Ambivalent Types for
Principal Type Inference with GADTs

Jacques Garrigue[1] and Didier Rémy[2]

[1] Nagoya University, Graduate School of Mathematics
[2] INRIA, Rocquencourt*

Abstract. *GADTs, short for* Generalized Algebraic DataTypes*, which allow constructors of algebraic datatypes to be non-surjective, have many useful applications. However, pattern matching on GADTs introduces local type equality assumptions, which are a source of ambiguities that may destroy principal types—and must be resolved by type annotations. We introduce* ambivalent types *to tighten the definition of ambiguities and better confine them, so that type inference has principal types, remains monotonic, and requires fewer type annotations.*

1 Introduction

GADTs, short for *Generalized Algebraic DataTypes*, extend usual algebraic datatypes with a form of dependent typing by enabling type refinements in pattern-matching branches [2,14,1]. They can express many useful invariants of data-structures, provide safer typing, and allow for more polymorphism [11]. They have already been available in some Haskell implementations (in particular GHC) for many years and now appear as a natural addition to strongly typed functional programming languages.

However, this addition is by no means trivial. In their presence, full type inference seems undecidable in general, even in the restricted setting of ML-style polymorphism [10]. Moreover, many well-typed programs lack a most general type. Using explicit type annotations solves both problems. Unfortunately, while it is relatively easy to design a sound typing algorithm for a language with GADTs, it is surprisingly difficult to keep principal types without requesting full type annotations on every case analysis.

Repeatedly writing full type annotations being cumbersome, a first approach to a stronger type inference algorithm is to *propagate* annotations. This comes from the basic remark that, in many cases, the type of a function contains enough information to determine the type of its inner case analyses. A simple way to do this is to use program transformations, pushing type annotations inside the body of expressions.

Stratified type inference for GADTs [9] goes further in that direction, converting from an external language where type annotations are optional to an internal language where the scrutinee of case analysis and all coercions between equivalent types must be annotated. This conversion is an elaboration phase that collects all *typing information* —not only type annotations— and propagates it where it is needed. The internal language allows for straightforward type inference and it has the principal type property.

---

* Part of this work has been done at IRILL.

It also enjoys *monotonicity*: strengthening the type of a free variable by making it more general preserves well-typedness. As expected, principality does not hold in general in the external type system (a program may be typable but have no principal type), but it does hold if we restrict ourselves to those programs whose elaboration in the internal language is typable. However, since elaboration extracts information from the typing context, monotonicity is lost: strengthening the type of a free variable by making it more general before elaboration can reduce the amount of type information available on the elaborated program and make it ill-typed. Monotonicity is a property that has often been underestimated, because it usually (but not always) holds in languages with principal types. However, losing monotonicity can be worse for the programmer than losing principal types. It reveals a lack of modularity in the language, since some simple program transformations such as simplifying the body of a function may end up inferring more general types, which may subsequently break type inference. Propagating only type annotations would preserve monotonicity, but it is much weaker.

GHC 7 follows a similar strategy, called *OutsideIn* [13], using constraint solving rather than elaboration to extract all typing information from the *outer context*. As a result, propagation and inference are interleaved. That is, the typing information obtained by solving constraints on the outer context enclosing a GADT case analysis is directly used to determine the types of both the scrutinee and the result in this case analysis. Type inference can then be performed in the body of the case analysis. By allowing information to flow only from the outside to the inside, principality is preserved when inference succeeds. Yet, as for stratified type inference [9], it lacks monotonicity.

While previous approaches have mostly attempted to propagate types to GADT case analyses, we aim in the opposite direction at reducing the need for type information in case analysis. This aspect is orthogonal to propagation and improving either one improves type inference as a whole. Actually, OutsideIn already goes one step in that direction, by allowing type information to flow out of a pattern-matching case when no type equation was added. But it stops there, because if type equations were added, they could have been used and consequently the type of the branch is flagged *ambiguous*.

This led us to focus our attention on the definition of ambiguity. Type equations are introduced inside a pattern-matching branch, but with a *local scope*: the equation is not valid outside of the branch. This becomes a source of ambiguities. Indeed, a type equation allows implicit type conversions, *i.e.* there are several inter-convertible forms for types that we need not distinguish while in the scope of the equation, but they become nonconvertible—hence ambiguous—when leaving its scope, as the equation can no longer be used. Ambiguity depends both on the equations available, and on the types that leak outside of the branch: if removing the equation does not impair convertibility for a type, either because it was not convertible to start with, or because other equations are available, it need not be seen as ambiguous.

Since ambiguities must generally be solved by adding type annotations, a more precise definition and better detection of ambiguities become essential to reduce the need for explicit type information. By defining ambiguity inside the type system, we are able to restrict the set of valid typings. In this paper we present a type system such that among the valid typings there is always a principal one (*i.e.* subsuming all of them) and we provide a type inference algorithm that returns the principal solution when it exists.

Moreover, our type system keeps the usual properties of ML, including monotonicity. This detection of ambiguity is now part of OCaml [7].

Since propagating type information and reducing the amount of type information needed by case analysis are orthogonal issues, our handling of ambiguity could be combined with existing type inference algorithms to further reduce the need for type annotations. As less type information is needed, it becomes possible to use a weaker propagation algorithm that preserves monotonicity. This is achieved in OCaml by relying on the approach previously developed for first-class polymorphism [4].

The rest of this paper is organized as follows. We give an overview of our solution in §2. We present our system formally and state its soundness in §3. We state principality and monotonicity in §4; by lack of space, we leave out some technical developments, all proofs, and the description of the type inference algorithm, which can all be found in the accompanying technical report [5]. Finally, we compare with related works in §5.

2 An overview of our solution

The standard notion of ambiguity is so general that it may just encompass too many cases. Consider the following program.[3]

```
type (_,_) eq = Eq : (α,α) eq
let f (type a) (x : (a,int) eq) = match x with Eq -> 1
```

Type `eq` is the classical equality witness. It is a GADT with two index parameters, denoted by the two underscores, and a single case `Eq`, for which the indices are the same type variable α. Thus, a value of type (a,b) `eq` can be seen as a witness of the equality between types a and b.

In the definition of `f`, we first introduce an explicit universal variable a, called a *rigid* variable, treated in a special way in OCaml as it can be refined by GADT pattern matching. By constraining the type of `x` to be $(a,$`int`$)$ `eq`, we are able to refine a when pattern-matching `x` against the constructor `Eq`: the equation $a =$ `int` becomes available in the corresponding branch, *i.e.* when typechecking the expression `1`, which can be assigned either type a or `int`. As a result, `f` can be given either type $(\alpha,$`int`$)$ `eq` $\rightarrow$ `int` or $(\alpha,$`int`$)$ `eq` $\rightarrow \alpha$. This fulfills the standard definition of ambiguity and so should be rejected. But should we really reject it? Consider these two slight variations in the definition of `f`:

```
let f1 (type a) (x : (a,int) eq) = match x with Eq -> true
let f2 (type a) (x : (a,int) eq) (y : a) = match x with Eq -> (y > 0)
```

In `f`$_1$, we just return `true`, which has the type `bool`, unrelated to the equation. In `f`$_2$, we actually use the equation to turn `y` into an `int` but eventually return a boolean. These variants are not ambiguous. How do they differ from the original `f`? The only reason we have deemed `f` to be ambiguous is that `1` could potentially have type a by using the equation. However, nothing forces us to use this equation, and, if we do not use it, the only possible type is `int`. It looks even more innocuous than `f`$_2$, where we indirectly need the equation to infer the type of the body.

