Principal Type Inference for GADTs [*]

Sheng Chen
CACS, UL Lafayette, USA
chen@louisiana.edu

Martin Erwig
Oregon State University, USA
erwig@oregonstate.edu

Abstract

We present a new method for GADT type inference that improves the precision of previous approaches. In particular, our approach accepts more type-correct programs than previous approaches when they do not employ type annotations. A side benefit of our approach is that it can detect a wide range of runtime errors that are missed by previous approaches.

Our method is based on the idea to represent type refinements in pattern-matching branches by choice types, which facilitate a separation of the typing and reconciliation phases and thus support case expressions. This idea is formalized in a type system, which is both sound and a conservative extension of the classical Hindley-Milner system. We present the results of an empirical evaluation that compares our algorithm with previous approaches.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications–Applicative (functional) languages; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs–Functional constructs, Type structure

General Terms Languages, Theory

Keywords Type Inference, GADT, Choice Type, Variational Unification, Type Reconciliation

1. Introduction

Generalized algebraic data types (GADTs) extend algebraic data types by allowing different data constructors to refine the result type differently. In accordance, a pattern-matching branch is allowed to bring in local assumptions that are effective only in that branch. This type system extension enables programmers to encode interesting properties and invariants about programs or data structures within types, which will then be checked and enforced during compile time, precluding a large class of runtime errors [6, 22, 30, 36]. Since their inception, GADTs have been adopted for many programming tasks, for example, generic programming [28], monad libraries [16], balanced trees [29, 35], and tagless language interpreters [36].

This seemingly simple extension brings up many tricky issues in GADT type inference, which after a decade of active research [10, 17, 18, 21, 22, 25, 27, 30, 32, 34] still remains an open problem [17]. The fundamental challenge results from type refinements in pattern-matching branches, which make some case expressions whose branches have different types well-typed.[1] Reconciling different types in accordance with type refinements is particularly hard. Worse, type refinements cause some programs to have different most general types, breaking the fundamental principle that underlies type inference. Consider, for example, the following program reproduced from [34]. (We have shortened the constructor names from the original paper to `RI`, `RB`, and `RC`, respectively.)

```
data R a where
  RI :: Int  -> R Int
  RB :: Bool -> R Bool
  RC :: Char -> R Char

flop1 (RI x) = x
```

Here the data type `R a` is refined to `R Int`, `R Bool`, and `R Char`, respectively, with the corresponding data constructor. The function `flop1` can be assigned many types. One possible type is $\forall\alpha.\texttt{R}\ \alpha \to \texttt{Int}$ since we can always generalize the argument type from `R Int` to `R` α as we usually do in ADT (algebraic data type) systems. The type $\forall\alpha.\texttt{R}\ \alpha \to \alpha$ is also a candidate in the presence of local assumptions. In this case pattern matching introduces the local assumption that maps α to `Int`, allowing the body to have the type α, which is `Int` under the local assumption. We observe that neither of the two candidates is more general than the other. The question is then: Which type should be inferred for `flop1`?

An obvious solution is to ask programmers for annotations, an approach taken in much of the previous work [21, 22, 25, 27, 30, 34]. However, this strategy has several shortcomings. First, as already noted in [10, 17], we lack precise and simple rules about where type annotations are needed. Thus, programmers may have to annotate all case expressions. Second, programmers are told to write annotations first [8, 26], even when the intention of a function is still in flux. As a result, annotations can often be incorrect [3], which has a negative impact on type inference.

The question is then: How shall we deal with the loss of the principality in GADT type inference, or more specifically, what should be the type of `flop1`? Lin [17] and Vytiniotis et al. [34] have argued that the best type for `flop1` is `R Int -> Int`. Although this type is not the most general type, it best describes the shape of `flop1` since it statically rejects expressions such as `flop1 (RB True)` or `flop1 (RC 'a')`. These expressions are well typed if

[*] This work is supported by the National Science Foundation under the grants CCF-1219165 and IIS-1314384.

POPL'16, January 20–22, 2016, St. Petersburg, FL, USA
© 2016 ACM. 978-1-4503-3549-2/16/01...$15.00
http://dx.doi.org/10.1145/2837614.2837665

[1] Another challenge is that GADT programs use polymorphic recursion extensively, but type inference with polymorphic recursion has been shown to be undecidable [11, 14]. We will not discuss this problem here.

```
data G a b where
  G1 :: a -> G Int a
  G2 :: a -> G Bool a

param1o e =
  case e of
    G1 i -> i+3
    G2 b -> case b of
              True -> 4
              False-> 7
```

(a) GADT expression `param1o`

```
Couldn't match expected type t with actual
type t2
  t is untouchable
    inside the constraints (t1 ~ Int)
    bound by a pattern with constructor
        G1 :: forall a. a -> G Int a,
      in a case alternative
... lines 7 through 24 omitted

Couldn't match expected type t2 with actual
type Bool
  t2 is untouchable
    inside the constraints (t1 ~ Bool)
    bound by a pattern with constructor
        G2 :: forall a. a -> G Bool a,
      in a case alternative
... lines 32 through 44 omitted
```

(b) Output of GHC

```
Error:
This pattern matches values of type (bool, 'a) g
but a pattern was expected which matches values
    of type (int, 'b) g
Type bool is not compatible with type int
```

(c) Output of Ambivalent

```
Type inference failed for param1o
ERROR: Cannot unify different type constructors
```

(d) Output of $\mathscr{P}$

```
The expression param1o has the type
before reconciliation:
A⟨G Int Int -> Int, G Bool Bool -> B⟨Int,Int⟩⟩
after reconciliation:
    G n n -> Int
```

(e) Output of Chore

Figure 1: A non-annotated GADT expression for which only the new approach "Chore" infers a correct type.

`flop1` receives the type $\forall\alpha.$`R` $\alpha \to$ `Int` or $\forall\alpha.$`R` $\alpha \to \alpha$, but they will always lead to runtime failures.

However, what should be the type of the following similar function [34]?

```
flop2 e = case e of
            RI x -> x
            RB x -> x
```

Lin [17] assigns the type $\forall\alpha.$`R` $\alpha \to \alpha$, which makes sense but doesn't preclude the application `flop2 (RC 'a')` at compile time, losing the benefit of detecting errors statically. Vytiniotis et al. [34] concluded that this situation can't be improved unless the type syntax is extended.

To address this issue, we propose to extend the type syntax with a *choice* construct, which has had several successful applications [2, 5, 13, 15]. With choice types, we can assign the following type to `flop2`.

$$\texttt{flop2} : D\langle\texttt{R Int},\texttt{R Bool}\rangle \to D\langle\texttt{Int},\texttt{Bool}\rangle$$

The choice type $D\langle$`R Int`,`R Bool`$\rangle$ expresses that the argument type can be one of its *alternative*s, that is, the argument type can either be `R Int` or `R Bool`. Moreover, the type says that the result type is `Int` when the argument type is `R Int` and `Bool` when the argument type is `R Bool`. This correlation is established through the use of the same choice name D in the both argument and result type. Here D gives a name to control the variation between two types. All variations under the same name are synchronized in the sense that the same decision should be made about choosing variants. With this precise characterization, we can now reject applications such as `flop2 (RC 'a')` because the type of the argument, which is `R Char`, matches neither `R Int` nor `R Bool`.

1.1 Principal Type Inference

In general, the use of choice types helps to restore principality in GADT type inference. Type inference works as follows. We first infer principal types for case branches. Then we put branch types into choices to form principal types for case expressions. Choice types and types for other parts of the program are put together using a set of rules dealing with choices. Thus, there is no need to address the hard question of finding the "best" type for each case expression.

Since all previous approaches attempt to assign a single type to each expression [10, 17, 18, 21, 22, 25, 27, 30, 32, 34], they face the challenge of assigning appropriate types to case expressions that may have conflicting branch types. As a result, most previous approaches have to reject such expressions although they are well typed. For example, although branches of `flop2` have conflicting types `R Int` $\to$ `Int` and `R Bool` $\to$ `Bool`, `flop2` is well typed and can be assigned the type `R` $\alpha \to \alpha$. The only exception is Lin [17], who first computes branch types and then reconciles them to get a type. However, reconciliation is performed separately for each case expression, which can cause the loss of type information that is important for the type inference of other program parts. As a result, many well-typed programs are rejected. A detailed comparison with this approach follows in Sections 7 and 9.

The use of choice types offers two fundamental advantages. First, choice types don't force premature typing decisions in the context of incomplete information. In contrast, they allow us to *delay typing decisions* until sufficient typing information is available. This aspect of choice types was exploited successfully already in an approach for improved type error debugging [2]. In this paper, we employ this idea to facilitate a more informed choice reconciliation. Second, choice types *delimit the boundaries of branch types*, which can thus be computed independently of one another. In general, choice types support localized computations. A particular problem of previous GADT type inference approaches is that they try to fit potentially conflicting computations into one global context, which requires computations to be consistent, a condition that GADT type inference violates in general. Consider again `flop2` as an example. Without choice types, the global computational context requires the scrutinee `e` to have both the types `R Int` and `R Bool`, which causes a type conflict. With choice types, localized computation contexts require `e` to have the type `R Int` and `R Bool` respectively, which can be satisfied by assigned `e` the type $D\langle$`R Int`,`R Bool`$\rangle$.

To illustrate these advantages with a concrete example, consider the expression `param1o` in Figure 1a, which was first introduced in [17]. Its subtlety is that the type refinements brought in through pattern matching are applied not to the bodies of the case branches, but to the scrutinee `e`.

For this expression, GHC produces the message shown in Figure 1b. Since OutsideIn [34], the type inference algorithm of GHC, applies type refinements only when type annotations are present, it is not surprising that GHC rejects this expression. For brevity, we have omitted much of the message, which is mainly about the rigidity of type variables and the binding information. GHC will accept this expression if we annotate it with a correct type, for example `G a a -> Int`.

The next algorithm we consider is Ambivalent [10], implemented in OCaml 4.01.0.[2] The main aim of Ambivalent is to reduce the amount of type annotation for GADT expressions, but it fails for this expression. (The type `(bool,'a) g` in the message corresponds to `G Bool a` in Haskell.)

Although the algorithm $\mathscr{P}$ [18] was designed specifically for inferring types for GADT expressions without type annotations, it fails for this expression with the output shown in Figure 1d. The reason is that $\mathscr{P}$ applies refinements to the body of case expressions, while this expression needs to apply them to the scrutinee.

For this expression, our approach successfully infers a type. Before reconciliation, the type of `param1o` is as follows.

$$A\langle\texttt{G Int Int -> Int},\texttt{G Bool Bool -> }B\langle\texttt{Int},\texttt{Int}\rangle\rangle$$

Here the choices A and B are created for the case expressions with the scrutinees `e` and `b`, respectively. Thus, we derive that the first case branch has the type `G Int Int -> Int`, and the second has the type `G Bool Bool ->` $B\langle\texttt{Int},\texttt{Int}\rangle$.

