit{tyvar}(s) \# \mathit{tyvar}(t)$, $\mathit{tyvar}(\theta(s)) \# \mathit{tyvar}(t)$, and that $\theta(s)$ is pointwise unifiable with t. Then, $\mathit{pwc}(s, t)$ maintains the invariant that, for every call to $\mathit{patch}(\sigma)$, there exists a substitution ρ such that $\theta(s) = (\rho \circ \sigma)(s)$.

Proof. I conduct the proof by structural induction over the call graph of $\mathit{pwc}(s, t)$, which, except for the initial call to $\mathit{pwc}(s, t)$, consists entirely of calls to *patch*. This lemma is equivalent to the following four properties, each derived from a call to *patch*:

1. There exists a substitution ρ such that $\theta(s) = (\rho \circ \mathrm{id})(s)$,

2. There exists a substitution ρ such that $\theta(s) = (\rho \circ [T\ \overline{\eta}/\alpha] \circ \sigma)(s)$, where $\sigma(s) \curlyvee t = \{(\alpha, T\ \overline{x})\} \uplus E$, $\alpha \in \mathit{tyvar}(E)$, $\overline{\eta}$ are fresh type variables, and there is a substitution ρ' such that $\theta(s) = (\rho' \circ \sigma)(s)$,

3. There exists a substitution ρ such that $\theta(s) = (\rho \circ [T\ \overline{\eta}/\alpha] \circ \sigma)(s)$, where $\sigma(s) \curlyvee t = \{(\alpha, T\ \overline{x}), (y, \beta)\} \uplus E$, $\mathit{ndom}(y, E)$ is true, $\beta \in \mathit{tyvar}(T\ \overline{x})$, $\overline{\eta}$ are

fresh type variables, and there is a substitution ρ' such that $\theta(s) = (\rho' \circ \sigma)(s)$, and

4. There exists a substitution ρ such that $\theta(s) = (\rho \circ mgu(x \sim y) \circ \sigma)(s)$, where $\sigma(s) \curlyvee t = \{(x, \beta), (y, \beta)\} \uplus E$, $ndom(y, E)$ is true, $x \neq y$, the types x and y are unifiable, and there is a substitution ρ' such that $\theta(s) = (\rho' \circ \sigma)(s)$.

Let μ be a pointwise unifier of $\theta(s) \sim t$. I shall prove each property in turn:

1. Defining $\rho = \theta$ satisfies the property.

2. To prove ρ exists, I need to show that $\rho'(\alpha) = T\,\overline{u}$ for some types $\overline{u}$. I prove this property by contradiction.

   Assume to the contrary that $\rho'(\alpha) = \beta$. Since $T\,\overline{x}$ in t is a pointwise counterpart of β in $\theta(s)$, $\mu(\beta) = T\,\overline{x}$. From $\alpha \in \mathit{tyvar}(E)$, I know that either $(\alpha, w) \in E$ where $w \neq T\,\overline{x}$, or $(S\,\overline{y}, \gamma) \in E$ where $\alpha \in \mathit{tyvar}(S\,\overline{y})$. Let us look at the two cases:

   - If $(\alpha, w) \in E$, then w in t is also a pointwise counterpart of β in $\theta(s)$. However, $\mu(\beta) = T\,\overline{x} \neq w$, so μ is not a pointwise unifier of $\theta(s) \sim t$, contradicting my assumption.
   - If $(S\,\overline{y}, \gamma) \in E$, then there is an occurrence of β in $\theta(s)$ that does not have a pointwise counterpart in t. Since $\beta \in \mathrm{dom}(\mu)$, μ is not a pointwise unifier of $\theta(s) \sim t$, contradicting my assumption.

   Since both cases lead to contradiction, $\rho'(\alpha)$ cannot be a type variable. Since $T\,\overline{x}$ in t is a pointwise counterpart of $\rho'(\alpha)$ in $\theta(s)$, $\rho'(\alpha) = T\,\overline{u}$ for some types $\overline{u}$. Then, defining $\rho = \rho' \circ \overline{[u/\eta]}$ satisfies the property because

   $$(\rho' \circ (\overline{[u/\eta]} \circ [T\,\overline{\eta}/\alpha]) \circ \sigma)(s) = (\rho' \circ [T\,\overline{u}/\alpha] \circ \sigma)(s) = (\rho' \circ \sigma)(s) = \theta(s)$$

3. To prove ρ exists, I need to show that $\rho'(\alpha) = T\,\overline{u}$ for some types $\overline{u}$. I prove this property by contradiction.

   Assume to the contrary that $\rho'(\alpha) = \gamma$. Since $T\,\overline{x}$ in t is a pointwise counterpart of γ in $\theta(s)$, $\mu(\gamma) = T\,\overline{x}$. By Lemma 28, $\beta \in \mathrm{dom}(\mu)$, and from $\beta \in \mathit{tyvar}(T\,\overline{x})$, I have $\mu(\gamma) = T\,\overline{x} \neq \mu(T\,\overline{x})$, contradicting my assumption that μ is a unifier of $\theta(s) \sim t$.

   Since $\rho'(\alpha)$ is not a type variable, and $T\,\overline{x}$ in t is a pointwise counterpart of $\rho'(\alpha)$ in $\theta(s)$, $\rho'(\alpha) = T\,\overline{u}$ for some types $\overline{u}$. Then, defining $\rho = \rho' \circ \overline{[u/\eta]}$ satisfies the property because

$$(\rho' \circ (\overline{[u/\eta]} \circ [T\,\overline{\eta}/\alpha]) \circ \sigma)(s) = (\rho' \circ [T\,\overline{u}/\alpha] \circ \sigma)(s) = (\rho' \circ \sigma)(s) = \theta(s)$$

4. To prove ρ exists, I need to show that $\rho'(x) = \rho'(y)$. By Lemma 28, $\beta \in \mathrm{dom}(\mu)$. Since $\rho'(x)$ in $\theta(s)$ and $\rho'(y)$ in $\theta(s)$ are both pointwise counterparts of β in t, I know $\rho'(x) = \mu(\beta) = \rho'(y)$.

   Since $\mathit{mgu}(x \sim y)$ computes the most general unifier of $x \sim y$, there exists a substitution ρ such that $\rho' = \rho \circ \mathit{mgu}(x \sim y)$, and the substitution ρ thus defined satisfies $\theta(s) = (\rho \circ \mathit{mgu}(x \sim y) \circ \sigma)(s)$.

The analysis shows that $\mathit{pwc}(s,t)$ maintains the desired invariant and completes the proof. □

Theorem 30. Let types s, t be given such that $\mathit{tyvar}(s) \,\#\, \mathit{tyvar}(t)$. Suppose there exists a substitution θ such that $\mathit{tyvar}(\theta(s)) \,\#\, \mathit{tyvar}(t)$ and $\theta(s)$ is pointwise unifiable with t. Then, $\mathit{pwc}(s,t) = \sigma' \neq \perp$ and there is a substitution ρ such that $\theta(s) = (\rho \circ \sigma')(s)$.

Proof. Let μ be a pointwise unifier of $\theta(s) \sim t$, and let $\mathrm{dom}(\theta) \subseteq \mathit{tyvar}(s)$ without loss of generality. Since $\mathit{pwc}(s,t)$ terminates, there exists a substitution σ

such that $\mathit{pwc}(s, t)$ evaluates to $\mathit{patch}(\sigma)$, and $\mathit{patch}(\sigma)$ evaluates to a result σ' without further recursive calls.