So, what would be a truly ambiguous type? We obtain one by mixing a's and `int`'s in the returned value (the left-margin vertical rules indicate failure):

---

[3] Examples in this section use OCaml syntax [7]. Letter α stands for a flexible variable as usual while letter a stands for a rigid variable that cannot be instantiated. This will be detailed later.

```
let g (type a) (x : (a,int) eq) (y : a) =
  match x with Eq -> if y > 0 then y else 0
```

Here, the `then` branch has type a while the `else` branch has type `int`, so choosing either one would be ambiguous.

How can we capture this refined notion of ambiguity? The idea is to track whether such mixed types are escaping from their scope. Intuitiveley, we may do so by disallowing the expression to have either type and instead viewing it with an ambivalent type $a \approx$ `int`, which we just see syntactically as a set of types.

An ambivalent type must still be *coherent*, *i.e.* all the types it contains must be provably equal under the equations available in the current scope. Hence, although $a \approx$ `int` can be interpreted as an intersection type, it is not more expressive than choosing either representation (since by equations this would be convertible to the other type), but more precise: it retains the information that the equivalence of a and `int` has been assumed to give the expression the type a or `int`.

Since coherence depends on the typing context, a coherent ambivalent type may suddenly become incoherent when leaving the scope of an equation. This is where *ambiguity* appears. Hence, while an ambivalent type is a set of types that have been assumed interchangeable, an ambiguity arises only when an ambivalent type becomes incoherent by escaping the scope of an equation it depends on.

Ambiguous programs are to be rejected. Fortunately, ambiguities can be eliminated by using type annotations. Intuitively, in an expression $(e : \tau)$, the expressions e and $(e : \tau)$ have sets of types ψ_1 and ψ_2 that may differ, but such that τ is included in both, ensuring soundness of the change of view. In particular, while the inner view, *e.g.* ψ_1, may be large and a potential source of ambiguities, the outer view, *e.g.* ψ_2, may contain fewer types and remain coherent; this way the ambivalence of the inner view does not leak outside and does not create ambiguities. Consider, for example the program:

```
let g1 (type a) (x : (a,int) eq) y =
  match x with Eq -> (if (y : a) > 0 then (y : a) else 0 : a)
```

Type annotations on `y` and the conditional let them have unique outer types, which are thus unambiguous when leaving the scope of the equation. More precisely, `(y : a)` and `0` can be both assigned type $a \approx$ `int`, which is also that of the conditional `if ... else 0`, while the annotation `(if ... else 0 : a)` and variable `y` both have the singleton type a. (Note that the type of the annotated expression is the inner view for y but the outer view for the conditional.)

Of course, it would be quite verbose to write annotations everywhere, so in a real language we shall let annotations on parameters propagate to their uses and annotations on results propagate inside pattern-matching branches. The function g$_1$ may be written more concisely as follows—but we will ignore this aspect in this work:

```
let g2 (type a) (x : (a,int) eq) (y : a) : a =
  match x with Eq -> if y > 0 then y else 0
```

A natural question at this point is why not just require that the type of the result of pattern-matching a GADT be fully known from annotations? This would avoid the need for this new notion of ambiguity. This is perhaps good enough if we only consider small functions: as shown for g$_2$, we may write the function type in one piece and still get the full type information. However, the situation degrades with local `let` bindings:

```
let p (type a) (x : (a,int) eq) : int =
let y = (match x with Eq -> 1) in y * 2
```

The return type `int` only applies to `y*2`, so we cannot propagate it automatically as an annotation for the definition of `y`. Basically, one would have to explicitly annotate all `let` bindings whose definitions use pattern-matching on GADTs. This may easily become a burden, especially when the type is completely unrelated to the GADTs (or accidentally related as in the definition of `f`, above).

We believe that our notion of ambiguity is simple enough to be understood easily by users, avoids an important number of seemingly redundant type annotations, and provides an interesting alternative to non-monotonic approaches (see §5 for comparison).

3 Formal presentation

Since our interest is type inference, we may assume without loss of generality that there is a unique predefined (binary) GADT $\mathsf{eq}(\cdot,\cdot)$ with a unique constructor Eq of type $\forall(\alpha)\ \mathsf{eq}(\alpha,\alpha)$. The type $\mathsf{eq}(\tau_1,\tau_2)$ denotes a witness of the equality of τ_1 and τ_2 and Eq is the unique value of type $\mathsf{eq}(\tau_1,\tau_2)$. For conciseness, we specialize pattern matching to this unique constructor and just write $\mathsf{use}\, M_1 : \tau \,\mathsf{in}\, M_2$ for `match` $M_1 : \tau$ `with Eq ->` M_2.

Types occurring in the source program are simple types:

$$\tau ::= \alpha \mid a \mid \tau \to \tau \mid \mathsf{eq}(\tau,\tau) \mid \mathtt{int}$$

Type variables are split into two different syntactic classes: flexible type variables, written α, and rigid type variables, written a. As usual, flexible type variable are meant to be instantiated by any type—either during type inference or after their generalization. Conversely, rigid variables stand for some unknown type and thus are not meant to be instantiated by an arbitrary type. They behave like skolem constants. We write $\mathscr{V}$, $\mathscr{V}_f$, and $\mathscr{V}_r$ for the set of variables, flexible variables, and rigid variables.

Terms are expressions of the λ-calculus with constants (written c), the datatype Eq, pattern matching $\mathsf{use}\, M_1 : \tau \,\mathsf{in}\, M_2$, the introduction of a rigid variable $\nu(a) M$ or a type annotation (τ), *i.e.* the usual annotation $(M : \tau)$ is seen as the application $(\tau)\, M$:

$$\begin{array}{rcl} M & ::= & x \mid c \mid M_1\, M_2 \mid \lambda(x) M \mid \mathsf{let}\, x = M_1 \,\mathsf{in}\, M_2 \\ & \mid & \mathsf{Eq} \mid \mathsf{use}\, M_1 : \tau \,\mathsf{in}\, M_2 \mid \nu(a) M \mid (\tau) \end{array}$$

Although type annotations in source programs are simple types, their flexible type variables are interpreted as universally quantified in the type of the annotation (see §3.5).

Besides, we use—and infer—*ambivalent types* internally to keep track of the use of typing equations and detect ambiguities more accurately.

3.1 Ambivalent types

Intuitively, ambivalent types are sets of types. Technically, they refine simple types to express certain type equivalences within the structure of types. Every node becomes a set of type expressions instead of a single type expression and is labeled with a flexible type variable. More precisely, ambivalent types, written ζ, are recursively defined as:

$$\rho ::= a \mid \zeta \to \zeta \mid \mathsf{eq}(\zeta,\zeta) \mid \mathtt{int} \qquad \psi ::= \varepsilon \mid \rho \approx \psi \qquad \zeta ::= \psi^{\alpha} \qquad \sigma ::= \forall(\bar\alpha)\ \zeta$$

A raw type ρ is a rigid type variable a, an arrow type $\zeta \rightarrow \zeta$, an equality type $\mathsf{eq}(\zeta, \zeta)$, or the base type `int`. A *proper* raw type is one that is not a rigid type variable. An *(ambivalent) type* ζ is a pair ψ^α of a *set* ψ of *raw types* ρ labeled with a *flexible type variable* α. We use $\approx$ to separate the elements of sets of raw types: it is associative commutative, has the empty set ε for neutral element, and satisfies the idempotence axiom $(\psi \approx \psi) = \psi$. An ambivalent type ζ is always of the form ψ^α and we write $\lfloor \zeta \rfloor$ for ψ. When ψ is empty ζ is a leaf of the form ε^α, which corresponds to a type variable in simple types, hence we may just write α instead of ε^α, as in the examples above.