Since choice types can be arbitrarily nested, their use can complicate the communication of types to programmers. We address this issue by providing the option of removing choices in the reported types and converting them into corresponding types in conventional syntax. We refer to this process as choice reconciliation and will talk about it in more detail in Section 3.4. After choice reconciliation, the result type is as follows.

```
G n n -> Int
```

(By adding a renaming step to our type pretty printer we could produce the more nicely looking type `G a a -> Int`.) The inferred choice type for `flop2` will be reconciled to the type.

```
R f -> f
```

1.2 Contributions and Structure of the Paper

In the remainder of this paper, we formalize this typing approach that we call “Chore”[3] in appreciation of the difficulty of GADT type inference. Overall, this paper makes the following contributions.

- We introduce choice types in Section 2 to precisely represent the types of GADT expressions. We achieve type inference precision without sacrificing the simplicity of the type language since choice types can be removed to obtain simpler types for communicating with programmers.
- We present the type system for GADTs in Section 3. Our type system separates the typing from the reconciliation process, which improves the use of type information during reconciliation. Our type system is conservative with respect to the traditional Hindley-Milner type system (Theorem 1), as well as sound (Theorem 2) and expressive (Theorem 3).
- We present a variational unification algorithm in Section 4. The algorithm is amenable to a simple implementation. Based on the unification algorithm, we present in Section 5 a type inference algorithm that is sound (Theorem 4) and principal before type reconciliation (Theorem 5).
- In Section 6, we discuss the impact of using choice types on rejecting ill-typed programs and error reporting. While it seems that the use of choice types complicates error reporting, this is actually not the case.
- In Section 7 we describe the results of an evaluation of several approaches, which shows that our approach accepts more well-typed programs and rejects more programs that will yield runtime errors.

[2] https://ocaml.org/

[3] An acronym for <u>cho</u>ice <u>re</u>conciliation.

In Section 8, we discuss the tradeoff between principality and precision of GADT type inference and present some empirical results showing that precision is more favored in practice. After discussing related work in Section 9, the paper concludes with Section 10.

2. Variational Types

The concept of choice types was introduced in [4, 5] to facilitate the type inference for program families. A choice has a name and contains two or more alternatives. For example, $D\langle\texttt{Int},\texttt{Bool}\rangle$ represents a choice between the two types `Int` and `Bool`. (The name D stands for “dimension” and reminds of the fact that each (non-nested) choice of a different name represents a variation point that is independent of other choices.) Types that contain choices are called *variational types*, and all other types are called *plain types*. We can extract plain types from a variational type ϕ with the help of a selection operation $\lfloor\phi\rfloor_{D.i}$ that takes a selector of the form $D.i$ and replaces each occurrence of choice D in ϕ with its ith alternative.

The definition of selection synchronizes choices with the same name; choices with different names are independent. Therefore, while $A\langle\texttt{Int},\texttt{Bool}\rangle \to A\langle\texttt{Bool},\texttt{Int}\rangle$ encodes two types, the type $A\langle\texttt{Int},\texttt{Bool}\rangle \to B\langle\texttt{Bool},\texttt{Int}\rangle$ encodes four types, where both the argument and return type may be either `Bool` or `Int`.

Variational types give rise to a notion of type equivalence, that is, different syntactic types may represent the same mapping of selectors to plain types. In general, two different types ϕ_1 and ϕ_2 can be equivalent (written as $\phi_1 \equiv \phi_2$) for three reasons. First, type constructors distribute over choices. For example, we have $A\langle\texttt{Int} \to \texttt{Bool},\texttt{Bool} \to \texttt{Int}\rangle \equiv A\langle\texttt{Int},\texttt{Bool}\rangle \to A\langle\texttt{Bool},\texttt{Int}\rangle$ because the arrow in the first type is lifted out of the choice A. Second, ϕ_2 is obtained by the elimination of one choice from ϕ_1. This happens when a choice is idempotent, that is, when all its alternatives are the same, or a choice is nested in another choice with the same name. For example, $A\langle A\langle\texttt{Int},\texttt{Bool}\rangle,\texttt{Int}\rangle \equiv A\langle\texttt{Int},\texttt{Int}\rangle \equiv \texttt{Int}$. The first relation holds because `Bool` in the first type is unreachable. When we select with $A.1$, the first alternative in both A choices is selected, which leads to `Int`, while selection with $A.2$ simply returns the second alternative of the outer A choice. Third, ϕ_2 is obtained by swapping nested choices of different names in ϕ_1. For example, $A\langle B\langle\texttt{Int},\texttt{Bool}\rangle,\texttt{Int}\rangle \equiv B\langle A\langle\texttt{Int},\texttt{Int}\rangle,A\langle\texttt{Bool},\texttt{Int}\rangle\rangle$. The type equivalence relation, which is presented in [5], is the reflexive, symmetric, and transitive closure of the union of these three relations.

3. Type System

This section presents the type system that assigns types to GADT programs. We show how choice types help to cast a type system that tracks type information flow precisely and thus turns many pattern matching failures into type errors. After introducing the syntax in Section 3.1, we present and discuss typing rules in Sections 3.2 through 3.4 and report some properties of the type system in Section 3.5.

3.1 Syntax

Figure 2 defines the syntax for types, expressions, and related environments. For simplicity, we assume that data types are predefined through data constructors, and we ignore nested patterns. We use bar notation for lists of objects, for example, $\overline{e}$ stands for expressions $e_1,\ldots,e_n$, where I is the set of subscripts $\{1,\ldots,n\}$ associated with all the objects.

The definitions of both expressions and types are conventional except for variational types. Here a choice type may contain any number of alternatives, while in [5] each choice contains only two alternatives. However, all the choices with the same name

Term variables	x, y, z	Type variables	α, β
Data Constructors	K	Type constructors	T
Dimensions	D		

Expressions	e, f	$::=$	$x \mid \lambda x.e \mid e\ e \mid K \mid \texttt{case}\ e\ \texttt{of}\ \{\overline{p \texttt{ -> } e}\}$
			$\texttt{let}\ x \texttt{ = } e\ \texttt{in}\ e \mid \texttt{let}\ x \texttt{::} \forall\overline{\alpha}.\tau \texttt{ = } e\ \texttt{in}\ e$
Patterns	p	$::=$	$K\ \overline{x}$

Monotypes	τ	$::=$	$\alpha \mid \tau \to \tau \mid T\ \overline{\tau}$
Variational types	ϕ	$::=$	$\tau \mid D\langle\overline{\phi}\rangle \mid \phi \to \phi \mid T\ \overline{\phi}$
Type schemas	σ	$::=$	$\phi \mid \forall\overline{\alpha}.\phi$

Type environments	Γ	$::=$	$\varnothing \mid \Gamma, x \mapsto \sigma$
Substitutions	θ	$::=$	$\varnothing \mid \theta, \alpha \mapsto \phi$

Figure 2: Syntax of expressions, types, and environments

must contain the same number of alternatives. For simplicity, we don't consider variational polymorphic types, because they can be converted into corresponding polymorphic variational types. The conversion process is detailed in [2] and will not be repeated here. While we don't include value and type constants in the syntax, we will use types like `Int` and `Bool` and values like `True` and numeric literals freely.

As usual, a type environment is a mapping from variables to type schemas, and a substitution is a mapping from type variables to variational types. We use the function $FV(\sigma)$ to collect the free type variables in σ. The definition is standard except for the choice type $D\langle\overline{\phi}\rangle$, where it is defined as the union of $\overline{FV(\phi)}$. The function $FV(\cdot)$ extends naturally to type environments and substitutions. We write $\theta(\sigma)$ to replace all occurrences of free variables in σ with their corresponding images in θ. Again, the definition is standard except for choice types, where it is defined as $\theta(D\langle\overline{\phi}\rangle) = D\langle\overline{\theta(\phi)}\rangle$.

3.2 Typing Overview

While type systems for traditional ADTs only allow scrutinee types to be more specific than the pattern types, GADT type systems only require pattern types and scrutinee types to be unifiable. Thus, scrutinee types may be more general than pattern types, which is also the case when local assumptions are introduced for typing pattern-match bodies. This accounts for the fundamental difficulty in typing GADT programs, because it is hard to decide what the appropriate local assumptions are and how to reconcile local assumptions among different pattern-matching branches. The over-generality of scrutinee types adds a loophole to type systems, causing them to accept programs such as `flop2 (RC 'a')` that will lead to runtime errors.

The driving factor behind the generalization of scrutinee types is the limitation that each expression can be assigned only one type. With choice types, we can remove this restriction since each choice type encodes a set of types, each of which may represent the type for a pattern-matching branch. Therefore, there is no need to generalize the scrutinee type for typing each branch. Unlike other GADT type systems that mix the typing and generalizing process, the type system in this paper separates them. During the typing process, we type GADT pattern-match branches as in the ADT case. However, we don't require that all branches have the same type and wrap different branch types into a choice type. We generalize typing results only when communicating with users. This separation brings the following benefits.

1. It simplifies the design of the type system since there is no need to introduce local assumptions for typing case branches.
2. It improves the reconciliation of demands from different branches on the same type. Our type system can reconcile types among different case branches.
3. It facilitates capturing more type errors. Since there is no need for local assumptions when typing branches, the branch types are canonical, reducing the possibilities of accepting programs that will result in runtime errors.

Figure 3 presents the typing rules that formalize this idea. The type system involves three judgments. First, $\Gamma \vdash e : \phi$ states that under the assumptions in Γ the expression e has the variational type ϕ. Second, $\Gamma \vdash_p p \texttt{ -> } e : \tau \to \phi$ expresses that the case branch $p \texttt{ -> } e$ has the type $\tau \to \phi$. Note that the pattern type is plain while the body type may be variational. Third, $\Gamma \vdash_m e : \tau$ states that the expression e has the plain type τ. This judgment is also the main interface to programmers. Unlike the first two judgments, it allows Γ to map variables to plain types only, although this is not formalized.

The rules VAR, CON, and ABS for typing variable references, data constructors, and abstractions are all standard. The rule LET for typing let expressions accounts for polymorphic recursion that is ubiquitous in GADT programs. In LET, we write $\overline{\alpha}\ \#\ FV(\Gamma)$ for a list $\overline{\alpha}$ that is disjoint from the free type variables in Γ. Since type inference with polymorphic recursion is undecidable, we also allow let expressions to be annotated with polymorphic plain types, which is handled by the rule LETA. We don't allow variational types to appear in type annotations because we want to control the generation and elimination of choice types. The idea of supporting two forms of let expressions is also employed in [21] and [27].