By Lemma 29, there exists a substitution ρ such that $\theta(s) = (\rho \circ \sigma)(s)$. Let ρ' be ρ with domain restricted to $\mathit{tyvar}(\sigma(s))$, so $\theta(s) = (\rho' \circ \sigma)(s)$. By Lemma 25, $\mathit{tyvar}(\sigma(s)) \mathbin{\#} \mathit{tyvar}(t)$, so $\rho'(t) = t$. The following derivation shows that $\mu \circ \rho'$ is a unifier of $\sigma(s) \sim t$.

$$(\mu \circ \rho')(\sigma(s)) = \mu((\rho' \circ \sigma)(s)) = \mu(\theta(s)) = \mu(t) = (\mu \circ \rho')(t)$$

Since $\sigma(s)$ is unifiable with t, $\sigma(s) \curlyveedownarrow t \neq \bot$. If $\sigma(s) \curlyveedownarrow t = \{(x, \beta), (y, \beta)\} \uplus E$, then x and y must be unifiable (because every unifier of $\sigma(s) \sim t$ is a unifier of $x \sim y$), so $\mathit{patch}(\sigma) \neq \bot$.

Since $\mathit{patch}(\sigma)$ evaluates to σ' without further recursive calls, $\sigma' = \sigma$, and $(\rho \circ \sigma')(s) = (\rho \circ \sigma)(s) = \theta(s)$. □

Appendix B

HASKELL IMPLEMENTATION OF ALGORITHM $\mathcal{P}$

This appendix contains the complete source code of my Algorithm $\mathcal{P}$ implementation in Haskell. Algorithm $\mathcal{P}$ is a type inference algorithm I developed for the plain GADT type system; please consult Chapter 6 for an informal description of Algorithm $\mathcal{P}$ and Chapter 7 for an evaluation of the algorithm.

B.1 MAIN MODULE

The `Run.hs` source file contains the `main` procedure and other glue code that connect the source language parser to the Algorithm $\mathcal{P}$ implementation.

```haskell
-- Time-stamp: <2010-04-01 14:27:11 cklin>

module Main where

import Control.Monad (unless)
import Data.List (partition)
import Data.Maybe (isJust)
import qualified Data.Map as Map

import Common
import Types
import Front
import Monad
import Inference

---------

type IdentType = (Ident, [String], Maybe Type)

makeConsE :: Program -> ConsE
makeConsE (Decl _ cx) = toMap cons where
    tuple = (",", ConsTy tupleType)
    unit  = ("()", ConsTy (TyCon "()" []))
    cons  = unit : tuple : map make cx
    make (Cons dc t) = (dc, ConsTy t)
```

```
inferPrograms :: [Program] -> [IdentType]
inferPrograms px =
    let (dx, vx) = partition declP px
        consE = Map.unions (map makeConsE dx)
    in runTi (mapM (inferProgram consE) vx) 1

inferProgram :: ConsE -> Program -> Ti IdentType
inferProgram _ (Info doc) =
    return ("Doc", doc, Nothing)
inferProgram consE (Value x e) =
    do let let_e = Let [(x, e)] (Var x)
       (w, mt) <- arrestTi (inferTop consE let_e)
       return (x, w, mt)

declP :: Program -> Bool
declP (Decl _ _) = True
declP _ = False

---------

printIT :: IdentType -> IO ()
printIT (x, w, Nothing) =
    do let oops = "Type inference failed for " ++ x
       unless (x == "Doc") (putStrLn oops)
       putStrLn (unlines w)
printIT (x, w, Just _) =
    do putStrLn (last w)
       putStrLn (unlines $ init w)

main :: IO ()
main = do p <- parseFile "Examples.lhs"
          mapM_ printIT (inferPrograms p)
```

B.2 GENERAL UTILITY FUNCTIONS

The Common.hs source file contains general-purpose utility functions that I use in the type inference algorithm.

```
-- Time-stamp: <2010-03-10 17:33:14 cklin>

module Common where

import Control.Monad (liftM)
import Data.List ((\\), nub, nubBy)
import Data.Map (Map, findWithDefault, fromList, toList)

type Endo a = a -> a

bug :: String -> a
bug msg = error ("BUG: " ++ msg)
```

```
same :: Eq a => [a] -> Bool
same [] = True
same xs = and (zipWith (==) (tail xs) xs)

unique :: [a] -> Maybe a
unique [x] = Just x
unique _ = Nothing

unions :: Eq a => [[a]] -> [a]
unions = nub . concat

unionMap :: Eq b => (a -> [b]) -> [a] -> [b]
unionMap f = unions . map f

map1st :: (a -> b) -> [(a, c)] -> [(b, c)]
map1st f = map (\(u, v) -> (f u, v))

nub1st :: Eq a => Endo [(a, b)]
nub1st = nubBy (\(a, _) (c, _) -> a == c)

overlaps :: Ord a => [a] -> [a] -> Bool
overlaps = any . flip elem

subset :: Eq a => [a] -> [a] -> Bool
subset ux wx = null (ux \\ wx)

lookupX :: Ord k => k -> Map k a -> a
lookupX = findWithDefault (bug "Missing key in lookupX")

lookupZ :: Ord k => k -> Map k k -> k
lookupZ k = findWithDefault k k

-- This is a version of Map.fromList that checks that there are no
-- duplicate keys in the input association list.

toMap :: (Ord k, Show k) => [(k, a)] -> Map k a
toMap assoc =
    let keys = map fst assoc
        dups = nub (keys \\ nub keys)
    in if null dups then fromList assoc
       else bug ("Duplicate keys " ++ show dups)