Type schemes σ are defined as usual, by generalizing zero or more flexible type variables. Rigid type variables may only be used free and cannot be quantified over. We introduce them in the typing environment but turn them into flexible type variables before quantifying over them, so they never appear as bound variables in type schemes.

In our representation, every node is labeled by a flexible type variable. This is essential to make type inference modular, as it is needed for incremental instantiation.

To see this, consider a context that contains a rigid type variable a, an equation $a \doteq \mathtt{int}$, and a variable x of type a, under which we apply a function `choice` of type $\alpha \rightarrow \alpha \rightarrow \alpha$ to x and 1. We first reason in the absence of labels on inner nodes. The partial application `choice` x has type $a \rightarrow a$. To further apply it to 1, we must use the equation to convert both 1 of type `int` and the domain of the partial application to the ambivalent type $\mathtt{int} \approx a$. The type of the full application is then a. However, if we inverted the order of arguments, it would be `int`. Something must be wrong. In fact, if we notice in advance that both types a and `int` will eventually have to be converted to $\mathtt{int} \approx a$, we may see both x and 1 with type $\mathtt{int} \approx a$ before performing the applications. In this case, we get yet another result $\mathtt{int} \approx a$, which happens to be the right one.

What is still wrong is that as soon as we instantiate α, we lose the information that all occurrences of α must be synchronized. The role of labels on inner nodes is to preverve this information. Revisiting the example, the partial application now has type $a^\alpha \rightarrow a^\alpha$ (we still temporarily omit the annotation on arrow types, as they do not play a role in this example). This is saying that the type is currently $a \rightarrow a$ but remembering that the domain and codomain must be kept synchronized. Then, the interger 1 of type $\mathtt{int}^\gamma$ can also be seen with type $(\mathtt{int} \approx a)^\gamma$ and unified with the domain of the function a^α, with the effect of replacing all occurrences of a^α and of $\mathtt{int}^\gamma$ by $(\mathtt{int} \approx a)^\alpha$. Thus, the function has type $(\mathtt{int} \approx a)^\alpha \rightarrow (\mathtt{int} \approx a)^\alpha$ and the result of the application has type $(\mathtt{int} \approx a)^\alpha$—the correct one. We now obtain the same result whatever the scenario.

This result type may still be unified with some other rigid variable a', as long as this is allowed by having some equation $a' \doteq \mathtt{int}$ or $a' \doteq a$ in the context, and refine its type to $(\mathtt{int} \approx a \approx a')^\alpha$. Since we cannot tell in advance which type constructors will eventually be mixed with other ones, all nodes must keep their label when substituted.

Replaying the example with full label annotations, `choice` has type $\forall(\alpha, \gamma, \gamma')$ $(\alpha \rightarrow (\alpha \rightarrow \alpha)^\gamma)^{\gamma'}$ and its partial application to `x` has type $\forall(\alpha, \gamma)$ $(a^\alpha \rightarrow a^\alpha)^\gamma$ after generalization. Observe that this is less general than $\forall(\alpha, \alpha', \gamma)$ $(a^\alpha \rightarrow a^{\alpha'})^\gamma$ but more general than $\forall(\alpha, \gamma)$ $((\mathtt{int} \approx a)^\alpha \rightarrow (\mathtt{int} \approx a)^\alpha)^\gamma$.

Type variables. Type variables are either rigid variables a or flexible variables α. We write $\mathsf{frv}(\zeta)$ for the set of rigid variables that are free in ζ and $\mathsf{ffv}(\zeta)$ for the set of flexible variables that are free in ζ. These definitions are standard. For example, free flexible variables are defined as:

$$\begin{array}{rlrl}
\mathsf{ffv}(\psi^{\alpha}) &= \{\alpha\} \cup \mathsf{ffv}(\psi) & \mathsf{ffv}(a) &= \emptyset \\
\mathsf{ffv}(\varepsilon) &= \emptyset & \mathsf{ffv}(\mathsf{int}) &= \emptyset \\
\mathsf{ffv}(\rho \approx \psi) &= \mathsf{ffv}(\rho) \cup \mathsf{ffv}(\psi) & \mathsf{ffv}(\zeta_1 \to \zeta_2) &= \mathsf{ffv}(\zeta_1) \cup \mathsf{ffv}(\zeta_2) \\
\mathsf{ffv}(\forall(\alpha)\ \sigma) &= \mathsf{ffv}(\sigma) \setminus \{\alpha\} & \mathsf{ffv}(\mathsf{eq}(\zeta_1, \zeta_2)) &= \mathsf{ffv}(\zeta_1) \cup \mathsf{ffv}(\zeta_2)
\end{array}$$

The definition is analogous for free rigid variables, except that $\mathsf{frv}(\psi^{\alpha})$ is equal to $\mathsf{frv}(\psi)$ and $\mathsf{frv}(a)$ is equal to $\{a\}$. We write $\mathsf{ftv}(\zeta)$ the subset of $\mathsf{ffv}(\zeta)$ of variables that appear as leaves, *i.e.* labeling empty nodes and $\mathsf{fnv}(\zeta)$ the subset of $\mathsf{ffv}(\zeta)$ that are labeling nonempty nodes. In well-formed types these two sets are disjoint, *i.e.* $\mathsf{ffv}(\zeta)$ is the disjoint union of $\mathsf{ftv}(\zeta)$ and $\mathsf{fnv}(\zeta)$.

Rigid type variables lie between flexible type variables and type constructors. A rigid variable a stands for explicit polymorphism: it behaves like a nullary type constructor and clashes, by default, with any type constructor and any other rigid variable but itself. However, pattern matching a GADT may introduce type equations in the typing context while type checking the body of the corresponding branch, which may allow a rigid type variable to be compatible with another type. Type equations are used to verify that all ambivalent types occurring in the type derivation are well-formed, which requires in particular that all types of a same node can be proved equal.

Interpretation of types. Ambivalent types may be interpreted as sets of simple types by unfolding ambivalent nodes as follows:

$$\begin{array}{rlrl}
[\![\varepsilon^{\alpha}]\!] &= \{\alpha\} & [\![\mathtt{int}]\!] &= \mathtt{int} \\
[\![(\rho_1 \approx \psi)^{\alpha}]\!] &= \bigcup_{\rho \in \rho_1 \approx \psi} [\![\rho]\!] & [\![\zeta_1 \to \zeta_2]\!] &= \{\tau_1 \to \tau_2 \mid \tau_1 \in [\![\zeta_1]\!], \tau_2 \in [\![\zeta_2]\!]\} \\
[\![a]\!] &= a & [\![\mathsf{eq}(\zeta_1, \zeta_2)]\!] &= \{\mathsf{eq}(\tau_1, \tau_2) \mid \tau_1 \in [\![\zeta_1]\!], \tau_2 \in [\![\zeta_2]\!]\}
\end{array}$$

The interpretation ignores labels of inner nodes. It is used below for checking coherence of ambivalent types, which is a semantic issue and does not care about sharing of inner nodes. For example, types $(\mathtt{int} \approx a)^{\alpha} \to (\mathtt{int} \approx a)^{\alpha}$ and $(\mathtt{int} \approx a)^{\alpha_1} \to (\mathtt{int} \approx a)^{\alpha_2}$ are interpreted in the same way, namely as $\{\mathtt{int} \to \mathtt{int}, a \to a, a \to \mathtt{int}, \mathtt{int} \to a\}$.