3.3 The Typing of Applications

Since the detection of most type errors happens in applications, the corresponding typing rule APP is doing most of the work and is consequently more involved. Note that we have to deal with three cases here. (1) The type of the argument (ϕ_2) matches the argument type (ϕ_1) exactly. In this situation, ϕ' will be the result type. (2) Some alternatives of (ϕ_2) match some alternatives of ϕ_1. In this situation, the result type is extracted from ϕ' by taking the alternatives where ϕ_2 and ϕ_1 match. We require all alternatives extracted from ϕ' to be the same. (3) No alternative of ϕ_2 matches any alternative of ϕ_1. In this case, the application is ill typed.

We handle these cases in a single APP rule with the help of the operations $\bowtie$ and $\ll$. The operation $\bowtie$ overlays two types and returns a pattern π that describes which parts of ϕ_2 and ϕ_1 agree. This pattern will then be used by $\ll$ to extract the part of ϕ' that will be returned as the result type of the application.

Here we reuse the machinery developed in [4] for manipulating patterns, defined as follows.

$$\pi ::= \bot \mid \top \mid D\langle\overline{\pi}\rangle$$

A pattern $\top$ says to keep the corresponding part, $\bot$ drops the corresponding part, and $D\langle\overline{\pi}\rangle$ recursively denotes whether to keep or drop each alternative in choice D.

The definition of the operation $\bowtie : \phi \times \phi \to \pi$ is given below. Note that $\bowtie$ is symmetric up to $\equiv$; the definition assumes that all idempotent choices have been eliminated by the rule $D\langle\phi,\phi\rangle \equiv \phi$.

$$\begin{aligned}
\phi \bowtie \phi &= \top \\
D\langle\overline{\phi}\rangle \bowtie \phi_r &= D\langle\overline{\phi \bowtie \lfloor\phi_r\rfloor_{D.i}}\rangle \\
\tau \bowtie D\langle\overline{\phi}\rangle &= D\langle\overline{\tau \bowtie \phi}\rangle \\
D\langle\overline{\phi}\rangle \to \phi_1 \bowtie \phi_r &= D\langle\overline{\phi \to \lfloor\phi_1\rfloor_{D.i}}\rangle \bowtie \phi_r \\
\tau \bowtie \tau' &= \bot \quad \text{where } \tau \neq \tau'
\end{aligned}$$

When two types are the same, the result is $\top$. For example, $D\langle\texttt{Int},\texttt{Bool}\rangle \bowtie D\langle\texttt{Int},\texttt{Bool}\rangle = \top$. Otherwise, if the first type is a

$$\boxed{\Gamma \vdash e : \phi} \qquad \boxed{\Gamma \vdash_p p \texttt{ -> } e : \tau \to \phi} \qquad \boxed{\Gamma \vdash_m e : \tau}$$

$$\textsc{Var}\ \frac{\Gamma(x) = \forall\overline{\alpha}.\phi_1 \qquad \phi = \{\overline{\alpha \mapsto \phi'}\}(\phi_1)}{\Gamma \vdash x : \phi} \qquad \textsc{Con}\ \frac{K : \forall\overline{\alpha}.\tau_1 \qquad \phi = \{\overline{\alpha \mapsto \phi'}\}(\tau_1)}{\Gamma \vdash K : \phi} \qquad \textsc{Abs}\ \frac{\Gamma, x \mapsto \phi \vdash e : \phi'}{\Gamma \vdash \lambda x.e : \phi \to \phi'}$$

$$\textsc{Let}\ \frac{\Gamma, x \mapsto \forall\overline{\alpha}.\phi_1 \vdash e : \phi_1 \qquad \overline{\alpha} \mathbin{\#} FV(\Gamma) \qquad \Gamma, x \mapsto \forall\overline{\alpha}.\phi_1 \vdash e' : \phi}{\Gamma \vdash \texttt{let } x \texttt{ = } e \texttt{ in } e' : \phi} \qquad \textsc{LetA}\ \frac{\Gamma, x \mapsto \forall\overline{\alpha}.\tau_1 \vdash e : \phi_1 \qquad \Gamma, x \mapsto \forall\overline{\alpha}.\tau_1 \vdash e' : \phi}{\Gamma \vdash \texttt{let } x \texttt{::} \forall\overline{\alpha}.\tau_1 \texttt{ = } e \texttt{ in } e' : \phi}$$

$$\textsc{App}\ \frac{\Gamma \vdash e_1 : \phi_1 \to \phi' \qquad \Gamma \vdash e_2 : \phi_2 \qquad \pi = \phi_1 \bowtie \phi_2 \qquad \phi = \pi \ll \phi'}{\Gamma \vdash e_1\ e_2 : \phi}$$

$$\textsc{Case}\ \frac{\overline{\Gamma \vdash_p p \texttt{ -> } e : \phi_a \to \phi_r} \qquad dom(\Gamma_i) =_{\forall i,j \in I} dom(\Gamma_j) \qquad D \text{ is fresh} \qquad D\langle\overline{\Gamma}\rangle \vdash e_1 : D\langle\overline{\phi_a}\rangle \qquad coherent(D\langle\overline{\phi_a}\rangle \to D\langle\overline{\phi_r}\rangle, D\langle\overline{\Gamma}\rangle)}{D\langle\overline{\Gamma}\rangle \vdash \texttt{case } e_1 \texttt{ of } \{\overline{p \texttt{ -> } e}\} : D\langle\overline{\phi_r}\rangle}$$

$$\textsc{Pat}\ \frac{K : \forall\overline{\alpha}.\overline{\tau_1} \to T\ \overline{\tau_2} \qquad \overline{\beta} = \overline{\alpha} \cap FV(\overline{\tau_1}) \qquad \theta = \{\overline{\beta \mapsto \tau}\} \qquad \overline{\tau_p} = \theta(\overline{\tau_2}) \qquad \overline{\alpha} \mathbin{\#} FV(\Gamma, \overline{\tau_p}, \phi) \qquad \Gamma \cup \theta\{\overline{x \mapsto \tau_1}\} \vdash e : \phi}{\Gamma \vdash_p K\ \overline{x} \texttt{ -> } e : T\ \overline{\tau_p} \to \phi}$$

$$\textsc{Main}\ \frac{\Gamma \vdash e : \phi \qquad (\tau, \Gamma_s) = reconcile(\phi, \Gamma)}{\Gamma_s \vdash_m e : \tau}$$

Figure 3: Rules for typing GADT programs

variational type, the operation is recursively applied to each alternative of the variational type. For example, $D\langle\texttt{Int},\texttt{Bool}\rangle \bowtie \texttt{Int} = D\langle\texttt{Int} \bowtie \texttt{Int}, \texttt{Bool} \bowtie \texttt{Int}\rangle = D\langle\top,\bot\rangle$. Note that in the recursive calls, we perform a corresponding selection in ϕ_r. This helps us deal with the situation that D is nested inside ϕ_r. For example, in the following case, D is replaced by its left and right alternative when the E choice is distributed into $D\langle\texttt{Int},\texttt{Bool}\rangle$.

$$\begin{aligned}
&D\langle\texttt{Int},\texttt{Bool}\rangle \bowtie E\langle D\langle\texttt{Int},\texttt{Bool}\rangle,\texttt{Int}\rangle \\
&= D\langle\texttt{Int} \bowtie E\langle\texttt{Int},\texttt{Int}\rangle, \texttt{Bool} \bowtie E\langle\texttt{Bool},\texttt{Int}\rangle\rangle \\
&= D\langle\top, E\langle\top,\bot\rangle\rangle.
\end{aligned}$$

A dual case is when the first type is plain and the second is variational, and we handle it similarly. If the left type is an arrow over variational types, we first push the arrow into choices and then delegate the call to the corresponding case. Again, we need to make selections to avoid building up choice nestings under the same choice name. Finally, if both types are plain but different, $\bowtie$ returns $\bot$.

While we reuse the definitions of π and $\bowtie$ from [4], the definition of the operation $\ll$ is very different. It has the type $\pi \times \phi \to \phi$ and is defined as follows.

$$\begin{aligned}
\top \ll \phi &= \phi \\
D\langle\overline{\pi}\rangle \ll \phi &= \lfloor\phi\rfloor_{D.i} && \text{if } \pi_i = \top \wedge \pi_j = \bot \text{ for all } j \neq i \\
D\langle\overline{\pi}\rangle \ll \phi &= \phi' && \text{if } \phi' = \pi_i \ll \lfloor\phi\rfloor_{D.i} \text{ for all } \pi_i \neq \bot
\end{aligned}$$

The definition for the first case is self-explanatory. The second case deals with the situation that there is only one $\top$ in the choice D, which leads to the selection of $D.i$ in ϕ. For example, $D\langle\top,\bot\rangle \ll D\langle\texttt{Int},\texttt{Bool}\rangle = \texttt{Int}$. Finally, if D contains more than one alternative that is not $\bot$, the rule requires that recursively applying $\ll$ to non-$\bot$ patterns and the corresponding types $\lfloor\phi\rfloor_{D.i}$ yields the same result. Again, all other cases denote a type error.

To see the App rule in action, consider typing the following two programs.

```
h1 = flop2 (RI 1)
h2 = flop2 (RC 'a')
```

For `h1`, we obtain the following judgments (we omit Γ for brevity).

$$\begin{aligned}
&\texttt{flop2 : } D\langle\texttt{R Int},\texttt{R Bool}\rangle \to D\langle\texttt{Int},\texttt{Bool}\rangle \\
&\texttt{ RI 1 : R Int} \\
&\quad \pi = D\langle\texttt{R Int},\texttt{R Bool}\rangle \bowtie \texttt{R Int} = D\langle\top,\bot\rangle \\
&\quad \phi = D\langle\top,\bot\rangle \ll D\langle\texttt{Int},\texttt{Bool}\rangle = \texttt{Int} \\
&\quad \texttt{h1 : Int}
\end{aligned}$$

For `h2`, we reason similarly as follows.

$$\begin{aligned}
&\texttt{flop2 : } D\langle\texttt{R Int},\texttt{R Bool}\rangle \to D\langle\texttt{Int},\texttt{Bool}\rangle \\
&\texttt{RC 'a' : R Char} \\
&\quad \pi = D\langle\texttt{R Int},\texttt{R Bool}\rangle \bowtie \texttt{R Char} = D\langle\bot,\bot\rangle \\
&\quad \phi = D\langle\bot,\bot\rangle \ll D\langle\texttt{Int},\texttt{Bool}\rangle = \text{undefined} \\
&\quad \texttt{h2 : } \text{undefined}
\end{aligned}$$

We observe that `h1` is assigned the type `Int` as expected. As for `h2`, the operation $\ll$ fails because no rule applies. Thus, we can successfully catch the type error. Previous type systems accept `h2` even though its evaluation leads to a runtime error.