```

B.3 DATA TYPE DEFINITIONS

The Types.hs source file defines data types for terms, types, and lexical tokens. It also provides some utility functions for the Type data type.

```
-- Time-stamp: <2010-05-13 23:35:27 cklin>

module Types where
```

```
import Control.Monad (guard)
import Data.List ((\\), intercalate, nub)
import qualified Data.Map as Map

import Common

--------- Program abstract syntax tree types

type Ident = String
type Branch = (Pat, Term)

data Term
    = Var Ident
    | Con String
    | Int Integer
    | App Term Term
    | Lam Ident Term
    | Let [(Ident, Term)] Term
    | Case Term [Branch]
      deriving (Eq, Show)

data Pat
    = PatCon String [Ident]
    | PatInt Integer
      deriving Eq

data Type
    = TyVar Ident
    | TyCon String [Type]
    | TyMeta Int
    | TySkol Int
      deriving Eq

data Data
    = Data String [Ident]
      deriving (Eq, Show)

data Cons
    = Cons String Type
      deriving (Eq, Show)

data Program
    = Value Ident Term
    | Decl Data [Cons]
    | Info [String]
      deriving (Eq, Show)

--------- Lexical analyzer token types

data Terminal
    = LexOp             -- +
    | LexDef            -- =
    | LexArr            -- ->
    | LexLam            -- \
    | LexSemi           -- ;
    | LexComa           -- ,
    | LexType           -- ::
```

```
    | LexParL           -- (
    | LexParR           -- )
    | LexBraL           -- {
    | LexBraR           -- }
    | LexVar Ident      -- identifier
    | LexCon String     -- Constructor
    | LexInt Integer    -- 42
    | LexData           -- data  (keyword)
    | LexWhere          -- where (keyword)
    | LexCase           -- case  (keyword)
    | LexOf             -- of    (keyword)
    | LexLet            -- let   (keyword)
    | LexIn             -- in    (keyword)
    | LexNext
    | LexDoc String
    | LexError
      deriving (Eq, Show)

--------- Type inference engine data structures

type Subst  = Map.Map Int Type
type Rename = Map.Map Int Int
type Type2  = (Type, Type)

-- The (polymorphic) variable types and the (polymorphic) data
-- constructor types no longer contain a list of their free type
-- variables because the list is easily reconstructed with freeType.

data VarTy  = VarTy Type
data ConsTy = ConsTy Type

type VarE   = Map.Map Ident VarTy
type ConsE  = Map.Map String ConsTy

mapVarE :: Endo Type -> Endo VarE
mapVarE = Map.map . mapVarTy

mapVarTy :: Endo Type -> Endo VarTy
mapVarTy f (VarTy t) = VarTy (f t)

--------- Frequently used types

botType       = TyVar "a"
intType       = TyCon "Int" []
arrType t1 t2 = TyCon "->"  [t1, t2]
plusType      = arrType intType (arrType intType intType)
tupleType     = arrType varx (arrType vary tuple)
    where tuple = TyCon "," [varx, vary]
          varx  = TyVar "x"
          vary  = TyVar "y"

--------- General type utility functions

metaP :: Type -> Bool
metaP (TyMeta _) = True
metaP _ = False
```

```
consP :: Type -> Bool
consP (TyCon _ _) = True
consP _ = False

deCons :: [Type] -> Maybe (String, [[Type]])
deCons tx =
    let consTc (TyCon tc _) = tc
        consAx (TyCon _ ax) = ax
    in do guard (tx /= [])
          guard (all consP tx)
          guard (same (map consTc tx))
          return (consTc (head tx), map consAx tx)

-- Collect free or meta type variables in either a type or a type
-- environment.

collectType :: Eq a => Endo (Type -> [a])
collectType f = walk where
    walk (TyCon _ tas) = unions (map walk tas)
    walk t             = f t

freeType :: Type -> [Ident]
freeType = collectType free where
    free (TyVar tv) = [tv]
    free _          = []

skolType :: Type -> [Int]
skolType = collectType skol where
    skol (TySkol idx) = [idx]
    skol _            = []

skolTypes :: [Type] -> [Int]
skolTypes = unionMap skolType

metaType :: Type -> [Int]
metaType = collectType meta where
    meta (TyMeta idx) = [idx]
    meta _            = []

metaTypes :: [Type] -> [Int]
metaTypes = unionMap metaType

metaVarE :: VarE -> [Int]
metaVarE = unionMap (metaType . unwrap) . Map.elems
    where unwrap (VarTy t) = t

-- Separate the type of a user-defined data constructor into a list of
-- argument types and the range type.  Used in pattern matching.

spine :: Type -> ([Type], Type)
spine = walk [] where
    walk ax (TyCon "->" [t1, t2]) = walk (t1:ax) t2
    walk ax t =
        if consP t then (reverse ax, t)
        else bug "Malformed data constructor type"

-- Check if a meta type variable appears in multiple elements of the
-- list of types.  Multiple occurrences in one element does not count.

multiP :: [Type] -> Int -> Bool
multiP tx = flip elem multi where
    skol = concat (map skolType tx)
    meta = concat (map metaType tx)
    tvs = meta ++ skol
    multi = nub (tvs \\ nub tvs)

--------- Pretty-printing types and constraints

instance Show Pat where
    show (PatCon "," ax) = paren (intercalate ", " ax)
    show (PatCon tc ax)  = unwords (tc:ax)
    show (PatInt i)      = show i

instance Show Type where
    show = showType

showType (TyVar tv)   = tv
showType (TyMeta idx) = ’?’ : show idx
showType (TySkol idx) = ’!’ : show idx
showType (TyCon "->" [t1, t2]) =
    unwords [showType1 t1, "->", showType t2]
showType (TyCon "," [t1, t2]) =
    paren (concat [showType1 t1, ", ", showType t2])
showType (TyCon tc ax) =
    unwords (tc : map showType2 ax)

paren :: String -> String
paren str = concat ["(", str, ")"]

showType1 t@(TyCon "->" _) = paren (showType t)
showType1 t = showType t

showType2 t@(TyCon "," _) = showType t
showType2 t@(TyCon _ (_:_)) = paren (showType t)
showType2 t = showType1 t

showLocal :: (Ident, VarTy) -> String
showLocal (x, vt) = unwords [x, "::", show vt]

instance Show VarTy where
    show (VarTy t) = showPolyType t

instance Show ConsTy where
    show (ConsTy t) = showPolyType t

showPolyType t = quant ++ show t where
    btvx = freeType t
    only ax s = if null ax then "" else s
    quant = only btvx (unwords ("forall" : btvx) ++ ". ")
```

