Ambivalent Types for Principal Type Inference with GADTs (extended version)

Jacques Garrigue, Didier Rémy

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 [**?**,**?**,**?**]. They can express many useful invariants of data-structures, provide safer typing, and allow for more polymorphism [**?**]. 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 [**?**]. 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 [**?**] 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. Still, the external type system is not compositional because of the elaboration phase. Moreover, 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* [**?**], 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 [**?**], 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 [**?**].

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 [**?**].

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 [**?**]. Finally, we compare with related works in §5.

2 An overview of our solution

2.1 Using ambivalence for refining the definition of ambiguity

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 $a = \texttt{int}$ 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 $(\alpha, \texttt{int})\ \texttt{eq} \to \texttt{int}$ or $(\alpha, \texttt{int})\ \texttt{eq} \to \alpha$. 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 $\texttt{f}_1$, we just return `true`, which has the type `bool`, unrelated to the equation. In $\texttt{f}_2$, 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

[3] Examples in this section use OCaml syntax [**?**]. 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.

only possible type is `int`. It looks even more innocuous than $\mathtt{f}_2$, 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):

```
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. Intuitively, we may do so by disallowing the expression to have either type and instead viewing it with an ambivalent type $a \approx \mathtt{int}$, 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 $a \approx \mathtt{int}$ 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 $a \approx \mathtt{int}$, 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 $\mathtt{g}_1$ may just be written:

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

or, using the OCaml syntax for explicitly polymorphic types:

```
let g2 : type a. (a,int) eq -> a -> a = fun x y ->
```

```
match x with Eq -> if y > 0 then y else 0
```

However, we will ignore this aspect in this work.

2.2 Avoiding aggressive propagation of type annotations

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 (as in either one of the last two versions) and still get the full type information. However, the situation degrades with local `let` bindings. For example, consider the function `p` below:

```
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). Some type annotations are still required, since the type of the (GADT part of the) scrutinee must always be provided. directly or indirectly. For example, the following variant `h` of f_2 without type annotations fails:

```
||| let h (type a) x (y : a) = match x with Eq -> (y > 0)
```

In OCaml, in the absence of annotation on the GADT part of a scrutinee, we do not immediately fail, but fall back to typechecking as an ordinary datatype, *i.e.* without enriching the context with equality constraints. Hence, removing all type information from the previous example, we actually succeed, but with type (α, α) `eq-> int -> bool`.

```
||| let h1 x y = match x with Eq -> (y > 0)
```

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 $\forall(\alpha)\ \mathsf{eq}(\alpha,\alpha)$ and existential types. The equivalence of expressiveness between the general case and this special case has been studied from a semantic point of view [**?**]; it also holds from a type inference perspective (it is easy to see that they generate similar type inference constraints [**?**]). However, we also omit existential types here, as this is an orthogonal issue, which does not raise any problem for type inference. 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.

Simple types. 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. 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.* a type annotation behaves as a coercion function and the usual annotation $(M : \tau)$ is seen as the application $(\tau)\,M$:

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

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 $\zeta \to \zeta$, an equality type $\mathsf{eq}(\zeta,\zeta)$, or the base type $\mathtt{int}$. 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 $(\psi \approx \psi) = \psi$. For example, $\mathtt{int} \approx \mathtt{int}$ is the same as $\mathtt{int}$. 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 $a \doteq \texttt{int}$, and a variable x of type a, under which we apply a function `choice` of type $\alpha \to \alpha \to \alpha$ to x and 1. We first reason in the absence of labels on inner nodes. The partial application `choice` x has type $a \to a$. 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 $\texttt{int} \approx a$. 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 $\texttt{int} \approx a$, we may see both x and 1 with type $\texttt{int} \approx a$ before performing the applications. In this case, we get yet another result $\texttt{int} \approx a$, 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 preserve this information. Revisiting the example, the partial application now has type $a^\alpha \to a^\alpha$ (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 $a \to a$ but remembering that the domain and codomain must be kept synchronized. Then, the integer 1 of type $\texttt{int}^\gamma$ can also be seen with type $(\texttt{int} \approx a)^\gamma$ and unified with the domain of the function a^α, with the effect of replacing all occurrences of a^α and of $\texttt{int}^\gamma$ by $(\texttt{int} \approx a)^\alpha$. Thus, the function has type $(\texttt{int} \approx a)^\alpha \to (\texttt{int} \approx a)^\alpha$ and the result of the application has type $(\texttt{int} \approx a)^\alpha$—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 $a' \doteq \texttt{int}$ or $a' \doteq a$ in the context, and refine its type to $(\texttt{int} \approx a \approx a')^\alpha$. 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 $\forall(\alpha, \gamma, \gamma')\ (\alpha \to (\alpha \to \alpha)^\gamma)^{\gamma'}$ and its partial application to `x` has type $\forall(\alpha, \gamma)\ (a^\alpha \to a^\alpha)^\gamma$ after generalization. Observe that this is less general than $\forall(\alpha, \alpha', \gamma)\ (a^\alpha \to a^{\alpha'})^\gamma$ but more general than $\forall(\alpha, \gamma)\ ((\texttt{int} \approx a)^\alpha \to (\texttt{int} \approx a)^\alpha)^\gamma$.