A type ζ is said *truly ambivalent* if its interpretation is not a singleton. Notice that ψ may be a singleton ρ even though ψ^{α} is truly ambivalent, since ambivalence may be buried deeper inside ρ, as in $((\mathtt{int} \approx a)^{\alpha} \to (\mathtt{int} \approx a)^{\alpha})^{\alpha_0}$.

Converting a simple type to an ambivalent type. Given a simple type τ, we may build a (not truly) ambivalent type ζ such that $[\![\zeta]\!] = \{\tau\}$. This introduces new variables $\bar{\gamma}$ that are in $\mathsf{fnv}(\zeta)$, while the variables of $\mathsf{ftv}(\zeta)$ come from τ. We write $\wr \tau \int$ for the most general type scheme of the form $\forall(\bar{\gamma})\ \zeta$, which is obtained by labeling all inner nodes of τ with different labels and quantifying over these fresh labels. For example, $\wr \mathtt{int} \to \mathtt{int} \int$ is $\forall(\gamma_0, \gamma_1, \gamma_2)\ (\mathtt{int}^{\gamma_1} \to \mathtt{int}^{\gamma_2})^{\gamma_0}$ and $\wr \alpha \to \alpha \int$ is $\forall(\gamma_0)\ (\varepsilon^{\alpha} \to \varepsilon^{\alpha})^{\alpha_0}$. Notice that free type variables of τ remain free in $\wr \tau \int$.

3.2 Typing contexts

Typing contexts Γ bind program variables to types, and introduce rigid type variables a, type equations $\tau_1 \doteq \tau_2$, and *node descriptions* $\alpha :: \psi$:

$$\Gamma ::= \emptyset \mid \Gamma, x : \sigma \mid \Gamma, a \mid \Gamma, \tau_1 \doteq \tau_2 \mid \Gamma, \alpha :: \psi$$

$$\textsc{wf-Ctx-Equal}\;\frac{\vdash \Gamma \qquad \Gamma \vdash \tau_1 \doteq \tau_2}{\vdash \Gamma, \tau_1 \doteq \tau_2} \qquad \textsc{wf-Type-Equal}\;\frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2 \qquad \mathsf{ftv}(\tau_1) = \mathsf{ftv}(\tau_2) = \emptyset}{\Gamma \vdash \tau_1 \doteq \tau_2} \qquad \textsc{wf-Type-Flex}\;\frac{\vdash \Gamma \qquad \alpha :: \psi \in \Gamma}{\Gamma \vdash \psi^{\alpha}}$$

$$\textsc{wf-Ctx-Flex}\;\frac{\vdash \Gamma \qquad \Gamma \vdash \psi \qquad \alpha \notin \mathsf{dom}(\Gamma)}{\vdash \Gamma, \alpha :: \psi} \qquad \textsc{wf-Type-Ambivalent}\;\frac{(\Gamma \vdash \rho)^{\rho \in \psi} \qquad \Gamma \Vdash \psi \qquad |\psi \setminus \mathcal{V}_r| \leq 1}{\Gamma \vdash \psi}$$

Fig. 1. Well-formedness of contexts and types (excerpt)

Both flexible and rigid type variables are explicitly introduced in typing contexts. Hence, well-formedness of types is defined relatively to some typing context.

In addition to routine checks, well-formedness judgments also ensure soundness of ambivalent types and coherent use of type variables.

Well-formedness of contexts $\vdash \Gamma$ is recursively defined with the well-formedness of types $\Gamma \vdash \rho$ and type schemes $\Gamma \vdash \sigma$. Characteristic rules are in Figure 1. It also uses the entailment judgment $\Gamma \Vdash \psi$, which means, intuitively, that all raw types appearing in the set ψ can be proved equal from the equations in Γ (see §3.3). The last premise of Rule WF-TYPE-AMBIVALENT ensures that ambivalent types contain at most one raw-type that is not a rigid variable. As usual well-formedness of contexts ensures that type variables are introduced before being used and that types are well-formed. It also ensures coherent use of type variables: alias constraints $\alpha :: \psi$ in the context Γ define a mapping that provides evidence that α is used coherently in the type σ. This is an essential feature of our system so that refining ambivalence earlier or later commutes, as explained above.

3.3 Entailment

Typing contexts may contain type equations. Type equations are used to express equalities between types that are known to hold when the evaluation of a program has reached a given program point. Type equations are added to the typing context while typechecking the expression at the current program point.

The set of equations in the context defines an equivalence between types. Rule WF-TYPE-AMBIVALENT shows that ambivalent types can only be formed between equivalent types: the well-formedness of the judgment $\Gamma \vdash \psi$ requires $\Gamma \Vdash \psi$, *i.e.* that all types in ψ are provably equal under the equations in Γ, which is critical for type soundness; the rightmost premise requires that at most one type in ψ is not a rigid variable. For example, the ambivalent types $\mathsf{int} \approx (\mathsf{int}^\gamma \to \mathsf{int}^\gamma)$ and $(\mathsf{int}^\gamma \to \mathsf{int}^\gamma) \approx (a^\gamma \to a^\gamma)$ are ill-formed. This is however not restrictive as the former would be unsound in any consistent context while the later could instead be written $(\texttt{int} \approx a)^\gamma \to (\texttt{int} \approx a)^\gamma$.

Well-formedness of a type environment requires that its equations do not contain free type variables. Equalities in Γ may thus be seen as unification problems where rigid variables are the unknowns. If they admit a principal solution, it is a substitution of the form $(a_i \mapsto \tau_i)^{i \in I}$; then, the set of equations $(a_i \doteq \tau'_i)^{i \in I}$ is equivalent to the equations in Γ. If the unification problem fails, then the equations are inconsistent—in the standard model where type constructors cannot be equated[4]. This is acceptable and it just means

---

[4] This is not always true for ML abstract types, as type constructors may be compatible in another context, but we do not address this problem here.

$$
\begin{array}{rcl}
(\psi^{\alpha_i})\theta = \zeta_i & \qquad & (a)\theta = a \\
(\psi^{\gamma})\theta = (\psi\theta)^{\gamma} & & (\mathtt{int})\theta = \mathtt{int} \\
(\rho_i^{i\in I})\theta = (\rho_i\theta)^{i\in I} & & (\zeta_1 \to \zeta_2)\theta = \zeta_1\theta \to \zeta_2\theta \\
(\forall(\alpha)\ \zeta)\theta = \forall(\alpha)\ \zeta(\theta \setminus \{\alpha\}) & & (\mathsf{eq}(\zeta_1, \zeta_2))\theta = \mathsf{eq}(\zeta_1\theta, \zeta_2\theta)
\end{array}
$$

Fig. 2. Application of substitution θ equal to $[\alpha_i \leftarrow \zeta_i]^{i\in I}$

that the current program point cannot be reached. Therefore, any ambivalent type is admissible in an inconsistent context.