B.4 SOURCE PROGRAM PARSER

The Front.g source file defines a parser for a simple functional programming language. I use the Frown parser generator [13] to produce a Haskell source file Front.hs that implements the parser specified in this source file.

```
-- Time-stamp: <2010-06-15 10:56:02 cklin>

module Front where

import Types
import Data.Char
import Data.List
import System.IO

%{
Terminal = LexOp   as "+"
         | LexDef  as "="
         | LexArr  as "->"
         | LexComa as ","
         | LexSemi as ";"
         | LexLam  as "\\"
         | LexType as "::"
         | LexParL as "(" | LexParR as ")"
         | LexBraL as "{" | LexBraR as "}"
         | LexVar {Ident}
         | LexCon {String}
         | LexInt {Integer}
         | LexData | LexWhere
         | LexCase | LexOf
         | LexLet  | LexIn
         | LexNext
         | LexDoc {String}
         | LexError ;

top     {[Program]} ;
top     {[]}                        : ;
        {p}                         | LexNext, top {p} ;
        {d:p}                       | decl {d}, LexNext, top {p} ;

decl    {Program} ;
decl    {Value v (foldr Lam t ax)}  : LexVar {v}, vars {ax}, "=", term {t} ;
        {Decl (Data c ax) cx}       | LexData, LexCon {c}, vars {ax},
                                      conb {cx} ;
        {Info doc}                  | docs {doc} ;

docs    {[String]} ;
docs    {[c]}                       : LexDoc {c} ;
        {c:cx}                      | LexDoc {c}, docs {cx} ;

vars    {[Ident]} ;
vars    {[]}                        : ;
        {v:ax}                      | LexVar {v}, vars {ax} ;
```

```
conb    {[Cons]} ;
conb    {[]}                         : ;
        {cx}                         | LexWhere, "{", cons {cx}, "}" ;

cons    {[Cons]} ;
cons    {[Cons c t]}                 : LexCon {c}, "::", tsig {t} ;
        {Cons c t:cx}                | LexCon {c}, "::", tsig {t}, ";",
                                       cons {cx} ;

tsig    {Type} ;
tsig    {t}                          : ftsig {t} ;
        {arrType t u}                | ftsig {t}, "->", tsig {u} ;

ftsig   {Type} ;
ftsig   {TyCon c ax}                 : LexCon {c}, stsig {ax} ;
        {t}                          | atsig {t} ;

stsig   {[Type]} ;
stsig   {[t]}                        : atsig {t} ;
        {t:tx}                       | atsig {t}, stsig {tx} ;

atsig   {Type} ;
atsig   {TyVar v}                    : LexVar {v} ;
        {TyCon c []}                 | LexCon {c} ;
        {TyCon "()" []}              | "(", ")" ;
        {TyCon "," [u, v]}           | "(", tsig {u}, ",", tsig {v}, ")" ;
        {t}                          | "(", tsig {t}, ")" ;

term    {Term} ;
term    {t}                          : sterm {t} ;
        {App (App (Var "+") t) u}    | sterm {t}, "+", term {u} ;
        {Lam a u}                    | "\\", LexVar {a}, "->", term {u} ;
        {Case c bx}                  | LexCase, term {c}, LexOf,
                                       "{", cases {bx}, "}" ;
        {Let dx t}                   | LexLet, "{", defs {dx}, "}",
                                       LexIn, term {t} ;

sterm   {Term} ;
sterm   {t}                          : aterm {t} ;
        {App t u}                    | sterm {t}, aterm {u} ;

aterm   {Term} ;
aterm   {Var v}                      : LexVar {v} ;
        {Con c}                      | LexCon {c} ;
        {Int i}                      | LexInt {i} ;
        {Con "()"}                   | "(", ")" ;
        {t}                          | "(", term {t}, ")" ;
        {App (App (Con ",") u) v}    | "(", term {u}, ",", term {v}, ")" ;

defs    {[(Ident, Term)]} ;
defs    {[(v, foldr Lam t ax)]}      : LexVar {v}, vars {ax}, "=", term {t} ;
        {(v, foldr Lam t ax):dx}     | LexVar {v}, vars {ax}, "=", term {t},
                                       ";", defs {dx} ;

cases   {[(Pat, Term)]} ;
cases   {[(p, t)]}                   : pat {p}, "->", term {t} ;
```

```
        {(p, t):cx}                 | pat {p}, "->", term {t},
                                      ";", cases {cx} ;

pat     {Pat} ;
pat     {PatCon c px}               : LexCon {c}, vars {px} ;
        {PatCon "," [u, v]}         | "(", LexVar {u}, ",", LexVar {v}, ")" ;
        {PatInt i}                  | LexInt {i} ;
}%

frown remain = fail (show (take 10 remain))

--------- Parse programs in files

parseFile :: String -> IO [Program]
parseFile file =
    do program <- readFile file
       top (segment program)

--------- Lexical analysis: top-level function

skiplf :: String -> String
skiplf = dropWhile ('\n' /=)

copydoc :: String -> ([String], String)
copydoc ('\n':cs@('-':'-':_)) = (doc:docs, rest)
    where (doc, wt) = span ('\n' /=) cs
          (docs, rest) = copydoc wt
copydoc cs = ([], cs)

segment :: String -> [Terminal]
segment [] = [LexNext]
segment ('\n':cs@('\n':'-':'-':_)) =
    let (doc, wt) = copydoc cs
    in LexNext : (map LexDoc doc ++ segment wt)
segment ('-':'-':cs) = segment (skiplf cs)
segment ('\n':'>':cs) = LexNext : segment (skiplf cs)
segment ('\n':'\n':'>':cs) = LexNext : segment (skiplf cs)
segment ('\n':cs@('\n':_)) = LexNext : segment cs
segment (c:cs) | isSpace c = segment cs
segment input =
    let (word, wt) = span isIdentChar input
        (symbol, st) = span isSymChar input
    in if null word
       then lexSymbol symbol ++ segment st
       else lexIdent word : segment wt

--------- Lexical analysis: symbols and operators

isIdentChar :: Char -> Bool
isIdentChar c = isAlphaNum c || c == '_'

isSymChar :: Char -> Bool
isSymChar c = not (isIdentChar c || isSpace c)

keysyms0 :: [(Char, Terminal)]
keysyms0 = [(';',  LexSemi), (',',  LexComa),
            ('\\', LexLam),
            ('(',  LexParL), (')',  LexParR),
            ('{',  LexBraL), ('}',  LexBraR)]

-- Note that here I use the same token, LexOp, to represent three
-- different integer binary operators.  This is hack, but as long as we
-- only type (but not evaluate) the programs, there really is no need to
-- distinguish the operators, which all have the same type.

keysyms :: [(String, Terminal)]
keysyms = [("+",  LexOp),  ("=",  LexDef),
           ("*",  LexOp),  ("/",  LexOp),
           ("->", LexArr), ("::", LexType)]

lexSymbol :: String -> [Terminal]
lexSymbol [] = []
lexSymbol symbol@(s:sx) =
    case lookup s keysyms0 of
      Just token -> token:lexSymbol sx
      Nothing -> case lookup symbol keysyms of
                   Just token -> [token]
                   Nothing -> [LexError]

--------- Lexical analysis: keywords and identifiers


keywords :: [(String, Terminal)]
keywords = [("data", LexData), ("where", LexWhere),
            ("case", LexCase), ("of",    LexOf),
            ("let",  LexLet),  ("in",    LexIn)]


lexIdent :: String -> Terminal
lexIdent word =
    if all isDigit word then LexInt (read word)
    else if isUpper (head word) then LexCon word
         else if isLower (head word) then
                  case lookup word keywords of
                    Just token -> token
                    Nothing -> LexVar word
              else LexError