3.4 Reconciling Local Assumptions

To type a case expression with the rule Case, we first type each of its branches under potentially different environments provided that they have the same domain. Note that branches may have different types. Each case expression is assigned a fresh choice in such a way that different case expressions don't interfere with each other. We then pack the environments for typing branches into a single environment using a choice to type the case scrutinee. We require the case scrutinee to have the same type as the type obtained by packing pattern types with the fresh choice assigned for the case expression. The application of a choice of environments yields the choice of types from the individual environments.

$$D\langle\overline{\Gamma}\rangle(\alpha) = D\langle\overline{\Gamma(\alpha)}\rangle$$

We will use this construction of a choice of functions again later in the unification algorithm. We require, as is made explicit in the second premise of rule Case, that all type environments have the same domain. This can be satisfied through a process known as weakening [24].

In addition, we need a coherence check to decide whether a case expression is well typed. This condition ensures that different branch body types can be reconciled by introducing appropriate local assumptions. For example, the function `flop2` can be assigned the type $\forall\alpha.\texttt{R}\ \alpha \to \alpha$ because in both branches the body has the same type α where the assumption maps α to `Int` or `Bool`, respectively. However, in some cases the branch bodies can't be reconciled through local assumptions and thus should be rejected by the type system. For example, consider the function `cross` [17] below,

```
cross (RI x) = even x
cross (RB x) = 1
```

Although both branches are well typed with return types `Bool` and `Int`, respectively, `cross` is ill typed since we can't assign a meaningful type to it. We can't generalize `Bool` to α with the local assumption that α maps to `Int` for the first branch.

Our type system rejects `cross` because it will not pass the coherence check, which is formally defined as follows.

$$\mathit{coherent}(\phi,\Gamma) \text{ iff } \exists(\tau,\Gamma')\colon (\tau,\Gamma') = \mathit{reconcile}(\phi,\Gamma)$$

The function $\mathit{reconcile}(\cdot)$ will be defined later when we discuss the rule Main.

The rule Pat for typing case branches is the same as for typing ADT case branches. As mentioned earlier, we don't introduce local assumptions when typing branches since that makes the scrutinee type too general, losing the benefits of catching more type errors. The only subtlety of the rule is to avoid the instantiation and escape of existentially quantified type variables. This is why we compute $\overline{\beta}$ in the rule.

We use the rule Main to communicate with users by using only the traditional type syntax. To realize this goal, we need to get rid of choices in the typing result. The overall idea is to systematically replace choices by type variables. However, such type variables must at least appear inside of GADT type constructors, which in turn must be used as function arguments. This is because only GADTs can introduce type refinements. For example, $D\langle\texttt{R Int},\texttt{R Bool}\rangle \to D\langle\texttt{Int},\texttt{Bool}\rangle$ can be reconciled to $\texttt{R}\ \alpha \to \alpha$, but not to $\alpha \to \beta$ or $\texttt{R}\ \alpha \to \beta$. The type $\alpha \to \beta$ doesn't allow any type refinement because neither of α and β appears inside any GADT type constructor. The type $\texttt{R}\ \alpha \to \beta$ only allows to refine α since β doesn't appear inside GADT type constructor. We call the variables that allow type refinements *type index variables*.

For a given type τ, the function $\mathit{FI}(\tau)$ collects the free type index variables in the argument types of τ. The auxiliary function L strips off the return type of the last top-level arrow in the type. For simplicity, we assume that all type constructors support GADT type refinements.

$$\begin{aligned} \mathit{FI}(\tau_l \to \tau_r) &= \mathit{FI}'(L(\tau_l \to \tau_r)) \\ \mathit{FI}(\tau) &= \varnothing \\ \mathit{FI}'(\alpha) &= \varnothing \\ \mathit{FI}'(\tau_l \to \tau_r) &= \mathit{FI}'(\tau_l) \cup \mathit{FI}'(\tau_r) \\ \mathit{FI}'(T\ \overline{\tau}) &= \mathit{FV}(T\ \overline{\tau}) \end{aligned} \qquad \begin{aligned} L(\tau \to \alpha) &= \tau \\ L(\tau \to T\ \overline{\tau}) &= \tau \\ L(\tau_l \to \tau_r) &= \tau_l \to L(\tau_r) \end{aligned}$$

With the help of FI, we have the following inference rule to decide whether (ϕ,Γ) can be reconciled to (τ,Γ_s).

$$\textsc{Reconcile}\quad \frac{\mathit{dom}(\theta) \subseteq \mathit{FI}(\phi) \cup \mathit{FI}'(\Gamma) \qquad \phi \equiv \theta(\tau) \qquad \Gamma = \theta(\Gamma_s)}{(\tau,\Gamma_s) = \mathit{reconcile}(\phi,\Gamma)}$$

With the rules in Figure 3, we can derive the following typing judgment for `flop2`.

$$\varnothing \vdash \texttt{flop2} : D\langle\texttt{R Int},\texttt{R Bool}\rangle \to D\langle\texttt{Int},\texttt{Bool}\rangle.$$

By choosing $\theta = \{\alpha \mapsto D\langle\texttt{Int},\texttt{Bool}\rangle\}$ in rule Reconcile, we derive the following type to be communicated to the user.

$$\varnothing \vdash_m \texttt{flop2} : \texttt{R}\ \alpha \to \alpha.$$

Our type system rejects the function `cross` because the branch types $D\langle\texttt{R Int},\texttt{R Bool}\rangle \to D\langle\texttt{Bool},\texttt{Int}\rangle$ can't be reconciled to any type. For example, both $\texttt{R}\ \alpha \to \alpha$ and $\texttt{R}\ \alpha \to \beta$ are not options since no θ exists such that they can be unified with the branch types. Note that $\mathit{dom}(\theta) = \{\alpha\}$ for both candidate types.

3.5 Properties

This section investigates the properties of the type system. First, our type system is a conservative extension of the Hindley-Milner type system. We express this fact in the following theorem, where $\Gamma \vdash^{HM} e : \tau$ states that the expression e has the type τ under the assumptions in Γ. Of course, Γ binds variables to plain types only.

Theorem 1 (Conservative extension of HM). *If e refers only to standard data constructors of type $\forall\overline{\alpha}\overline{\beta}.\overline{\tau} \to T\ \overline{\alpha}$, then $\Gamma \vdash^{HM} e : \tau \Rightarrow \Gamma \vdash_m e : \tau$ and $\Gamma \vdash_m e : \tau \Rightarrow \Gamma \vdash^{HM} e : \tau$.*

Proof The formal proof proceeds by an induction over the typing rules. We present here only an informal argument. First, we observe that the rule Pat for typing pattern-matching branches is exactly the same as the rule in HM. Next, the rule Case also collapses to the rule in HM for typing case expressions since $\mathit{coherent}(\cdot)$ requires all the branches to have the same type as there are no type index variables due to the absence of GADT type constructors. The rule App for applications will also be simplified to the traditional rule as the first case of $\bowtie$ and $\ll$ will be chosen for applications to be well typed. The proof for the other rules is obvious. □

Next, we show the soundness of our type system by relating it to the type system $\mathscr{P}$ presented in [17], which consists of standard typing rules for GADT programs and has been proved to be sound. We write $\Gamma \vdash^{\mathscr{P}} e : \tau$ if the expression e has the type τ in $\mathscr{P}$.

Theorem 2 (Type system soundness). *If $\Gamma \vdash_m e : \tau$, then $\Gamma \vdash^{\mathscr{P}} e : \tau$.*

Proof As usual, we can establish the proof based on induction over the typing rules. The proof is obvious except for the rule Case since all other rules are standard in both systems and since our App rule is more restrictive than the one in $\mathscr{P}$. For the rule Case we rely on Lemma 1, whose conclusions can be directly used as premises for the typing rule for case expressions in $\mathscr{P}$. □

The following lemma states that there is a close relationship between the typing for case expressions and that for case branches, reflected in the choice introduced for typing that case expression. In the following lemma we write $\lfloor\Gamma\rfloor_{D.i}$ and $\lfloor\theta\rfloor_{D.i}$ for selecting from the range of Γ and θ, respectively, with the selector $D.i$.

Lemma 1 (Case branch typing equivalence).

- *If $\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \{\overline{p\ \texttt{->}\ e}\} : D\langle\overline{\phi_r}\rangle$ and $\Gamma \vdash e : D\langle\overline{\phi_p}\rangle$, then $\lfloor\Gamma\rfloor_{D.i} \vdash p_i\ \texttt{->}\ e_i : \phi_{ri} \to \phi_{pi}$.*
- *If $\Gamma_s \vdash_m \texttt{case}\ e\ \texttt{of}\ \{\overline{p\ \texttt{->}\ e}\} : \tau_r$ and $\Gamma_s \vdash_m e : \tau_p$ with the reconciling substitution θ, then $\Gamma_s \vdash_m p_i\ \texttt{->}\ e_i : \tau_p \to \tau_r$ with the reconciling substitution $\lfloor\theta\rfloor_{D.i}$.*

The proof of this lemma is again an induction over the typing rules and the choice structure.

Finally, we present a result about the expressiveness of the type system, again by relating it to system $\mathscr{P}$. This property shows that our type system detects more type errors without sacrificing expressiveness.

Theorem 3 (Type system expressiveness). *If $\Gamma \vdash^{\mathscr{P}} e : \tau$, then $\Gamma \vdash_m e : \tau$, provided that $\Gamma' \vdash e_1\ e_2 : \phi'$ for each subexpression $e_1\ e_2$ in e and the corresponding environment Γ'.*

The side condition in the last theorem states that if the typing of $\Gamma \vdash_m e : \tau$ requires the typing of the subexpression $e_1\ e_2$ under the environment Γ', then there exists some ϕ' such that $\Gamma' \vdash e_1\ e_2 : \phi'$ holds. Informally, this means that the typing of each application doesn't fail. If this condition holds, then $\Gamma \vdash^{\mathscr{P}} e : \tau$ implies $\Gamma \vdash_m e : \tau$. When the side condition doesn't hold, it means that although $\mathscr{P}$ accepts the expression e, evaluating it will lead to a runtime pattern matching failure. One such example is the expression `h2`.

Again, we can prove the theorem by an induction over the typing rules with the help of Lemma 1.