The semantic judgment $\Gamma \Vdash \psi$ means by definition that any ground instance of Γ that satisfies the equations in Γ makes all types in the semantics of ψ equal. Formally:

Definition 1 (Entailment). *Let Γ be a typing environment. A ground substitution θ from rigid variables to simple types* models *Γ if $\theta(\tau_1)$ and $\theta(\tau_2)$ are equal for each equation $\tau_1 \doteq \tau_2$ in Γ. We say that Γ entails ψ and write $\Gamma \Vdash \psi$ if $\theta([\![\psi]\!])$ is a singleton for any ground substitution θ that models Γ.*

This gives a simple algorithm to check for entailment: compute the most general unifier θ of Γ; then $\Gamma \Vdash \psi$ holds if and only if $\theta([\![\psi]\!])$ is a singleton or θ does not exist.

3.4 Substitution

In our setting, substitutions operate on ambivalent types where type variables are used to label inner nodes of types and not just their leaves. They allow the replacement of an ambivalent node ψ^{α} by a "more ambivalent" one $\psi \approx \psi'^{\alpha}$, using the substitution $[\alpha \leftarrow (\psi \approx \psi')^{\alpha}]$; or merging two ambivalent nodes $\psi_1^{\alpha_1}$ and $\psi_2^{\alpha_2}$ using the substitution $[\alpha_1, \alpha_2 \leftarrow \psi_1 \approx \psi_2{}^{\alpha_1}]$. To capture all these cases with the same operation, we define in Figure 2 a general form of substitution $[\alpha_i \leftarrow \zeta_i]^{i\in I}$ that may graft arbitrary nodes ζ_i at every occurrence of a label α_i, written $[\alpha \leftarrow \zeta]$;

As a result of this generality, substitutions are purely syntactic and may replace an ambivalent node with a less ambivalent one—or even prune types replacing a whole subtree by a leaf. Of course, we should only apply substitutions to types when they preserve (or increase) ambivalence.

Definition 2. *A substitution θ* preserves ambivalence in a type *ζ if and only if, for any α in $\mathsf{dom}(\theta)$ and any node ψ^{α} in ζ, we have $\psi\theta \subseteq \lfloor (\psi^{\alpha})\theta \rfloor$.*

As a particular case, an atomic substitution $[\alpha \leftarrow \zeta_0]$ preserves ambivalence in ζ if for any node ψ^{α} in ζ, we have $\psi \subseteq \lfloor \zeta_0 \rfloor$—since well-formedness of ψ^{α} implies that α may not occur free in ψ, hence $\psi\theta$ is just ψ.

3.5 Typing rules

Typing judgments are of the form $\Gamma \vdash M : \sigma$ as in ML. However, typing rules, defined in Figure 3, differ from the traditional presentation of ML typing rules in two ways. On the one hand, we use a constraint framework where Γ carries node descriptions $\alpha :: \psi$ to enforce their sharing within different types. On the other hand, typing rules

$$\textsc{m-Var}\quad \frac{\vdash \Gamma \qquad x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad\qquad \textsc{m-Inst}\quad \frac{\Gamma \vdash M : \forall(\alpha)\ (\sigma[\alpha \leftarrow \psi_0^{\alpha}]) \qquad \psi_0 \subseteq \psi \qquad \Gamma \vdash \psi^{\gamma}}{\Gamma \vdash M : \sigma[\alpha \leftarrow \psi^{\gamma}]}$$

$$\textsc{m-Gen}\quad \frac{\Gamma, \alpha :: \psi \vdash M : \sigma}{\Gamma \vdash M : \forall(\alpha)\ \sigma} \qquad\qquad \textsc{m-New}\quad \frac{\Gamma, a, \alpha :: a \vdash M : \sigma \qquad \Gamma \vdash \forall(\alpha)\ \sigma[\alpha \leftarrow \varepsilon^{\alpha}]}{\Gamma \vdash \nu(a) M : \forall(\alpha)\ \sigma[\alpha \leftarrow \varepsilon^{\alpha}]}$$

$$\textsc{m-Fun}\quad \frac{\Gamma, x : \zeta_0 \vdash M : \zeta}{\Gamma \vdash \lambda(x) M : \forall(\gamma)\ (\zeta_0 \rightarrow \zeta)^{\gamma}} \qquad\qquad \textsc{m-App}\quad \frac{\Gamma \vdash M_1 : ((\zeta_2 \rightarrow \zeta) \approx \psi)^{\alpha} \qquad \Gamma \vdash M_2 : \zeta_2}{\Gamma \vdash M_1\ M_2 : \zeta}$$

$$\textsc{m-Let}\quad \frac{\Gamma \vdash M_1 : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash M_2 : \zeta_2}{\Gamma \vdash \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 : \zeta_2} \qquad\qquad \textsc{m-Ann}\quad \frac{\Gamma \vdash \forall(\mathsf{ftv}(\tau))\ \tau}{\Gamma \vdash (\tau) : \forall(\mathsf{ftv}(\tau))\ \wr \tau \rightarrow \tau \int}$$

$$\textsc{m-Witness}\quad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Eq} : \forall(\alpha, \gamma)\ \mathsf{eq}(\alpha, \alpha)^{\gamma}} \qquad\qquad \textsc{m-Use}\quad \frac{\Gamma \vdash (\mathsf{eq}(\tau_1, \tau_2))\ M_1 : \zeta_1 \qquad \Gamma, \tau_1 \doteq \tau_2 \vdash M_2 : \zeta_2}{\Gamma \vdash \mathsf{use}\ M_1 : \mathsf{eq}(\tau_1, \tau_2)\ \mathsf{in}\ M_2 : \zeta_2}$$

Fig. 3. Typing rules

also carry type equations $\tau_1 \doteq \tau_2$ in typing contexts that are used to show the coherence of ambivalent types via direct or indirect uses of well-formedness judgments.

All axioms require well-formedness of Γ so that whenever a judgement $\Gamma \vdash M : \sigma$ holds, we have $\vdash \Gamma$. Rule m-Inst instantiates the outermost variable of a type scheme. It is unusual in two ways. First, we write $\sigma[\alpha \leftarrow \psi_0^{\alpha}]$ rather than just σ in the quantified type. This trick ensures that all nodes labeled with α were indeed ψ_0^{α} and overcomes the absence of ψ_0 in the binder. Intuitively, the instantiated type should be $\sigma[\alpha \leftarrow \psi_0^{\alpha}][\alpha \leftarrow \psi^{\gamma}]$, but this happens to be equal to $\sigma[\alpha \leftarrow \psi^{\gamma}]$. Second, we require $\psi_0 \subseteq \psi$ to ensure preservation of ambivalence, as explained in the previous subsection. Finally, the premise $\Gamma \vdash \psi^{\gamma}$ ensures that the resulting type is well-formed.

Rule m-Gen introduces polymorphism implicitly, as in ML: variables that do not appear in the context can be generalized. The following rule is derivable from m-Gen and m-Inst, and can be used as a shortcut when variable α does not appear in ψ^{γ}:

$$\textsc{m-Bind}\quad \frac{\Gamma, \alpha :: \psi_1 \vdash M : \psi^{\gamma} \qquad \alpha \neq \gamma}{\Gamma \vdash M : \psi^{\gamma}}$$

Rule m-New enables explicit polymorphism (and explicit type equations using witnesses). For that purpose, it introduces a rigid type variable a in the typing context that may be used inside M—typically for introducing type annotations. However, polymorphism becomes implicit in the conclusion by turning the rigid type variable a into a quantified flexible type variable α when exiting the scope of the ν-form. Polymorphism can then be eliminated implicitly[5] as regular polymorphism in ML. The second premise ensures that the rigid type variable a does not appear anywhere else but in a^{α}.