```

B.5 TYPE INFERENCE MONAD

The Monad.hs source file defines a type inference monad Ti. It also provides some utility functions for the Ti monad.

```
-- Time-stamp: <2010-03-31 20:47:48 cklin>

module Monad where

import Control.Monad (liftM, mapM_, when)
import Data.Maybe (isJust)
import Types
import Common

type EndoTi a = a -> Ti a

--------- Type inference monad and its combinators

-- Type inference state consists of a sequence number for generating
-- fresh meta type variables and a list of strings that record diagnosis
-- and error messages in reverse chronological order.

type TiS a = (Int, [String], a)
type TiM a = TiS (Maybe a)

newtype Ti a =
    Ti { unTi :: Int -> TiM a }

instance Monad Ti where
    fail w   = Ti $ \s -> (s, ["ERROR: " ++ w], Nothing)
    return a = Ti $ \s -> (s, [], Just a)
    m >>= k  = Ti $ \s -> mapJust bind (unTi m s) where
        bind (s1, w1, v1) = (s2, w2 ++ w1, v2)
            where (s2, w2, v2) = unTi (k v1) s1

die :: Ti a
die = Ti $ \s -> (s, [], Nothing)

mapJust :: (TiS a -> TiM b) -> TiM a -> TiM b
mapJust _ (s, w, Nothing) = (s, w, Nothing)
mapJust f (s, w, Just a)  = f (s, w, a)

runTi :: Ti a -> Int -> a
runTi m s = a where (_, _, Just a) = unTi m s

-- Arrest any failure in a monadic computation.  The arrested
-- computation returns Nothing if a failure occurred.

catchTi :: Ti a -> Ti (Maybe a)
catchTi m = Ti $ m1 where
    m1 s = (s1, w, Just x)
        where (s1, w, x) = unTi m s

succeedTi :: Ti a -> Ti Bool
succeedTi = liftM isJust . catchTi . catchNotes

-- Unleash the inner Maybe monad.  Warning: all messages in the
-- attempted computations, error or otherwise, are discarded.  To
-- preserve the messages, set the verbose flag to True.

verbose = False

attemptTi :: [Ti a] -> Endo (Ti a)
attemptTi ax final = attempt ax where
    attempt [] = final
    attempt (m:mx) =
        do (w, result) <- arrestTi m
           when verbose (mapM_ (mesg . ("o " ++)) w)
           case result of
             Just a -> return a
             Nothing -> attempt mx

-- Generate fresh meta type variables, or just the serial number.

newMetaTv :: Ti Type
newMetaTv = liftM TyMeta newMetaIndex

newMetaIndex :: Ti Int
newMetaIndex = Ti $ \s -> (s+1, [], Just s)

freshenTv :: [a] -> Ti [Type]
freshenTv = mapM (const newMetaTv)

freshenIndex :: [a] -> Ti [Int]
freshenIndex = mapM (const newMetaIndex)

freshenTyCon :: EndoTi Type
freshenTyCon (TyCon tc ax) = liftM (TyCon tc) (freshenTv ax)
freshenTyCon v = bug ("Non-constructor type " ++ show v)

renameToNew :: [Int] -> Ti Rename
renameToNew xs = liftM (toMap . zip xs) (freshenIndex xs)

-- Write to or read from the internal messages store.  Unlike the fail
-- function, the mesg function writes a message without declaring a
-- failure.  The catchNotes function erases all messages, even if the
-- transformed computation fails.

mesg :: String -> Ti ()
mesg w = Ti $ \s -> (s, [w], Just ())

replay :: [String] -> Ti ()
replay = mapM_ mesg

arrestTi :: Ti a -> Ti ([String], Maybe a)
arrestTi = catchNotes . catchTi

catchNotes :: Ti a -> Ti ([String], a)
catchNotes m = Ti m1 where
    m1 s = (s1, [], liftM attach x)
        where (s1, w, x) = unTi m s
              attach a = (reverse w, a)

-- To ensure freshness of newly generated type variables in the presence
-- of hard-coded type variables in the Ti computation (for example, in
-- unit tests), we choose 100 as the initial sequence number.  The
-- programmer should make sure that all hard-coded meta type variables
-- have index numbers < 100.

initSeq :: Int
initSeq = 100

trapTi :: Ti a -> Maybe a
trapTi m = runTi (catchTi m) initSeq
```

```
examineTi :: Ti a -> IO (Maybe a)
examineTi m =
    let (w, v) = runTi (arrestTi m) initSeq
    in do mapM_ putStrLn w
          return v
```

B.6 TYPE SUBSTITUTION FUNCTIONS

The `Substitution.hs` source file provides type unification, type substitution combination, and other utility functions on type substitutions.

```
-- Time-stamp: <2010-05-14 11:08:06 cklin>

module Substitution where

import Data.List ((\\), union)
import qualified Data.Map as Map

import Types
import Common
import Monad

--------- General substitution utility functions

-- Construct a substitution.  This factory function checks that the
-- mapping is idempotent and reports an error otherwise.  Note that
-- compoSub (and perhaps other too) can produce identity mappings (e.g.,
-- [x/x]) in the associative list, so the algorithm must sanitize the
-- mapping (to mp0) to avoid tripping the idempotency checking.

makeSub :: [(Int, Type)] -> Subst
makeSub mp =
    let mp0 = filter (\(i, t) -> TyMeta i /= t) mp
        dom = map fst mp0
        rng = unionMap (metaType . snd) mp0
    in if overlaps dom rng
       then bug "Mapping is not idempotent"
       else toMap mp0

zeroSub :: Subst
zeroSub = Map.empty

oneSub :: Int -> Type -> Subst
oneSub = Map.singleton

nullSub :: Subst -> Bool
nullSub = Map.null

-- Compute the domain and the range of a substitution.

domSub :: Subst -> [Int]
domSub = Map.keys

rngSub :: Subst -> [Type]
rngSub = Map.elems

metaSub :: Subst -> [Int]
metaSub sub = domSub sub `union` metaTypes (rngSub sub)

-- Apply a substitution to a type.  Since substitution mappings are
-- idempotent, there is no need for iterative application.

zonk :: Subst -> Endo Type
zonk sub = replace where
    replace (TyCon tc ax) = TyCon tc (map replace ax)
    replace t@(TyMeta i) = Map.findWithDefault t i sub
    replace (TySkol i) | Map.member i sub =
        bug "Substitution on Skolem type"
    replace t = t

-- Apply a substitution directly to a meta type variable index.

zonkIndex :: Subst -> Int -> Type
zonkIndex sub i = Map.findWithDefault (TyMeta i) i sub

-- Rename meta type variables in a type.  Unlike substitutions, this
-- dedicated renaming function does not care about the orientation of
-- type variable renaming.