4. Unification

The most important component of type inference is the unification algorithm. This is especially true for GADT type inference. In our approach, unification is complicated by the fact that our algorithm allows types to be partially matched for the programs to be well-typed. This is in contrast to traditional type inference where types are required to match exactly. Working with partial matches raises an intriguing question: Shall we adopt an exact match whenever possible, or might pursuing partial matching be beneficial? To illustrate this issue, consider the following program.

```
h3 x y = (case x of {RI _ -> RI; RB _ -> RB}) y
```

The inference algorithm assigns the following type to the case expression of `h3`.

$$D\langle \texttt{Int}, \texttt{Bool}\rangle \to D\langle \texttt{R Int}, \texttt{R Bool}\rangle$$

To compute the type of the body it has to solve the following unification problem.

$$D\langle \texttt{Int} \to \texttt{R Int}, \texttt{Bool} \to \texttt{R Bool}\rangle \equiv^? \alpha \to \beta$$

Here we use $\equiv^?$ to denote a unification problem modulo the equivalence relationship on variational types. Moreover, α represents the type for the variable `y`, and β represents the result type of the application.

For this unification problem, the following are all sensible unifiers.

1. $\theta_1 = \{\alpha \mapsto \texttt{Int}, \beta \mapsto \texttt{R Int}\}$. In this case, the result type is `R Int`, the type of `h3` is $D\langle \texttt{R Int}, \texttt{R Bool}\rangle \to \texttt{Int} \to \texttt{R Int}$, and the reconciled result type is $\texttt{R}\ \alpha \to \texttt{Int} \to \texttt{R Int}$.
2. $\theta_2 = \{\alpha \mapsto \texttt{Bool}, \beta \mapsto \texttt{R Bool}\}$. In this case, the result type is `R Bool`, the type of `h3` is $D\langle \texttt{R Int}, \texttt{R Bool}\rangle \to \texttt{Bool} \to \texttt{R Bool}$, and the reconciled result type is $\texttt{R}\ \alpha \to \texttt{Bool} \to \texttt{R Bool}$.
3. $\theta_3 = \{\alpha \mapsto D\langle \texttt{Int}, \texttt{Bool}\rangle, \beta \mapsto D\langle \texttt{R Int}, \texttt{R Bool}\rangle\}$. In this case, the result type is $D\langle \texttt{R Int}, \texttt{R Bool}\rangle$, the type of `h3` is $D\langle \texttt{R Int}, \texttt{R Bool}\rangle \to D\langle \texttt{Int}, \texttt{Bool}\rangle \to D\langle \texttt{R Int}, \texttt{R Bool}\rangle$, and the reconciled result type is $\texttt{R}\ \alpha \to \alpha \to \texttt{R}\ \alpha$.

Although all the reconciled types are acceptable, the last one is arguably best since it subsumes the other two. This example demonstrates that when unifying types, we should search for a solution that makes the two types match exactly. The patterns that are produced during the unification process, $D\langle \top, \bot\rangle$, $D\langle \bot, \top\rangle$, and $\top$, substantiate this argument since the last pattern is better than the other two because it doesn't contain any mismatch.

This example suggests that the general strategy for solving unification problems should be to look for solutions that unify types completely. If that is impossible, we should try to unify as many parts as possible. At the same time, we should search for most general unifiers. The unification algorithm $\mathscr{U}$ shown in Figure 4 achieves both of these goals. It computes the same results as the unification algorithm presented by Chen et al. [4], but it is significantly simpler. In particular, we do not need three separate phases of qualification, qualified unification, and completion. This simplification is possible because choices support localized computations. Essentially, the computation in one alternative doesn't affect that of the other. Thus, we should perform computations locally rather than globally whenever possible, beause it likely causes fewer conflicts. Also, the implementation is now just 87 lines of Haskell code, compared to 345 lines for the old one. For lack of space, we will not explain the rules in detail and refer to [4] for a discussion of unification with choice types.

$$
\begin{array}{l}
\mathscr{U} : \phi \times \phi \to \theta \times \pi \\
\text{(a) } \mathscr{U}(D\langle\overline{\phi_l}\rangle, D\langle\overline{\phi_r}\rangle) = (D\langle\overline{\theta}\rangle, D\langle\overline{\pi}\rangle) \\
\qquad\qquad\qquad\qquad \text{where } \overline{(\theta, \pi)} = \overline{\mathscr{U}(\phi_l, \phi_r)} \\
\qquad\qquad\qquad\qquad\quad \text{and } dom(\theta_i) =_{\forall i, j \in I} dom(\theta_j) \\
\text{(b) } \mathscr{U}^\star(D\langle\overline{\phi}\rangle, \phi_r) = \mathscr{U}(D\langle\overline{\phi}\rangle, D\langle\overline{\lfloor \phi_r \rfloor_{D.i}}\rangle) \\
\text{(c) } \mathscr{U}^\star(\alpha, \phi_1 \to \phi_2) \\
\qquad \mid \alpha \notin FV(\phi_1 \to \phi_2) \quad = (\{\alpha \mapsto \phi_1 \to \phi_2\}, \top) \\
\qquad \mid D \in dims(\phi_l \to \phi_r) = \mathscr{U}(D\langle\overline{\alpha}\rangle, \phi_l \to \phi_r) \\
\qquad \mid \text{otherwise} \qquad\qquad\; = (\varnothing, \bot) \\
\text{(d) } \mathscr{U}(\phi_1 \to \phi_2, \phi_3 \to \phi_4) = (\theta_2 \circ \theta_1, \pi_1 \otimes \pi_2) \\
\qquad\qquad\qquad\qquad \text{where } (\theta_1, \pi_1) = \mathscr{U}(\phi_1, \phi_3) \\
\qquad\qquad\qquad\qquad\qquad\quad (\theta_2, \pi_2) = \mathscr{U}(\theta_1(\phi_2), \theta_1(\phi_4)) \\
\text{(e) } \mathscr{U}(\tau_l, \tau_r) \mid robinson(\tau_l, \tau_r) = \theta = (\theta, \top) \\
\qquad\qquad\quad\;\; \mid \text{otherwise} \qquad\qquad\quad\; = (\varnothing, \bot)
\end{array}
$$

Figure 4: A unification algorithm

```
$infer : \Gamma \times e \to \theta \times \phi \times 2^\alpha$
$infer(\Gamma, e_1\ e_2) =$
   $(\theta_1, \phi_1, i_1) \leftarrow infer(\Gamma, e_1)$
   $(\theta_2, \phi_2, i_2) \leftarrow infer(\theta_1(\Gamma), e_2)$
   $(\theta, \pi) \leftarrow \mathscr{U}(\theta_2(\phi_1), \theta_2(\phi_2) \to \alpha)$     where $\alpha$ is fresh
   return $(\theta \circ \theta_2 \circ \theta_1, \pi \ll \theta(\alpha), i_1 \cup i_2)$
$infer(\Gamma, \texttt{case}\ e_s\ \texttt{of}\ \{\overline{p\ \texttt{->}\ e}\}) =$
   $(\theta_s, \phi_s, i_s) \leftarrow infer(\Gamma, e_s)$
   $(\tau_s, \overline{\theta}, \overline{\tau_p}, \overline{\phi_r}, \overline{i}) \leftarrow inferAlts(\theta_s(\Gamma), \overline{p\ \texttt{->}\ e})$
   $\theta_p = \mathscr{U}'(\phi_s, D\langle\overline{\tau_p}\rangle)$                 where $D$ is fresh
   $\theta_u \leftarrow D\langle\overline{\theta \circ \theta_p \circ \theta_s}\rangle$
   $\chi \leftarrow FV(\tau_s) \cap \bigcup \overline{dom(\theta)}$
   for each $D\langle\overline{\phi}\rangle$ in $atomic(\theta_p(D\langle\overline{\tau_p}\rangle), \theta_p(D\langle\overline{\phi_r}\rangle), \theta_u(FV(\Gamma)))$
      if $\forall \alpha \in \chi : \nexists \theta' : \mathscr{U}'(D\langle\overline{\phi}\rangle, \theta_u(\alpha)) = \theta'$
         fail
   return $(\theta_u, \theta_p(D\langle\overline{\phi_r}\rangle), i_s \cup \bigcup \overline{i} \cup \chi)$
```

Figure 5: A type inference algorithm

5. Type Inference

The addition of choice types has provided two distinctive benefits for our type system. First, the type system can capture more type errors in applications. Second, by separating typing from reconciliation, the latter can be improved. However, these two features also pose some challenges for the inference algorithm. For example, to infer types for applications, we need to find a function argument type and a type of the argument that match as much as possible. Moreover, the rule RECONCILE only checks whether we can reconcile a variational type to some given plain type τ, but τ has to be computed during type inference.

Our type inference algorithm employs Mycroft's extension [19] to algorithm $\mathscr{W}$ to deal with type inference for polymorphic recursion. The overall idea is to type a recursive function in iterations and stop when the result type between two consecutive iterations stays the same or the number of iterations has passed a threshold.

We present part of the inference algorithm in Figure 5. Since the algorithm involves lots of details, we only present its skeleton. The inference algorithm takes two arguments, a type environment and an expression, and returns the resulting substitution, the result type, and the set of index variables that we can use to refine the result types and substitutions. Since during type inference we can't rely on *FI* to compute the set of type index variables, we need to build those sets along with the inference process.

We show the two cases for which our algorithm deviates the most from traditional inference algorithms. The first case infers the type of applications. Interestingly, the algorithm $\mathscr{W}$ requires only a small adjustment. This is because the bulk of the work is performed by the unification algorithm that handles the combinations of variational types. We call $\mathscr{U}$ to unify the type of the function and the type of the argument. The other component π returned by $\mathscr{U}$ is used to extract corresponding parts from the return type.

The second case infers the type for case expressions. The algorithm employs the helper function $atomic(\phi)$, which moves choices into type constructors to make choices as small as possible. It returns all those choices that are not equivalent to plain types. For example, $atomic(D\langle \texttt{R Int} \to \texttt{Bool} \to \texttt{Int}, \texttt{R Bool} \to \texttt{Int} \to \texttt{Int}\rangle)$ returns $\{D\langle \texttt{Int}, \texttt{Bool}\rangle, D\langle \texttt{Bool}, \texttt{Int}\rangle\}$. This case also employs the following shorthand for calling the unification algorithm.

$$\mathscr{U}'(\phi_l, \phi_r) = \theta \quad \text{if } \mathscr{U}(\phi_l, \phi_r) = (\theta, \pi) \wedge \pi = \top$$

Note that $\mathscr{U}'$ fails if the condition is not fulfilled.

The algorithm first infers the type for the scrutinee e_s. It then calls *inferAlts* to compute the type information for case branches. The following information is returned. (1) The scrutinee type τ_s that best describes the common structures of the pattern types $\overline{\tau_p}$. We can also view τ_s as the template for $\overline{\tau_p}$. (2) The list of substitutions obtained by type inference for each branch. (3) The pattern types $\overline{\tau_p}$ and the branch body types $\overline{\phi_r}$. (4) The list of index type variable $\bar{i}$. We omit the definition of *inferAlts* since it is not very interesting.