Our version of Rule m-Fun generalizes the type γ introduced for annotating the arrow type, which avoids introducing $\gamma :: \zeta_0 \rightarrow \zeta$ in the premise. Rule m-App differs from the standard application rule in two ways: a minor difference is that the arrow type has a label as in Rule M-FUN; a major difference is that the type of M_1 may be ambivalent—as long as it contains an arrow (raw) type of the form $\zeta_2 \to \zeta$. In particular, the premise $\Gamma \vdash M_1 : ((\zeta_2 \to \zeta) \approx \psi)^\alpha$ does not, in general, imply $\Gamma \vdash M_1 : (\zeta_2 \to \zeta)^\alpha$, as this could lose sharing. Hence, we have to read the arrow structure directly from the ambivalent type. Rule M-LET is as usual.

---

[5] This is why we write this $\nu(a) M$ rather than $\Lambda a\, M$.

Rule M-ANN allows explicit loss of sharing via type annotations. It is presented as a retyping function of type scheme (τ), *i.e.* a function that changes the labeling of the type of its argument without changing its behavior. The types of the argument and the result need not be exactly τ but consistent instances of τ—see the definition of $\wr\tau\int$, above. Annotations are typically meant to be used in expressions such as $(\tau)\, M$, which forces M to have a type that is an instance of τ. While this is the only effect it would have in ML, here it also duplicates the polymorphic skeleton of M, which allows different labeling of *inner nodes* in the type of M passed to the annotation and its type after the annotation. By contrast, free type variables of τ remain shared between both types. The example below illustrates how type annotations can be used to remove ambivalence.

Rule M-WITNESS says that the Eq type constructor can be used to witness an equality between equal types as $\mathsf{eq}(\zeta, \zeta)^\gamma$, for any type ζ. Conversely, an equality type $\mathsf{eq}(\zeta_1, \zeta_2)^\gamma$, can only have been built from the Eq type constructor.

Rule M-USE uses this fact to learn and add the equation $\tau_1 \doteq \tau_2$ in the typing context while typechecking the body of M_2; the witness M_1 must be typable as an instance of the type $\mathsf{eq}(\tau_1, \tau_2)$ up to sharing of inner nodes. Since the equation is only available while typechecking M_2, it is not present in the typing context of the conclusion. Hence, the type ζ_2 must be well-formed in Γ. But this is a direct consequence of the second premise: it implies $\Gamma, \tau_1 \doteq \tau_2 \vdash \zeta_2$, which in turn requires that all labels of ζ_2 (which contain no quantifiers) have node descriptions in Γ, so that they cannot depend on $\tau_1 \doteq \tau_2$. Typically, ambivalent types needed for the typing of M_2 are introduced using rule M-BIND, which means that they cannot remain inside ζ_2, so that there is no way to keep an ambiguous type. Notice that the well-formedness of $\Gamma, \tau_1 \doteq \tau_2$ implies that τ_1 and τ_2 contain no flexible type variables (rules WF-TYPE-EQUAL and WF-CTX-EQUAL).

We now illustrate the typing rules on an example. Assume that $(\mathsf{if}\ \_\ \mathsf{then}\ \_\ \mathsf{else}\ \_)$ is given as a primitive with type scheme $\forall(\gamma_b, \gamma_2, \gamma_1, \gamma_0)\ \forall(\alpha)\ (\mathtt{bool}^{\gamma_b} \to (\alpha \to (\alpha \to \alpha)^{\gamma_2})^{\gamma_1})^{\gamma_0}$. Let Γ be $\Gamma_a, \Delta, \Delta', y : (\mathtt{int} \approx a)^\alpha$ where Γ_a is $a, a \doteq \mathtt{int}$ and Δ is $\alpha :: \mathtt{int}, \gamma_2 :: \alpha \to \alpha, \gamma_1 :: \alpha \to (\alpha \to \alpha)^{\gamma_2}$ and Δ' is $\gamma_b :: \mathtt{bool}, \gamma_0 :: \gamma_b \to (\alpha \to (\alpha \to \alpha)^{\gamma_2})^{\gamma_1}$. Using M-VAR for premises, we have:

$$\textsc{m-App}\ \frac{\Gamma \vdash \mathsf{if}\ \_\ \mathsf{then}\ \_\ \mathsf{else}\ \_ : (\mathtt{bool}^{\gamma_b} \to (\alpha \to (\alpha \to \alpha)^{\gamma_2})^{\gamma_1})^{\gamma_0} \qquad \Gamma \vdash \mathsf{true} : \gamma_b}{\Gamma \vdash \mathsf{if\ true\ then}\ \_\ \mathsf{else}\ \_ : (\alpha \to (\alpha \to \alpha)^{\gamma_2})^{\gamma_1}}$$

We also have $\Gamma \vdash 1 : (\mathtt{int} \approx a)^\alpha$ and $\Gamma \vdash y : (\mathtt{int} \approx a)^\alpha$ by M-INST and M-VAR. Hence, we have $\Gamma \vdash \mathsf{if\ true\ then\ 1\ else}\ y : (\mathtt{int} \approx a)^\alpha$ by M-APP. This leads to:

$$\textsc{m-Inst}\ \frac{\textsc{m-Fun}\ \dfrac{\Gamma \vdash \mathsf{if\ true\ then\ 1\ else}\ y : (\mathtt{int} \approx a)^\alpha}{\Gamma_a, \Delta, \Delta' \vdash \lambda(y)\, \mathsf{if\ true\ then\ 1\ else}\ y : \forall(\gamma)\ ((\mathtt{int} \approx a)^\alpha \to (\mathtt{int} \approx a)^\alpha)^\gamma}}{\textsc{m-Gen}\ \dfrac{\textsc{m-Bind}\ \dfrac{\Gamma_a, \Delta, \Delta' \vdash M : ((\mathtt{int} \approx a)^\alpha \to (\mathtt{int} \approx a)^\alpha)^{\gamma_2}}{\Gamma_a, \Delta \vdash M : ((\mathtt{int} \approx a)^\alpha \to (\mathtt{int} \approx a)^\alpha)^{\gamma_2}}}{\Gamma_a \vdash M : \forall(\alpha, \gamma)\ ((\mathtt{int} \approx a)^\alpha \to (\mathtt{int} \approx a)^\alpha)^\gamma}}$$

where M is $\lambda(y)\,\mathsf{if\ true\ then}\ 1\ \mathsf{else}\ y$. Rule m-Bind is used for variables γ_b and γ_0 in Δ' that are no longer used (we omitted the other premisses), while Rule m-Gen is used for variables α and γ_2 in Δ. Notice that neither $\Gamma_a \vdash M : \forall(\alpha,\alpha',\gamma)\ ((\mathtt{int} \approx a)^{\alpha} \to \mathtt{int} \approx a^{\alpha'})^{\gamma}$ nor $\Gamma_a \vdash M : \forall(\alpha,\gamma)\ (\mathtt{int}^{\alpha} \to \mathtt{int}^{\alpha})^{\gamma}$ are derivable. It is a key feature of our system that sharing and ambivalence can only be increased *implicitly*. Still, it is sound to decrease them *explicitly*, using a type annotation, as in $\Gamma_a \vdash (a \to \mathtt{int})\ M : \forall(\alpha,\alpha',\gamma)\ (a^{\alpha} \to \mathtt{int}^{\alpha'})^{\gamma}$. This is obtained by applying the coercion $(a \to \mathtt{int})$ of type $\wr(a \to \mathtt{int}) \to (a \to \mathtt{int})\int$, *i.e.*