renameMeta :: Rename -> Endo Type
renameMeta ren = replace where
    replace (TyCon tc ax) = TyCon tc (map replace ax)
    replace (TyMeta i) = TyMeta (lookupZ i ren)
    replace t = t

-- Replace free type variables with (supposedly fresh) meta type
-- variables.  Not strictly a substitution, but ...

instType :: Map.Map Ident Int -> Endo Type
instType inst = replace where
    replace (TyCon tc ax) = TyCon tc (map replace ax)
    replace (TyVar tv) = TyMeta (lookupX tv inst)
    replace t = t

-- Compute a substitution that forces the second type (t) to have the
-- same top-level type constructor as the first type (c) .

imprintTc :: Type -> Type -> Ti Subst
imprintTc c t =
    do u <- freshenTyCon c
       unify2 u t

-- The skolemize function computes a substitution that replaces meta
-- type variables with Skolem types that have the same indices.  The
-- unskolemize function replace Skolem types to their corresponding meta
-- type variables.

skolemize :: [Int] -> Subst
skolemize = makeSub . map skol
    where skol m = (m, TySkol m)

unskolemize :: Endo Type
unskolemize = replace where
    replace (TyCon tc ax) = TyCon tc (map replace ax)
    replace (TySkol i) = TyMeta i
    replace t = t

-- Compose two substitutions.  The function uses nub1st to arbitrate
-- (favoring sub2) when the domains overlap.  Composition is not
-- commutative, and tv(rng(sub1)) must be disjoint from dom(sub2) with
-- the exception of reverse renaming (see example).
-- zonk sub1 (zonk sub2 t) == zonk (compoSub sub1 sub2) t
-- compoSub [x/y] [y/x] == [x/y]

compoSub :: Subst -> Endo Subst
compoSub sub1 sub2 = makeSub (nub1st (mp2 ++ mp1)) where
    mp1 = Map.toList sub1
    mp2 = Map.toList (Map.map (zonk sub1) sub2)

compoSubs :: [Subst] -> Subst
compoSubs = foldl compoSub zeroSub

-- Restrict the domain of a substitution.

restrictSub :: [Int] -> Endo Subst
restrictSub mx = Map.filterWithKey relevant
    where relevant i _ = i ‘elem‘ mx

-- Apply an idempotent variable renaming to a both the domain and the
-- range of a substitution.  Here is an example:
-- switchMetaSub [a/x, b/y] [T x/y] == [T a/b]

switchMetaSub :: Rename -> Endo Subst
switchMetaSub ren = makeSub . map switch . Map.toList
    where switch (i, t) = (lookupZ i ren, renameMeta ren t)

-- Restrict an idempotent substitution to the parts that have nontrivial
-- effects on the given set of type variables.  More specifically, the
-- function restricts the domain to the type variables of interest, and
-- then it eliminates trivial type variable mappings (i.e., renaming).

shaveSub :: [Int] -> Endo Subst
shaveSub mx sub = shaven where
    trimmed = restrictSub mx sub
    nontriv = multiP (rngSub trimmed)
    keep (TyMeta i) = nontriv i || elem i mx
    keep _          = True
    shaven = Map.filter keep trimmed

-- Extend a substitution such that every meta type variable in the input
-- list dom is in the domain of the extended subsitution.

divertSub :: [Int] -> EndoTi Subst
divertSub dom sub =
    do let gap = dom \\ domSub sub
```

```
       fresh <- freshenTv gap
       let fill = makeSub (zip gap fresh)
       return (compoSub fill sub)

--------- Plain unification functions

unify :: [Type2] -> Ti Subst
unify [] = return zeroSub
unify ((t1, t2):tx) =
    do this <- unify2 t1 t2
       let norm (u1, u2) = (zonk this u1, zonk this u2)
       rest <- unify (map norm tx)
       return (compoSub rest this)

unify2 :: Type -> Type -> Ti Subst
unify2 (TyVar _) _ = bug "Bound type variable in unify2"
unify2 _ (TyVar _) = bug "Bound type variable in unify2"
unify2 (TyMeta i1) (TyMeta i2) | i1 == i2 = return zeroSub
unify2 (TyMeta i1) t2 = unifyMeta i1 t2
unify2 t1 (TyMeta i2) = unifyMeta i2 t1
unify2 (TySkol i1) (TySkol i2) | i1 == i2 = return zeroSub
unify2 (TySkol _) _ = fail "Cannot unify with a Skolem type"
unify2 _ (TySkol _) = fail "Cannot unify with a Skolem type"
unify2 (TyCon tc1 ax1) (TyCon tc2 ax2) =
    if tc1 == tc2 && length ax1 == length ax2
    then unify (magic $ zip ax1 ax2)
    else fail "Cannot unify different type constructors"

unifyMeta :: Int -> Type -> Ti Subst
unifyMeta i t =
    if elem i (metaType t)
    then fail "Unification produces an infinite type"
    else return (oneSub i t)

unifyTypes :: [Type] -> Ti Subst
unifyTypes [] = unify []
unifyTypes tx = unify (zip (tail tx) tx)

-- Unifiability testing.  If the given type equations / types are
-- satisfiable, return a most-general unifier as evidence.  The function
-- uses trapTi to present a non-monadic interface.

unifiable :: [Type2] -> Maybe Subst
unifiable = trapTi . unify

unifiableTypes :: [Type] -> Maybe Subst
unifiableTypes = trapTi . unifyTypes

--------- Substitution unification algorithm

-- Compute a most-general common instance of the input substitutions.
-- This algorithm is so much simpler than McAdam's substitution
-- unification, and it extends naturally to more than two substitutions.

combineSub :: Subst -> EndoTi Subst
combineSub sub1 sub2 = combineSubs [sub1, sub2]
```

```
combineSubs :: [Subst] -> Ti Subst
combineSubs = unify . map1st TyMeta
            . concatMap (magic . Map.toAscList)

--------- Substitution orientation bias

-- The "magic" switch controls how the algorithm orients type
-- substitutions, which in turn affects how it picks type parameters and
-- type indices by breaking the symmetry in scrutinee and pattern types.
-- With magic turned off (i.e., magic = id), the algorithm exhibits the
-- bias that type indices come before type parameters.  With magic
-- turned on (i.e., magic = reverse), the algorithm exhibits the
-- opposite bias: type parameters come before type indices.  The magic
-- does not formally change the expressiveness of Algorithm P, but it
-- does seem to fit currently programming practices better (and it
-- allows the implementation to infer expected types for both runState_o
-- and fdComp1).

magic = reverse
```