Next, *infer* wraps the pattern types in a fresh choice, unifies it with the scrutinee types ϕ_s, and updates substitutions correspondingly. After that, *infer* computes the index type variables of the case expression as the intersection of free type variables of τ_s and the union of domains of branch substitutions, meaning that only variables in pattern templates that map to something more specific can be used as indices.

Finally, *infer* checks if the variational types $D\langle\overline{\tau_p}\rangle$, $D\langle\overline{\phi_r}\rangle$, and those in θ_u are coherent. It achieves this by checking if each atomic choice is unifiable with some type index variable. If this fails for any of the atomic choice types, type inference fails.

We omit the definition of the global reconciliation, which is essentially the same process as used in *infer* for the case expressions. The reconciliation replaces atomic choices with corresponding type index variables and updates the result type when a substitution exists. (In contrast, in case expressions it is only determined whether such a substitution exists.) The reconciliation also uses the following criteria. First, if an atomic choice is equivalent to a plain type or contains just one alternative, then the atomic choice is replaced by that plain type or the single alternative in the result type. For example, the inferred type for `flop1` is $\texttt{R}\,A\langle\texttt{Int}\rangle \to A\langle\texttt{Int}\rangle$ and the reconciliation result is $\texttt{R Int} \to \texttt{Int}$. Second, if more than one type index variable can be used and their corresponding substitutions are equal up to variable renaming, then these type index variables collapse into one variable. We collapse two type index variables α_1 and α_2 into a new one β by just renaming α_1 and α_2 to β. This happens in `param1o` where two type index variables can be used to substitute $A\langle\texttt{Int}, \texttt{Bool}\rangle$ and they collapse to an index variable, pretty-printed as `n`, which leads to the type `G n n -> Int` for `param1o`. Third, if more than one type index variable can be used to substitute an atomic choice, and if their corresponding substitutions are not equal, then reconciliation fails.

We write $inferMain(\Gamma, e) = (\theta, \tau)$ to express that reconciliation of the inference result $infer(\Gamma, e)$ yields (θ, τ).

Our type inference algorithm enjoys the following useful properties.

THEOREM 4 (Inference soundness).
If $infer(\Gamma, e) = (\theta, \phi, \chi)$*, then* $\theta(\Gamma) \vdash e : \phi$*. If* $inferMain(\Gamma, e) = (\theta, \tau)$ *and* Γ *contains plain types only, then* $\theta(\Gamma) \vdash_m e : \tau$*.*

THEOREM 5 (Inference principality).
If $\theta_1(\Gamma) \vdash e : \phi$ *and* $infer(\Gamma, e) = (\theta_2, \phi', \chi)$*, then* $\theta_1 = \theta_3 \circ \theta_2$ *and* $\phi = \theta_4(\phi')$ *for some substitutions* θ_3 *and* θ_4*.*

Theorem 4 states that our inference algorithms are sound and only compute correct results. Theorem 5 says that if *infer* terminates and computes a result, then it is principal. Principality comes at the price of having choice types in the type language. If we want to get rid of choice types, we lose principality during reconciliation that converts variational types to plain types. Unsurprisingly, the inference algorithms are incomplete mainly due to polymorphic recursion. Section 7 provides many examples showing that our inference algorithms fail to infer correct types.

6. Beyond Principal Type Inference

The introduction of choice types allows Chore to not only restore principality of type inference but also reject more programs that lead to runtime errors and deliver more informative error messages for ill-typed GADT programs.

6.1 Rejecting More Programs Yielding Runtime Errors

An important goal for introducing GADTs is to encode invariants over programs and data structures in types and have them enforced statically. However, this goal is partially compromised when we allow some errors to escape to runtime even though they may be caught at compile time through a better use of type information. We illustrate this aspect with the expression `gadt7q` presented in Figure 6a. This expression is an extended version of the following expression `gadt7o` that was first introduced by Lin [17].

```
data Z
data S n

data L a b where
   Nil :: L Z a
   Cons :: a -> L n a -> L (S n) a

gadt7o e = (case e of {Nil -> True},
            case e of {Cons x xs -> False})
```

The expression `gadt7o` should be considered ill typed, because when run, it will always cause a runtime pattern matching error. While Ambivalent and $\mathscr{P}$, but not GHC, reject this expression, only Chore is able to detect the error statically in `gadt7q`, which should be rejected on the same grounds since, no matter what `e` is, one component of the triple will always fail to match.

To make GHC work, we had to annotate `gadt7q`. Otherwise, GHC would produce a similar message as in Figure 1b for `param1o`. As shown in Figure 6b, GHC accepts `gadt7q` without complaint.

The output of Ambivalent is shown in Figure 6c. It first warns about the non-exhaustiveness of pattern-matching in the first case expression. (For brevity, we have omitted similar warnings for the next two case expressions.) Although useful, the warnings are not

```
gadt7q :: R a -> (a,a,a)
gadt7q e = (
    case e of
      RI i -> i; RB b -> b,
    case e of
      RB b -> b; RC c -> c,
    case e of
      RC c -> c; RI i -> i)
```

(a) GADT expression gadt7q

```
gadt7q :: R a -> (a, a, a)
```

(b) Output of GHC

```
File "gadt7q.ml", line 9, characters 8-54:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
RC _
... lines 5 through 12 omitted

val qadt7q : 'a r -> 'a * 'a * 'a = <fun>
```

(c) Output of Ambivalent

```
gadt7q :: forall a. R a ->
                    ((a, a), a)
```

(d) Output of $\mathscr{P}$

```
The expression gadt7q is ill typed
because the types of the
expressions
    case e of
      RI i -> i; RB b -> b
    ... lines 5 through 8 omitted
do not overlap
```

(e) Output of Chore

Figure 6: A GADT expression that will always cause a runtime error and that is only rejected by Chore.

directly related to the detection of the runtime failure. Ambivalent finally assigns a type to `gadt7q`, while it shouldn't.

The output from algorithm $\mathscr{P}$ is presented in Figure 6d. Since $\mathscr{P}$ doesn't support triples, we have to encode `gadt7q` with a nesting of tuples, which accounts for a slightly different type inferred by $\mathscr{P}$. The idea of $\mathscr{P}$ is to infer the types of branches and then reconcile them with type refinements introduced through pattern matching. However, this idea is limited to the level of each case expression. This makes $\mathscr{P}$ infer the type `R a -> a` for each case expression, losing the opportunity to detect the inconsistencies between different tuple components.

Finally, Figure 6e presents the output of Chore, which rejects `gadt7q` since the three case expressions will not match any common expression. Chore achieves this by inferring that the three case expressions require `e` to have types $A\langle$`R Int`,`R Bool`$\rangle$, $B\langle$`R Bool`,`R Char`$\rangle$, and $C\langle$`R Char`,`R Int`$\rangle$, respectively. When they are unified, Chore concludes that they have nothing in common and that no type can be assigned to `e`, and `gadt7q` is therefore deemed ill typed and rejected.

6.2 Generating More Informative Error Messages

The debugging of type errors in GADT programs is complicated by type refinements. How do choice types affect type-error reporting in GADT programs? To illustrate the contribution of choice types, consider the expression `cross` introduced in Section 3.4. It is ill typed because the types of the case branches `Bool` and `Int` can't be reconciled with type refinements.

When the expression is not annotated, GHC produces a message of 32 lines long. The content is similar to the one in Figure 1b. This message is hard to understand for programmers since it explains the problem using compiler jargon. What is confusing is that the first four lines in Figure 1b involve three type variables, `t`, `t1`, and `t2`, which are not directly related. Moreover, the message complains about a small part of the source code and fails to reveal the problem on a higher level.

When we annotate `cross` with the type `R a -> a`, GHC produces the following message.

```
Couldn't match type Int with Bool
Expected type: a
  Actual type: Bool
In the expression: even y
... lines 5 through 14 omitted
```

Although this message is an improvement over the one in Figure 1b, it is still doesn't explain the overall problem.

Ambivalent produces a message similar to the one in Figure 1c. This message is more concise, but also doesn't reveal the problem. The algorithm $\mathscr{P}$ displays the following message.

```
Type inference failed for cross
ERROR: Cannot reconcile branch body types
```

Chore produces the following message.

```
The expression cross is ill typed
because the types of the bodies vary inconsistently
with the types of the patterns in the expression
    case x of
      RI y -> even y
      RB y -> 1
The types
    R Int -> Bool
    R Bool -> Int
of the case branches can't be reconciled
```

We observe that both $\mathscr{P}$ and Chore reveal the fundamental problem of `cross` and convey it at a higher level. Compared to $\mathscr{P}$, however, Chore shows more detailed information. It presents the computed types of relevant branches.

While for this simple example, $\mathscr{P}$ may be engineered to produce a message similar to Chore's, our approach can keep highly granular type information around as long as needed, which supports the generation of better error messages in general. In particular, this allows us to apply reconciliation only just before types are reported to users, which means that types of case branches involved in type errors are always available for displaying. This is not the case for $\mathscr{P}$, which reconciles types for each case expression and discards information about case branches immediately. This makes it difficult for $\mathscr{P}$ to produce informative descriptions about type errors that involve multiple case expressions.

In summary, while intuitively the use of choice types may seem to complicate the communication of type errors, they are actually quite useful for producing more informative error messages.

7. Evaluation

Implementing GADT type inference algorithms is a challenging undertaking. Such an implementation has to account for undecidability and the difficulty of reconciling competing demands among case branches. Undecidability makes implementations incomplete, and reconciliations causes them to lose principality. This makes it difficult to compare the capabilities of different type inference algorithms theoretically. For this reason, we have evaluated a prototype implementing our type inference algorithm experimentally.