$$\forall(\alpha_1,\alpha_2,\alpha_1',\alpha_2',\gamma,\gamma',\gamma_0)\ \big((a^{\alpha_1} \to \mathtt{int}^{\alpha_2})^{\gamma} \to (a^{\alpha_1'} \to \mathtt{int}^{\alpha_2'})^{\gamma'}\big)^{\gamma_0}$$

to M. The expression M_0 equal to $\mathsf{use}\ x : \mathsf{eq}(a,\mathtt{int})\ \mathsf{in}\ (a \to \mathtt{int})\ M$ is not ambiguous thanks to the annotation around M. Hence, we have:

$$\textsc{m-App*}\ \dfrac{\textsc{m-New}\ \dfrac{\textsc{m-Fun*}\ \dfrac{\textsc{m-Use*}\ \dfrac{\Gamma' \vdash (\mathsf{eq}(a,\mathtt{int}))\,x : \zeta_1 \qquad \Gamma', a \doteq \mathtt{int} \vdash (a \to \mathsf{int})\,M : \wr a \to \mathtt{int}\int}{\Delta'', a, \Delta''', x : \mathsf{eq}(a^{\gamma_1}, \mathtt{int}^{\gamma_2})^{\gamma} \vdash M_0 : \wr a \to \mathtt{int}\int}}{\Delta'', a, \alpha :: a \vdash \lambda(x)\,M_0 : \wr \mathsf{eq}(a,\mathtt{int}) \to a \to \mathtt{int}\int}}{\Delta'' \vdash \nu(a)\,\lambda(x)\,M_0 : \forall(\alpha)\ \wr \mathsf{eq}(\alpha,\mathtt{int}) \to \alpha \to \mathtt{int}\int} \quad \Delta'' \vdash \mathsf{Eq} : \ldots}{\vdash (\nu(a)\,\lambda(x)\,M_0)\ \mathsf{Eq} : \wr \mathtt{int} \to \mathtt{int}\int}$$

for some well-chosen Δ'', Δ''' and Γ', where $R*$ means R preceded and followed by a sequence of m-Inst, m-Bind, and m-Gen. The rigid variable a is turned into the polymorphic variable α which is then instantiated to $\mathtt{int}^{\alpha}$ before the application to Eq.

4 Properties

By lack of space, we omit formal statements and their proofs, as well as a description of type inference, and we refer the reader to the accompanying technical report [5].

Type soundness Type soundness is established by seeing our system as a subset of $\mathsf{HMG(X)}$ [12]. Formally, we exhibit a translation from our language to $\mathsf{HMG(X)}$ that preserves typing judgments. The key is that well-formed ambivalent types are such that all simple types in their interpretation are provably equal in the current context, *i.e.* under the equations introduced by use expressions. Ambivalent types are only used for type inference and are dropped during the translation.

Monotonicity Let $\Gamma \vdash \sigma' \prec \sigma$ be the instantiation relation, which says that any monomorphic instance of σ well-formed in Γ is also a monomorphic instance of σ'. This relation is extended point-wise to typing contexts: $\Gamma' \prec \Gamma$ if for any term variable x in $\mathsf{dom}(\Gamma)$, $\Gamma \vdash \Gamma'(x) \prec \Gamma(x)$, all other components of Γ and Γ' being identical. We may now state monotonicity: in our system, if $\Gamma \vdash M : \zeta$ and $\Gamma' \prec \Gamma$, then $\Gamma' \vdash M : \zeta$.

Existence of principal solutions to type inference problems This is our main result. A typing problem is a typing judgment skeleton $\Gamma \triangleright M : \zeta$, where Γ omits all node descriptions $\alpha :: \psi$ (hence, Γ is usually not well-formed, but can be extended into a well-formed environment by interleaving the appropriate node descriptions with bindings in Γ). A solution to a typing problem is a pair of a substitution θ that preserves ambivalence for the types in Γ and ζ, together with a context Δ that contains only node descriptions, such that $\Gamma\theta$ and Δ can be interleaved to produce a well-formed typing context, written $\Gamma\theta \mid \Delta$, and the judgment $\Gamma\theta \mid \Delta \vdash M : \zeta\theta$ holds.

For any typing problem, the set of solutions is stable by substitution and is either empty or has a principal solution (Δ, θ), *i.e.* one such that any other solution (Δ', θ') is of the form $\theta' = \theta'' \circ \theta$ for some substitution θ'' that preserves well-formedness in $\Gamma\theta \mid \Delta$, *i.e.* for any type ζ' such that $\Gamma\theta \mid \Delta \vdash \zeta'$, we have $\Gamma\theta' \mid \Delta' \vdash \zeta'\theta''$.

Sound and complete type inference Principality of type inference is proved as usual by exhibiting a concrete type inference algorithm. This algorithm (presented in the extended version) relies on a variant of the standard unification algorithm that works on ambivalent types and preserves their sharing. It uses a typing constraint approach, which converts typing problems to unification problems, while also ensuring that inferred types are well-formed, *i.e.* coherent, properly scoped, and acyclic. The use of constraints here is however just a convenience: since the ambivalence information is contained in types themselves, constraints can always be solved prior to type generalization so that we do not need constrained type schemes. That is, constraints are just a way to describe the algorithmic steps without getting into implementation details: OCaml itself uses a variant of Milner's algorithm $\mathscr{J}$ [**?**].

5 Related works

While GADTs have been an active research area for about 10 years, early works usually focused on their type checking and expressiveness, ignoring ML-style type inference. Typically, they rely on an explicitly typed core language and use local type inference techniques to leave some type information implicit. Other recent works with rich dependent type systems also fit in this category and are only loosely related to ours.

Relatively few papers are dedicated to *principal* type inference for GADTs. The tension between ambiguity and principality is so strong that it has been assumed that the only way to reach principality is to know exactly the external type of each GADT match case. As a result, research has not been so much focused on finding a type system with principal types, but rather on clever propagation of type information so that programs have enough type annotations after propagation to admit principal types—or are rejected otherwise. Hence, some existing approaches always return principal solutions, but do not have a clear specification of when they will succeed, because this depends on the propagation algorithm (or some idealized version of it) which does not have a compositional specification.

OutsideIn improves on this by using uses constraint solving in place of directional annotation propagation, which greatly reduces the need for annotations. Stratified type inference [9] is another interesting approach to type inference for GADTs that uses several sophisticated passes to propagate local typing constraints (and not just type annotations) progressively to the rest of the program.

The following table summarizes the typability of the programs given in the overview, for our approach (including simple syntactic propagation of type annotations), OutsideIn[6], and stratified type inference [9].

Program	f	$\mathtt{f}_1$	$\mathtt{f}_2$	g	$\mathtt{g}_1$	$\mathtt{g}_2$	p	$\mathtt{p}_1$
Ambivalent	√	√	√	−	√	√	√	−
OutsideIn	−	−	−	−	−	√	√	√
Stratified	−	√	√	−	−	√	−	−

The results for `f` are unsurprising: this example is not even principal in the naive type system: without an internal notion of ambivalence, a type system is unable to tell that the equality between two types is only accidental and should not be considered as a source of ambiguity. The results for $\mathtt{f}_1$ and $\mathtt{f}_2$ are more interesting. While OutsideIn requires an external type annotation in both cases, stratified type inference accepts to infer the type of the branch from its body. More precisely, the propagation algorithm operates in a bi-directional way and is able to extract non-ambiguous information from GADT pattern-matching branches. The exported information is pruned so that it remains compatible with any interpretation of the internal information, even in a context with fewer type equations. Thus, the type of the result is pruned in function `f`, but it can be propagated for $\mathtt{f}_1$ and $\mathtt{f}_2$. This corresponds exactly to the naive notion of ambiguity.