B.7 BRANCH TYPE INFERENCE

The Branches.hs source file provides utility functions that support type inference for GADT pattern-matching branches.

```
-- Time-stamp: <2010-05-12 14:23:36 cklin>

module Branches where

import Control.Monad (liftM, unless)
import Data.List ((\\), transpose)
import Data.Maybe (isJust, mapMaybe)
import qualified Data.Map as Map

import Types
import Common
import Monad
import Substitution

--------- Branch type manipulation functions

-- Create a minimal (i.e., most general) type that has all the specified
-- meta type variables in the same position as the input type.

transcribe :: [Int] -> EndoTi Type
transcribe keep = render where
    render t@(TyMeta i) =
        if i `elem` keep
        then return t else newMetaTv
    render (TyCon tc ax) =
        if metaTypes ax `overlaps` keep
        then liftM (TyCon tc) (mapM render ax)
        else newMetaTv
    render (TyVar _) =
        bug "Bounded type variable in transcribe"
    render (TySkol _) =
        bug "Skolem type in transcribe"

-- Compute a scrutinee type from the branch scrutinee type templates and
-- type indexing substitutions.  We use the extractTc function to
-- specialize the scrutinee type, which ensures that GADT type
-- refinements for each scrutinee type variable do not all have the same
-- top-level type constructor.

scrutineeType :: [Subst] -> [Type] -> Ti Subst
scrutineeType sub_branch scrutinees =
    do unifier <- unifyTypes scrutinees
       sub <- reach (mapM (combineSub unifier) sub_branch)
       let indices = metaType (zonk unifier $ head scrutinees)
           indexing = map (branchTypes sub) indices
       extract <- liftM fst (extractTc indexing)
       return (compoSub extract unifier)

-- The Split type describes how a meta type variable (represented by the
-- Int value) appears in each pattern-matching branch, and the
-- branchTypes function is its factory function.

type Split = (Int, [Type])

branchTypes :: [Subst] -> Int -> Split
branchTypes sub i = (i, indexed) where
    indexed = map (flip zonkIndex i) sub

zonkSplit :: Subst -> Endo Split
zonkSplit sub (i, tx) = (i, map (zonk sub) tx)

-- The extractTc function extracts common top-level type constructors in
-- the branch types and apply them to the corresponding type variables.
-- extractTc [(x, [Int])] == ([Int/x], [])
-- extractTc [(x, [T Int, T Bool])] == ([T y/x], [(y, [Int, Bool])])
-- extractTc [(x, [Bool, T Int])] == ([], [(x, [Bool, T Int])])

extractTc :: [Split] -> Ti (Subst, [Split])
extractTc [] = return (zeroSub, [])
extractTc ((i, tx) : ux) =
    case deCons tx of
      Nothing ->
          do (sub, vx) <- extractTc ux
             return (sub, (i, tx) : ux)
      Just (tc, ax) ->
          do ix <- freshenIndex (head ax)
             let newTc = TyCon tc (map TyMeta ix)
                 uax = zip ix (transpose ax)
             (sub, vx) <- extractTc (uax ++ ux)
             return (compoSub sub (oneSub i newTc), vx)

-- The matchTc function is similar to extractTc, except that it requires
-- only that the branch types be coercible to a single type constructor.
-- The function reports failure if the condition is not met.

matchTc :: Split -> Ti (Subst, [Split])
matchTc (i, tx) =
    case deCons (filter consP tx) of
      Nothing -> fail "Cannot match type constructors"
      Just (tc, ax) ->
          do let template = TyCon tc (head ax)
             imprint <- mapM (imprintTc template) tx
             let imprinted = zipWith zonk imprint tx
             (inst, vx) <- extractTc [(i, imprinted)]
             sub <- combineSubs (inst : imprint)
             return (sub, vx)

-- Skolemize all occurrences of pattern type variables to prevent the
-- type indexing process from introducing unintentional instantiations
-- (which can create new generalized existential types == BAD).

freeze :: [Int] -> Endo ([Split], [Split])
freeze excl (indexing, env) = (skol indexing, skol env) where
    pattern_v = unionMap (metaTypes . snd) indexing \\ excl
    skol = map (zonkSplit (skolemize pattern_v))

equalize :: Split -> Ti Subst
equalize (i, tx) =
    if null (skolTypes tx)
    then attemptTi [unifyTypes (TyMeta i : tx)]
         (fail "Cannot reconcile branch body types")
    else fail "A pattern type escapes in equalize"

-- Reconcile the mapping from environment meta type variables to branch
-- types into a single type substitution with the help of type indexing.

reconcile :: [Int] -> ([Split], [Split]) -> Ti (Subst, [Split])
reconcile params = driver zeroSub . freeze params where

    -- The driver function repeatedly invokes the matcher until it
    -- reaches a fixed-point (at which point the matcher cannot make
    -- more progress), and then it returns the combined matching along
    -- with the unsolved branch types.

    driver prev (indexing, variety) =
        do result <- mapM matcher variety
           sub <- combineSubs (map fst result)
           let update = map (zonkSplit sub)
               unsolved = update (concatMap snd result)
           combined <- combineSub prev sub
           let param_inst = map (zonkIndex combined) params
           unless (null (skolTypes param_inst))
                  (fail "A pattern type escapes in reconcile")
           if nullSub sub
              then return (combined, unsolved)
              else driver combined (update indexing, unsolved)

    -- The matcher function tries to reconcile pattern types that are
    -- not unifiable by either matching them to a unique type index or
    -- matching (and unwrapping) a top-level type constructor.
```

```
     where
      matcher variety@(_, tx) =
        if isJust (unifiableTypes tx) && null (skolTypes tx)
        then return (zeroSub, [variety])
        else attemptTi [matchIndex variety, matchTc variety]
                       (return (zeroSub, [variety]))

      matchIndex (i, tx) =
        let runSnd f (x, y) = do { z <- f y ; return (x, z) }
            couple = runSnd (unifiable . zip tx)
        in case mapMaybe couple indexing of
             [(k, sub)] -> let index = oneSub i (TyMeta k)
                           in return (compoSub index sub, [])
             _ -> fail "Cannot find unique matching type index"

--------- Branch reachability constraints

-- We use Type2 to represent the type-level (potential) reachability of
-- a branch.  The first component is the scrutinee type, and the second
-- component is the pattern type.  The Reach type represents the
-- reachability constraints in disjunctive normal form: nested patterns
-- are linked by conjunctions, and non-nested patterns (whether in the
-- same case expression or not) are linked by disjunctions.

type Reach = [[Type2]]

-- A trivial reachability constraint is the disjunction of an empty
-- conjunction (True), which differs from an empty disjunction (False).

trueR :: Reach
trueR = [[]]

-- Since inferred type substitutions should never affect the pattern
-- types, the zonkR function applies the substitution only to the
-- scrutinee types (first component of the pair).

zonkR :: Subst -> Endo Reach
zonkR = map . map1st . zonk

-- Add a nested pattern to a reachability constraint.

attachR :: Type2 -> Endo Reach
attachR r = map (r:)

-- Check the satisfiability of a reachability constraint: type equations
-- in each conjunction must be simultaneously unifiable.

checkR :: [Int] -> EndoTi Reach
checkR ix r =
    do reach (mapM_ unify r)
       return r

reach :: Endo (Ti a)
reach m = attemptTi [m] (fail "A branch is unreachable")
```

B.8 PROGRAM TYPE INFERENCE

The `Inference.hs` source file implements Algorithm $\mathcal{P}$.

```
-- Time-stamp: <2010-05-05 15:07:22 cklin>

module Inference (inferTop) where

import Control.Monad
import Data.List ((\\), intersect, nubBy, union)
import qualified Data.Map as Map

import Types
import Common
import Monad
import Substitution
import Branches

--------- Type instantiation and generalization

instantiate :: EndoTi Type
instantiate t =
    do let btvx = freeType t
       inst <- liftM (zip btvx) (freshenIndex btvx)
       return (instType (toMap inst) t)

names :: [String]
names = expand ("" : names) (map (:[]) [’a’..’z’])
    where expand ax bx = [ a ++ b | a <- ax, b <- bx ]

generalize :: [Int] -> Endo Type
generalize outer t = zonk (toMap gen) t where
    gen = zip (metaType t \\ outer) (map TyVar names)

--------- Constructor and variable type lookup functions

lookupCons :: ConsE -> String -> Ti Type
lookupCons consE dc =
    case Map.lookup dc consE of
      Nothing -> fail ("Unknown constructor " ++ dc)
      Just (ConsTy t) -> instantiate t

lookupVar :: VarE -> Ident -> Ti Type
lookupVar varE v =
    case Map.lookup v varE of
      Nothing -> fail ("Unknown variable " ++ v)
      Just (VarTy t) -> instantiate t

--------- Type inference for top-level definitions

inferTop :: ConsE -> Term -> Ti Type
inferTop consE e =
    do let vx = ["+", "undefined"]
           tx = [plusType, botType]
           env = (consE, vars vx tx)
       (_, _, t) <- inferType env e
       return (generalize [] t)

--------- Main type inference algorithm

type Env = (ConsE, VarE)
type Result = (Subst, Reach, Type)

var :: Ident -> Type -> VarE
var x t = Map.singleton x (VarTy t)

vars :: [Ident] -> [Type] -> VarE
vars xs = Map.map VarTy . toMap . zip xs

zonkE :: Subst -> Endo VarE
zonkE = mapVarE . zonk

simple :: Type -> Result
simple t = (zeroSub, trueR, t)

inferType :: Env -> Term -> Ti Result
inferType (consE, varE) = infer where
    showEnv = mapM_ (mesg . showLocal) . Map.assocs
    metaE = metaVarE varE

    -- Type inference in an extended variable type environment.  I use
    -- the left-bias property of Map.union to implement shadowing.

    inferVar newE =
        inferType (consE, Map.union newE varE)

    -- This wrapper function applies the inferred substitution to the
    -- reachability constraint, and check that the refined reachability
    -- constraint is satisfiable.

    infer e =
        do (sub, r, t) <- infer' e
           trimmed <- checkR (metaType t ++ metaE) (zonkR sub r)
           return (restrictSub metaE sub, trimmed, t)

    infer' (Int _) = return (simple intType)
    infer' (Con c) = liftM simple (lookupCons consE c)
    infer' (Var v) = liftM simple (lookupVar varE v)

    -- This type inference algorithm for function application is taken
    -- from the paper "On the Unification of Substitutions in Type
    -- Inference" by Bruce J. McAdam.

    infer' (App f e) =
        do (sub_f, r_f, t_f) <- infer f
           (sub_e, r_e, t_e) <- infer e
           t_r <- newMetaTv
           sub_r <- unify2 t_f (arrType t_e t_r)
           sub <- combineSubs [sub_f, sub_e, sub_r]
           return (sub, union r_f r_e, zonk sub t_r)

    infer' (Lam u e) =
        do t_u <- newMetaTv
           (sub, r, t_e) <- inferVar (var u t_u) e
           return (sub, r, arrType (zonk sub t_u) t_e)

    infer' (Let l e) =
        do (sub_l, r_l, vE) <- inferLocal l
           showEnv vE
           (sub_e, r_e, t_e) <- inferVar vE e
           sub <- combineSub sub_l sub_e
           return (sub, union r_l r_e, zonk sub t_e)

    infer' (Case e w) =
        do (sub_e, r_e, t_e) <- infer e
           (sub_wx, r_w, i_b, t_s) <- inferBranches w

           -- Combine the inferred type of the case scrutinee with the
           -- requirement from the pattern-matching branches.  The flip
           -- in the computation for sub_branches is important to avoid
           -- type variable renaming from scrutinee type variables.

           unifier <- unify2 t_e t_s
           let scrutinee = zonk unifier t_s
           sub_ctx <- combineSub sub_e unifier
           sub_branches <- mapM (flip combineSub sub_ctx) sub_wx

           -- Fill in the empty (unconstrained) positions in the branch
           -- type substitutions.

           let dom     = unionMap domSub sub_branches
               indices = metaType scrutinee `intersect` dom
               params  = metaType scrutinee \\ indices
               i_body  = i_b : metaE
               labels  = indices `union` i_body
           map_wx <- mapM (divertSub labels) sub_branches

           -- Compute how GADT type refinement in each branch affects
           -- the scrutinee and the environment / body type variables.

           let indexing = map (branchTypes map_wx) indices
               branches = map (branchTypes map_wx) i_body

           -- Compute a single type substitution to represent the
           -- varying type assumptions inferred from each
           -- pattern-matching branch.  This is the part of the
           -- algorithm that takes advantage of type indexing.

           (match, unmatched) <- reconcile params (indexing, branches)
           tie <- (combineSubs <=< mapM equalize) unmatched

           sub <- combineSub match tie
           return (sub, union r_e r_w, zonkIndex sub i_b)

    -- Infer the types of pattern matching branches.  This function
    -- computes a single scrutinee type for all branches, but it does
    -- not attempt to resolve differences in environment and branch body
    -- types (which may require using type indexing).