Figure 7 presents the evaluation results of four different approaches over a set of programs that were chosen to cover the space of GADT typing possibilities and illustrate the strengths and weaknesses of the different approaches. We compare our approach, Chore, with the algorithm $\mathscr{P}$, the OutsideIn approach implemented in GHC 7.8,[4] and the Ambivalent approach implemented in OCaml 4.01.0. The top 11 test programs are taken from [17], the bottom five are defined in this paper. The programs `gadt7p` and `gadt7q` are variations of `gadt7o`, to be explained later. Programs whose names

[4] https://www.haskell.org/platform/

	Chore		$\mathscr{P}$	OutsideIn		Ambivalent	
		+ann			+ann		+ann
`equ1`	◐	●	◐	○	●	◐	●
`refine`	●	●	●	○	●	○	●
`param1o`	●	●	○	○	●	○	●
`head`	●	●	●	○	●	●	●
`eval4`	●	●	●	○	●	○	●
`gadt7o`⚠	●	●	●	○	○	●	●
`delmin_o`	●	●	○	○	●	○	●
`rotl`	●	●	●	○	●	○	●
`fcComp1`	●	●	○	○	●	○	●
`leq_o`	●	●	○	○	●	○	●
`runState_o`	○	●	○	○	●	○	●
`gadt7p`	●	●	◐	○	●	○	●
`gadt7q`⚠	●	●	○	○	○	○	○
`h1`	●	●	●	○	●	○	●
`h2`⚠	●	●	○	○	○	○	○
`h3`	●	●	●	○	●	○	●
`u1`⚠	●	●	●	○	●	○	●
`u2`	●	●	●	○	○	○	○

Type checker is: ●=correct, ○=incorrect, ◐=partially correct

Figure 7: Evaluation results for different approaches for a set of programs. The "+ann" columns show the results when GADT functions are given correct explicit type annotations. Note that $\mathscr{P}$ does not support type annotations. The circle fillings are chosen so that the darker a column, the better the corresponding approach.

have a warning sign (⚠) attached are ill typed and will cause a runtime error when evaluated. For GHC and OCaml, we also include the results when annotations are added to function definitions. Correctness is evaluated as follows. A type checker is expected to infer a type for a type-correct program and report a type error otherwise. An approach receives a ● if it does that, otherwise it gets a ○. In some cases, a systems may infer a type that is considered only partially accurate. In this case the approach gets a ◐.

We first look at results for programs that are well typed. When annotations are absent, we observe that Chore and $\mathscr{P}$ perform significantly better than OutsideIn and Ambivalent, with Chore being better than $\mathscr{P}$. One reason is that OutsideIn and Ambivalent don't reconcile different branches when they have different types. We can see that whenever $\mathscr{P}$ successfully infers a type for a program, then so does Chore. On the other hand, Chore infers types for several programs on which $\mathscr{P}$ fails.

Two facts explain this behavior. First, since both approaches use a similar idea to reconcile competing types from different branches, they can do better than OutsideIn and Ambivalent. Second, since Chore uses a more precise choice type representation and reconciles types after type inference is finished (and thus when more information is available), it can reconcile types from different case expressions, which $\mathscr{P}$ cannot. The choice representation also allows us to pass more precise types when recursive definitions are involved.

It is interesting to look at the programs for which Chore (partially) fails. For `equ1`, Chore infers a type, but the result is not satisfying. The program `equ1` is defined as follows.

```
data Equ a b where {Refl :: Equ a a}

equ1 e x = case e of Refl -> x
```

The function definition is very simple, and it is exactly this simplicity that causes Chore, and also $\mathscr{P}$, to infer a less precise type. The function doesn't contain enough structure to allow Chore to correctly apply reconciliation. Without type annotations, the intended type of this function is also not clear. There are infinitely many types that can be assigned to `equ1`, for example,

```
    Equ a b -> a -> b, or
    Equ a b -> (Int -> a) -> (Int -> b)
```

Chore, $\mathscr{P}$, and Ambivalent infer `Equ a a -> b -> b`, while OutsideIn fails to infer a type. In fact, no approach can perform better for this example unless annotations are given. (This is also the reason we exclude the examples from [10]: All their examples are minor variations of `equ1`.)

Next let's turn our attention to the functions `u1` and `u2`, which are defined as follows.

```
u1 x y = (case x of {RI _ -> RI; RB _ -> RB})
         (case y of {RI z -> z})

u2 x y = (case x of {RI _ -> RI; RB _ -> RB})
         (case y of {RI z -> z; RB z -> z})
```

While `u1` is not well typed because the second alternative of the first case expression is unreachable, only Chore and $\mathscr{P}$ reject this expression for the correct reason. OutsideIn and Ambivalent reject it for wrong reasons. The very similar expression `u2`, however, is well typed. While both Chore and $\mathscr{P}$ infer the correct type `R` $\alpha \rightarrow$ `R` $\alpha \rightarrow$ `R` α for `u2`, OutsideIn and Ambivalent fail to infer a type, even when the type annotation is added.

Let's look at some more ill-typed programs. We begin with the expression `gadt7o` defined in Section 6.1. This program is ill typed because for any list either the first or second tuple component will fail to match. We can see that Chore, $\mathscr{P}$, and Ambivalent correctly detect this error. GHC fails to infer a type for it, but the reason is that type inference has touched some untouchable type variables [27]. When annotated with `L n a -> (a,a)`, GHC type inference succeeds, which it shouldn't.

While $\mathscr{P}$ and Ambivalent can reject the programs in this simple form, they are incapable of detecting errors in more complicated cases. For example, both of them accept the program `gadt7q`, introduced in Section 6.1. Ambivalent and $\mathscr{P}$ accept `gadt7q` because they apply reconciliation only at each case expressions and thus miss the global type constraint that exists across the three components and that ensures a pattern-match error.

Another interesting example is `h2`. Previous approaches are unable to detect that the type of the argument doesn't match any of the pattern types; they infer the type `R a -> a`.

To summarize, Chore can better detect two kinds of runtime failures than previous approaches: inconsistent requirements from different places of a program on the same variable and mismatches between functions and their arguments.

Besides the programs presented in Figure 7, we evaluated our approach on a large set of programs collected by Lin [17], which consists of 63 GADT programs covering a wide range of applications, including dimensional types, length-indexed lists, N-way zip, tagless term interpreters, balance-indexed AVL trees, and many others. $\mathscr{P}$ successfully infers types for 52 programs and fails on 11 others. Chore successfully infers types for 58 programs, including the 52 programs that $\mathscr{P}$ is successful on. Chore fails on 5 programs. Since OutsideIn and Ambivalent fail to infer types for almost all of the programs, a detailed comparison is not very illuminating.

Finally, we also measured the running time of type inference to determine the overhead of our approach. Compared to $\mathscr{P}$, the overhead of our algorithm for each program is always within 55%. A comparison with GHC is of limited use since it requires type annotations for all the programs, and type inference is more efficient when annotations are present. Moreover, unlike GHC, $\mathscr{P}$ and our approach are not optimized for performance. While 55% overhead is not great, it is still surprisingly low considering the potential complexity added by choice types and their unification. There are several reasons for this. First, in many examples there are only few type index variables, which makes coherence checking and reconciliation quite fast. Second, most choices contain few alternatives, and not too many choices are generated during type inference. Finally, the nesting of choices, which has a major influence on the complexity of choice unification, is also quite limited.

Theoretically, as type inference with let-polymorphism is exponential in the number of nesting levels of let expressions, our type inference algorithm is additionally exponential in the number of nesting levels of case subexpressions. This theoretical worst-case complexity has never affected the practical applicability of our inference algorithms. In practice, the nesting level seldom exceeds five. In fact, no example program presented in Section 7 or any program in our study of the Hackage libraries exceeds that number. This coincides with Henglein's observation that theoretical intractability of type inference problems usually doesn't affect the practical utility of type inference algorithms [11].

8. Discussion

First, we discuss the relation between precision and principality of GADT type inference and present some empirical evaluation results. Then, based on the evaluation results in Section 7 and the discussion in Section 8.1, we compare the different approaches to GADT type inference in Section 8.2 along different criteria.

8.1 Precision or Principality: An Empirical Evaluation

Since GADTs lose the principal type property, an important design decision for a type inference algorithm is to choose between principality and precision. For example, both OutsideIn [27, 34] and Ambivalent [10] are principal, but $\mathscr{P}$ [17] trades principality for precision. Principality is the fundamental principle that underlies the algorithm $\mathscr{W}$ [7], the foundation of most type inference algorithms in use. The advantage of precise types is that they reflect more closely the shapes of expressions.

Theorem 5 shows that before reconciliation our type inference algorithm is principal. Thus, we don't need to worry about this design decision until we present a result type to a programmer. Reconciliation favors precision over principality for the following reasons. First, while many programs don't have principal types, they have most precise types. For example, the expression `flop1` doesn't have a principal type, but it has a most precise type, `R Int` $\to$ `Int`, the one reported by Chore. Second, this decision allows us to type more programs without type annotations. Third, it aligns better with the fact the choice types make the GADT type system more precise to catch more errors since precise types better characterize the shapes of programs.

Notably, this decision aligns with the usage found in actual Haskell programs. To see what Haskell programmers prefer in practice, we looked at how GADT programs in Haskell libraries are annotated. In particular, if a function can have multiple annotations, we determined whether a more general or more specific annotation was chosen to glean the preference for principality or precision, respectively. We randomly chose 300 files that use GADTs from libraries hosted on Hackage (as of Oct. 23rd, 2014) and inspected each file. We observed that in each case there were many functions that can be annotated with different types, and that for every one of them, the most specific type was chosen as the annotation. This lends strong support for our decision to favor more precise types over principal types.

8.2 Comparison of Approaches to GADT Type Inference

Based on the evaluation in Section 7 and the discussion in Section 8.1, we can compare the different GADT type inference approaches on a high level and put our observations into perspective.

First of all, every approach has as one of its goal reducing the number of type annotations required of programmers. While Chore and $\mathscr{P}$ approach this goal by designing advanced reconciliation strategies, OutsideIn and Ambivalent resort to propagating user annotations and type information across parts of the program. The goal of saving programmers from annotations is best supported by Chore. $\mathscr{P}$ comes in second, followed by Ambivalent and GHC.

Unfortunately, none of the approaches can provide a simple rule about when and where type annotations are needed. While Chore and $\mathscr{P}$ can infer types for most programs without type annotations, they fail to do so for some programs for a variety of reasons (discussed in Section 7). Also, while in most cases annotating the top-level function definitions are sufficient for OutsideIn and Ambivalent, this is not always the case. For example, GHC fails to accept `u2` even with annotations. In contrast, Chore successfully accepts all well-typed programs and rejects ill-typed programs.

Second, regarding principality, GHC and Ambivalent have principal types, while $\mathscr{P}$ hasn't. Chore has principal types before reconciliations. Moreover, it seems to be quite easy to turn our type system into a principal one by changing the reconciliation rules.

Third, somewhat dual to principality is the goal of preciseness, where we observe that Chore and $\mathscr{P}$ are more precise than GHC and Ambivalent.

Finally, the goal of detecting runtime errors is best achieved by Chore, followed by $\mathscr{P}$ and Ambivalent with GHC coming in last.

9. Related Work

This work is inspired conceptually by [17] and technically by [4]. Lin and Sheard [18] were the first to investigate full GADT type inference without the need for type annotations. Their approach immediately resolves potential inconsistencies between branch types. Performing reconciliation for each case expression precludes the possibility of using global type information, and it can also lose some type information in recursive definitions. Both aspects reduce the precision of inferred types.

To address the shortcomings of that approach, we have employed the concept of choice types from [4, 5] to represent the types for case expressions, which relieves the need for local reconciliations and enables reconciliation to exploit global type information. The use of choice types also allows us to detect more runtime errors statically, which distinguishes our approach from others.