Typing of `g` fails in all three systems, as it is fundamentally ambiguous, whichever definition is chosen. The results for $\mathtt{g}_1$ may look surprising: while it contains many type annotations, both OutsideIn and stratified type inference still fail on it. The reason is that type annotations are inside the branch: in both systems, only type annotations outside of a branch can disambiguate types for which an equation has been introduced. We find this behavior counter-intuitive. The freedom of where to add type annotations stands as a clear advantage of ambivalent types. By contrast, $\mathtt{g}_2$ provides full type annotations in a standard style, so that all systems succeed—although ambivalent types need some (simple) propagation mechanism to push annotations inside.

Programs `p` and $\mathtt{p}_1$ demonstrate the power of OutsideIn. The program $\mathtt{p}_1$ is the following variant of `p`, which we deem ambiguous:

```
let p1 (type a) (x : (a,int) eq) (y : a) =
  let z = (match x with Eq -> if y>0 then y else 0) in z + 1
```

Indeed, the match expression in $\mathtt{p}_1$ would have to be given the ambivalent type $a \approx \mathtt{int}$, which is not allowed outside the scope of the equation $a = \mathtt{int}$. Both `p` and $\mathtt{p}_1$ are accepted by OutsideIn, since type information can be propagated upward, even for local let definitions. This comes at a cost, though: local let-definitions are monomorphic by default (but can be made polymorphic by adding a type annotation). While local polymorphic definitions are relatively rare, so that this change of behavior appears as a good compromise for Haskell, they are still frequent enough, and their coresponding type annotations large enough, so that we prefer to keep local polymorphism in OCaml [**?**]. Moreover, local polymorphism is critical for the annotation propagation mechanism used by OCaml as a complement to ambivalent types and for polymorphic methods.

---

[6] Results differ for GHC 7.6, as it slightly departs from OutsideIn allowing some biased choices, but next versions of GHC should strictly comply with OutsideIn.

All examples above are specifically chosen to illustrate the mechanisms underlying ambivalence and do not cover all uses of GADTs. Thus, they do not mean that our approach always outperforms other ones, but they emphasize the relevance of ambivalence. The question is not whichever approach taken alone performs better, but rather how ambivalence can be used to improve type inference with GADTs. Indeed, ambivalence could be added to other existing approaches to improve them as well.

Besides this comparison on examples, the main advantage of ambivalent types is to preserve principal type inference and monotonicity, so that type inference and program refactoring are less surprising.

An interesting proposal by Lin and Sheard [8], called point-wise type inference, is also tackling type inference *à la* ML, but restricting the expressiveness of the system—some uses of GADT will be rejected—so that more aggressive type propagation can be done in a principal way. Point-wise type inference is hard to compare to our approach, as many programs have to be modified. For instance, it rejects all our examples, because equality witnesses can only be matched on if they relate two rigid type variables. To be accepted, we could replace `eq` by a specialized version, `type _ t = Int : int t`.

Ambivalent types borrow ideas from earlier works. The use of sharing to track known type information was already present in our work on semi-explicit first-class polymorphism [4]. There, we only tracked sharing on a special category of nodes containing explicitly polymorphic types. Here, we need to track sharing on all nodes, as any type can become ambivalent. In our type inference algorithm, we also reuse the same definition style, describing type inference as a constraint resolution process, but introducing some points where constraints have to be solved before continuing.

The formalization itself borrows a lot from previous work on *structural polymorphism* for polymorphic variant and record types [3]. In particular, unification of ambivalent types, which merges sets of rigid variables and requires checking coherence constraints, can be seen as an instance of the unification of structurally polymorphic nodes. The difference is again that all nodes are potentially ambivalent in our case, while structural polymorphism only cares about variant and record types.

6 Conclusion

Ambivalent types are a refinement of ML types, which represents within types themselves ambiguities resulting from the use of local equations. They permit a more accurate definition of ambiguity, which in turn reduces the need for type annotations while preserving both the principal type property and monotonicity.

This approach has been implemented in OCaml. We have not addressed propagation of type information in this work, although this is quite useful in practice. A simple propagation mechanism based on polymorphism, similar to that used for semi-explicit first-class polymorphism, as already in use in OCaml, seems sufficient to alleviate the need for most local type annotations, while preserving principality of type inference.

The notion of ambivalence is orthogonal to previous techniques used for GADT type inference. Therefore, it should also benefit other approaches such as OutsideIn or stratified type inference. Hopefully, ambivalent types might be transferable to ML^F

[6], as the techniques underlying both ambivalent types and semi-explicit first-class polymorphism have many similarities.

Acknowledgements We thank Gabriel Scherer and the anonymous reviewers for detailed comments on this paper.

References

1. A. I. Baars and S. D. Swierstra. Typing dynamic typing. In *ICFP '02: Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming*, pages 157–166. ACM Press, 2002.
2. J. Cheney and R. Hinze. First-class phantom types. Computer and Information Science Technical Report TR2003-1901, Cornell University, 2003.
3. J. Garrigue. A certified implementation of ML with structural polymorphism. In *Proc. Asian Symposium on Programming Languages and Systems*, volume 6461 of *Springer-Verlag LNCS*, pages 360–375, Shanghai, Nov. 2010.
4. J. Garrigue. Monomophic let in OCaml? Blog article at: `http://gallium.inria.fr/blog/monomorphic_let/`, Sept. 2013.
5. J. Garrigue and D. Rémy. Semi-explicit first-class polymorphism for ML. *Information and Computation*, 155:134–171, Dec. 1999.
6. J. Garrigue and D. Rémy. Ambivalent types for principal type inference with GADTs. Available electronically at `http://gallium.inria.fr/~remy/gadts/`, June 2013.
7. D. Le Botlan and D. Rémy. Recasting MLF. *Information and Computation*, 207(6):726–785, 2009.
8. X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The OCaml system release 4.00, Documentation and user's manual*. Projet Gallium, INRIA, July 2012.
9. C.-k. Lin and T. Sheard. Pointwise generalized algebraic data types. In *Proceedings of the 5th ACM SIGPLAN workshop on Types in Language Design and Implementation*, TLDI '10, pages 51–62, New York, NY, USA, 2010. ACM.
10. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
11. F. Pottier and Y. Régis-Gianas. Stratified type inference for generalized algebraic data types. In *Proceedings of the 33rd ACM Symposium on Principles of Programming Languages (POPL'06)*, pages 232–244, Charleston, South Carolina, Jan. 2006.
12. T. Schrijvers, S. Peyton Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for gadts. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 341–352, New York, NY, USA, 2009. ACM.
13. T. Sheard and N. Linger. Programming in Omega. In Z. Horváth, R. Plasmeijer, A. Soós, and V. Zsók, editors, *Central European Functional Programming School*, volume 5161 of *Lecture Notes in Computer Science*, pages 158–227. Springer, 2007.
14. V. Simonet and F. Pottier. A constraint-based approach to guarded algebraic data types. *ACM Transactions on Programming Languages and Systems*, 29(1), Jan. 2007.
15. 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.
16. H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, pages 224–235, New York, NY, USA, 2003. ACM.