```

```
inferBranches :: [Branch] -> Ti ([Subst], Reach, Int, Type)
inferBranches branches =
    do i_body <- newMetaIndex
       inferred <- mapM (inferBranch i_body) branches
       let (sub, rx, scrutinees) = unzip3 inferred

       unifier <- scrutineeType sub scrutinees

       let r = zonkR unifier (unions rx)
           scrutinee = zonk unifier (head scrutinees)
       unified_sub <- mapM (flip combineSub unifier) sub
       return (unified_sub, r, i_body, scrutinee)

-- Infer the type of a single pattern-matching branch.  The integer
-- i represents a placeholder meta type variable for the branch body
-- type; see comment at the end of the function for the meaning of
-- the returned substitution.

inferBranch :: Int -> Branch -> Ti (Subst, Reach, Type)
inferBranch i (p, e) =
    do (newE, pattern) <- patternType p
       (sub_e, r_e, t_e) <- inferVar newE e

       -- To avoid false positives in recognizing generalized
       -- existential types, we trim the inferred type substitution
       -- sub_e to retain only the mappings that are relevant.

       let i_e = metaType t_e
           i_local = metaVarE newE
           i_env = unions [metaE, i_local, i_e]
           i_pat = metaType pattern
           eta = shaveSub i_env sub_e
           eta_local = shaveSub i_pat eta

       -- Compute the set v_esc of instantiated or escaped pattern
       -- type variables and check for existential type violations.

       let i_ext = intersect (metaSub eta ‘union‘ i_e) i_local
       unless (i_ext ‘subset‘ i_pat)
              (fail "Existential type escape or instantiation")

       -- Compute a most general scrutinee type by transcribing
       -- instantiated type variables from the pattern type to the
       -- scrutinee type.  Use the computed scrutinee type to update
       -- the reachability constraint for the branch.

       copy <- transcribe (metaSub eta_local) pattern
       scrutinee <- if metaP copy
                    then freshenTyCon pattern
                    else return (zonk eta copy)
       let r = attachR (scrutinee, pattern) r_e

       -- The ordering of "pattern" and "scrutinee" matters: placing
       -- the pattern type first avoids generating trivial renaming
       -- from scrutinee type variables to pattern type variables.
       -- Such renaming makes the type inference algorithm mistake
       -- ADT type variables for GADT type variables and could cause
       -- type inference failure for some ADT case expressions.

       refinement <- unify2 pattern scrutinee

       -- The substitution sub combines two different kinds of type
       -- information into one.  With domain restricted to the
       -- scrutinee type variables, it represents GADT type
       -- refinements for the branch (sub_ps).  With domain
       -- restricted to the outer environment type variables, it
       -- represents the assumptions that the branch makes on the
       -- environment (eta).  With domain restricted to i, it
       -- represents the type of the branch body.

       sub <- combineSubs [refinement, eta, oneSub i t_e]
       return (sub, r, scrutinee)

-- Given a pattern, compute a pattern type and construct a
-- corresponding type environment fragment for the pattern-bound
-- variables.

patternType :: Pat -> Ti (VarE, Type)
patternType (PatInt _) = return (Map.empty, intType)
patternType (PatCon c xs) =
    do (t_ax, t_p) <- liftM spine (lookupCons consE c)
       unless (length t_ax == length xs)
              (fail "Data constructor arity mismatch")
       return (vars xs t_ax, t_p)

-- Infer polymorphic types for recursive local let definitions using
-- Mycroft's extension to Algorithm W (described in "Polymorphic
-- Type Schemes and Recursive Functions", 1984).  The function
-- returns, among other things, a type environment fragment for
-- local definitions.

inferLocal :: [(Ident, Term)] -> Ti (Subst, Reach, VarE)
inferLocal locals = mycroft limit zeroSub fullpoly where
    (ux, ex) = unzip locals
    fullpoly = repeat (TyVar "a")

    -- Since Mycroft's fix rule is only a semi-algorithm, we must
    -- enforce termination by limiting the number of iterations.
    -- The limit defined here is fairly conservative; a limit of 3
    -- is sufficient for all the included examples.

    limit = 20

    mycroft 0 _ _ = fail "Mycroft iteration limit reached"
    mycroft n init types =
        do result <- arrestTi (refinePoly init types)
           case result of
             (w, Nothing) -> do { replay w ; die }
             (w, Just (sub, r, sigma)) ->
                 if sigma /= map (zonk sub) types
                 then mycroft (n-1) sub sigma
                 else do replay w
                         return (sub, r, vars ux sigma)
```

```
-- Infer polymorphic types for local definitions under a
-- substitution and a polymorphic type environment.  Unlike in
-- the Hindley-Milner type inference algorithm, we do not unify
-- the types in the environment and the inferred types of the
-- local definitions.

refinePoly sub_init types_init =
    do let env_init = vars ux types_init
       inferred <- mapM (inferVar env_init) ex
       let (subs, rx, tx) = unzip3 inferred
       sub <- combineSubs (sub_init : subs)
       let outer = metaVarE (zonkE sub varE)
           types_new = map (zonk sub) tx
           sigma_new = map (generalize outer) types_new
       return (sub, unions rx, sigma_new)
```