We reuse parts of the pattern formalism from [4], but there are significant differences in how we apply it. First, we use $\perp$ to denote that corresponding parts of two types don't match, whereas in [4] $\perp$ is part of type syntax and is used to denote a type error. Second, result types are extracted differently from the return types when typing applications. All the $\perp$s in patterns become part of the result types of applications in [4]. Finally, choices are introduced and removed differently. While in [4] choices are given by the input expressions and may be carried over to the typing result, choices in this work are created on the fly, and when communicating with users, they are removed through the reconciliation process.

Due to the presence of polymorphic recursion and the need for reconciling different local assumptions, type annotations are needed to restore decidable and principal GADT type inference. The idea of using type annotations to assist GADT type inference was independently proposed by Simonet and Pottier [30] and by Stuckey and Sulzmann [31], who demonstrated the difficulty of full type inference. Simonet and Pottier showed that the problem of type inference for HMG(X), an extension of HM(X) [20] with GADTs, can be reduced to the problem of satisfiability checking of formulas consisting of finite trees, conjunctions, implications, and existential and universal quantifications. The latter problem was shown to be intractable [33].

The first GADT type inference approach using type annotations was presented in [21, 22]. The notions of rigid types and wobbly types denote the type information derived from user annotations and computed by the inference algorithm, respectively. Only rigid types support type refinements in case branches. Wobbly types are similar to choice types in that they distribute over type constructors. However, other operations on choice types behave differently. For example, substitutions are applied recursively to choice alternatives, while substitutions don't affect wobbly types.

While wobbly types mix annotation propagation and the type inference process, stratified types [25] handle GADT type inference in a modular way by separating the two. The first phase transforms annotated GADT programs into an intermediate language and generates annotations for case scrutinees and all local assumptions for case branches. The second phase, which is decidable and complete, infers types for the intermediate language. Since the first phase is incomplete, the whole problem is still incomplete. Still, the separation facilitates the exploration of different propagation strategies.

OutsideIn also separates propagation and inference [27, 34]. However, the first phase propagates not only user annotations but also the inferred types of the program parts that don't involve GADTs. Type inference for GADT case branches is postponed until the second phase, when the type information about the context is available. Although OutsideIn propagates more type information, this doesn't seem to lower the amount of required user annotations (cf. Figure 7), a phenomenon also observed by Lin [17].

While enough type annotations will make GADT type inference decidable and complete, it is unclear how many are needed and where [10, 17]. Annotating functions is not always sufficient for OutsideIn and Ambivalent. On the other hand, based on the evaluation in Section 7 we observe that our approach Chore can successfully infer types for most of the programs without annotations, which indicates that Chore is complementary to annotation-based approaches.

Regarding information propagation, stratified types don't allow inferred types to be propagated, OutsideIn allows inferred types of expressions to be propagated, except for GADT case branches, and ambivalent types [10] even allow type information from GADT case branches to be propagated as long as this doesn't change the set of convertible types.[5] Both $\mathscr{P}$ and Chore have no restrictions on what kind of information can be propagated, as long as it can be reconciled later, either locally, as in $\mathscr{P}$, or globally, as in Chore.

To some extent, choice types are similar to union types [23]. We have previosuly discussed the relation between union types and choice types in depth [5]. For this paper, a natural question is whether we can use union types instead of choice types for GADT type inference. The answer is no. With union types, we can attempt to solve the type inference problem with two potential different type representations, both of which fall short. The first representation is to allow union types combined freely with other type constructors, much like choice types can interact with other types. This representation is too imprecise for GADT type inference. Consider, for example, the function `cross` introduced in Section 6.2. The union type representation yields the type $\texttt{R}\ (\texttt{Int} \vee \texttt{Bool}) \rightarrow \texttt{Bool} \vee \texttt{Int}$ for `cross`. Since union types are equivalent modulo commutativity, we can reconcile the inferred type to $\texttt{R}\ \alpha \rightarrow \alpha$, which is incorrect. The second representation allows unions to only occur at the top level. However, this representation is too restrictive. Consider, for example, the function `u2` defined in Section 7. The body of `u2` is an application of one case expression to another. With the second representation, we would assign $\texttt{Int} \rightarrow \texttt{R Int} \vee \texttt{Bool} \rightarrow \texttt{R Bool}$ to the function and $\texttt{Int} \vee \texttt{Bool}$ to the argument. Since the only operations that can be applied to values of union types are those that make sense for all types constituting the union type [23], we can't assign a type to `u2`. However, `u2` is well typed and successfully receives a type with choice types.

The improvement of GADT type inference in this paper is realized through a better characterization of case-branch types with choice types. Through a compact symbolic value representation of a set of values, Karachalias et al. [12] proposed an approach that gives accurate warning for missing and overlapping patterns in presence of GADTs. By extending the pattern checking algorithm in OCaml, Garrigue and Le Normand [9] presented an approach for reporting missing patterns with GADTs. Our work also checks problems related with patterns. The difference is that our work detects conflicting type requirements that originate from different case expressions while others focus on a single case expression. Another difference is that we can detect a mismatch between the type of a case expression's scrutinee and the type of the expression it is applied to.

The work of first-class cases [1] also introduced a notion of type refinements. GADTs and first-class cases are quite different. For example, GADTs support type-level computations that are missing in first-class cases. Also, only GADTs introduce local assumptions, leading to the loss of the principality of type inference.

[5] These are types that have the same meaning within a certain context. For example, if `flop1` has the type $\texttt{R}\ \alpha \rightarrow \alpha$, then α and `Int` are convertible inside of `flop1`.

10. Conclusions

We have presented Chore, a new method for GADT type inference that improves the precision of previous approaches by accepting more well-typed programs and rejecting more programs that will lead to runtime errors. Our approach is based on an extension of the type language by choice types, which facilitate a more precise characterization of the types of case alternatives and a separation of typing and reconciliation.

Like previous approaches, Chore can benefit from type annotations to expand the set of typeable program. In future work we will study under which conditions exactly our algorithm needs type annotations to provide clear type-annotation guidelines for programmers. We will also investigate how to exploit choice types to provide better GADT type error messages. Finally, we will apply our approach to detect more pattern-matching failures for traditional ADT programs.

References

[1] M. Blume, U. A. Acar, and W. Chae. Extensible programming with first-class cases. In *ACM SIGPLAN International Conference on Functional Programming*, pages 239–250, 2006.

[2] S. Chen and M. Erwig. Counter-factual typing for debugging type errors. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 583–594, 2014.

[3] S. Chen and M. Erwig. Guided type debugging. In *Functional and Logic Programming*, LNCS 8475, pages 35–51. 2014.

[4] S. Chen, M. Erwig, and E. Walkingshaw. An error-tolerant type system for variational lambda calculus. In *ACM SIGPLAN International Conference on Functional Programming*, pages 29–40, 2012.

[5] S. Chen, M. Erwig, and E. Walkingshaw. Extending type inference to variational programs. *ACM Trans. Program. Lang. Syst.*, 36(1):1:1–1:54, Mar. 2014.

[6] J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics. In *ACM SIGPLAN Workshop on Haskell*, pages 90–104, 2002.

[7] L. Damas and R. Milner. Principal Type-Schemes for Functional Programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[8] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to design programs*. MIT Press Cambridge, 2001.

[9] J. Garrigue and J. L. Normand. Adding gadts to ocaml: the direct approach. In *Workshop on ML*, 2011.

[10] J. Garrigue and D. Rémy. Ambivalent types for principal type inference with GADTs. In *Programming Languages and Systems*, LNCS 8301, pages 257–272. 2013.

[11] F. Henglein. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289, Apr. 1993.

[12] G. Karachalias, T. Schrijvers, D. Vytiniotis, and S. P. Jones. Gadts meet their match: Pattern-matching warnings that account for gadts, guards, and laziness. In *ACM SIGPLAN International Conference on Functional Programming*, pages 424–436, 2015.

[13] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *ACM SIGPLAN Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 805–824, 2011.

[14] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):290–311, Apr. 1993.

[15] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable analysis of variable software. In *Foundations of Software Engineering*, pages 81–91, 2013.

[16] C.-k. Lin. Programming monads operationally with unimo. In *ACM SIGPLAN International Conference on Functional Programming*, pages 274–285, 2006.

[17] C.-k. Lin. *Practical Type Inference for the GADT Type System*. PhD thesis, Portland State University, 2010.

[18] C.-k. Lin and T. Sheard. Pointwise generalized algebraic data types. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 51–62, 2010.

[19] A. Mycroft. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*, LNCS 167, pages 217–228, 1984.

[20] M. Odersky, M. Sulzmann, and M. Wehr. Type Inference with Constrained Types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[21] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ACM SIGPLAN International Conference on Functional Programming*, pages 50–61, 2006.

[22] S. Peyton Jones, G. Washburn, and S. Weirich. Wobbly types: type inference for generalised algebraic data types. Technical Report MS-CIS-05-26, University of Pennsylvania, July 2004.

[23] B. C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, School of Computer Science, Carnegie Mellon University, 1991.

[24] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[25] F. Pottier and Y. Régis-Gianas. Stratified type inference for generalized algebraic data types. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–244, 2006.

[26] N. Ramsey. On teaching *how to design programs*: Observations from a newcomer. In *ACM SIGPLAN International Conference on Functional Programming*, pages 153–166, 2014.

[27] T. Schrijvers, S. Peyton Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for GADTs. In *ACM SIGPLAN International Conference on Functional Programming*, pages 341–352, 2009.

[28] T. Sheard. Generic programming in Omega. In *Datatype-Generic Programming*, LNCS 4719, pages 258–284, 2006.

[29] T. Sheard and N. Linger. Programming in Omega. In *CEFP*, LNCS 5161, pages 158–227, 2007.

[30] V. Simonet and F. Pottier. A constraint-based approach to guarded algebraic data types. *ACM Trans. on Programming Languages and Systems*, 29(1):1–38, 2007.

[31] P. J. Stuckey and M. Sulzmann. Type inference for guarded recursive data types. *CoRR*, abs/cs/0507037:1–15, 2005.

[32] M. Sulzmann, T. Schrijvers, and P. J. Stuckey. Type inference for GADTs via Herbrand constraint abduction. Technical Report CW507, University of Leuven, January 2008.

[33] S. Vorobyov. An improved lower bound for the elementary theories of trees. In *International Conference on Automated Deduction*, LNCS 1104, pages 275–287. 1996.

[34] D. Vytiniotis, S. Peyton Jones, T. Schrijvers, and M. Sulzmann. Outsidein(x) modular type inference with local assumptions. *Journal of Functional Programming*, 21(4-5):333–412, Sept. 2011.

[35] S. Weirich. Depending on types. In *ACM SIGPLAN International Conference on Functional Programming*, pages 241–241, 2014.

[36] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 224–235, 2003.