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:

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

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 con-

structor 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:

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

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* $\alpha :: \psi$:

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

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

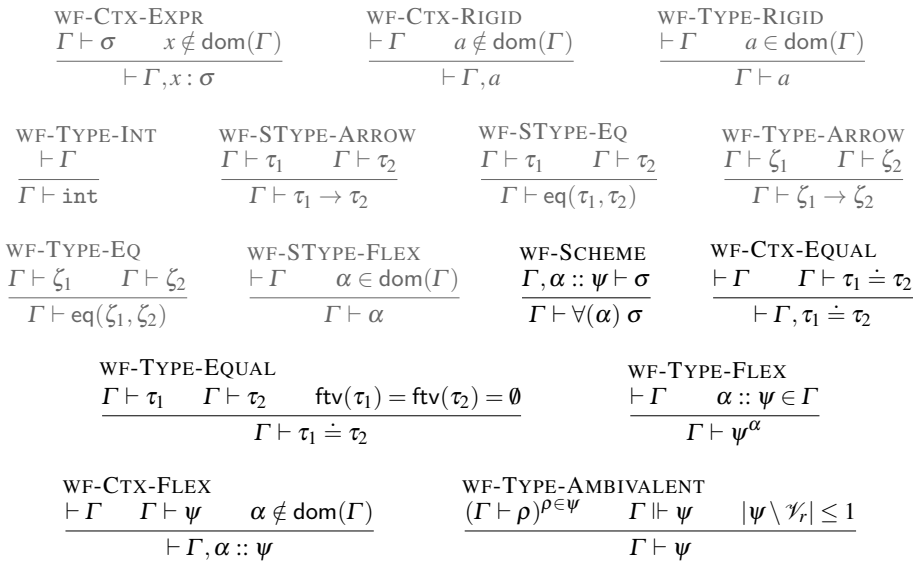

Fig. 1. Well-formedness of contexts and types

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 $(\mathtt{int} \approx a)^{\gamma} \to (\mathtt{int} \approx a)^{\gamma}$.

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 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.

Notice that the entailment depends on the equivalence relation defined by the equations in Γ and not on the particular set of equations that generates this equivalence. Hence, replacing equations in Γ by another set of equivalent equations or, in particular, adding an equation that is already a consequence of equations in Γ does not change entailment.

For instance, the following program is well-typed. because the inner equation learn on the inner match is a consequence of the equation learned on the outer match, hence z is not ambivalent on the inner match and only the outer match requires an annotation.

```
let q (type a) (type b)
        (x0 : (b * b, a * int) eq) (x2 : (a, int) eq) (y : a) =
  match x0 with Eq ->
    let z = match x2 with Eq -> if y > 0 then y else 0
    in (z : a)
```

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 $\psi\theta \subseteq \lfloor(\psi^\alpha)\theta\rfloor$.*

[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.

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

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

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

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

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

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

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

Fig. 3. Typing rules

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 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 judgment $\Gamma \vdash M : \sigma$ 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:

$$\text{M-BIND}\quad \frac{\Gamma, \alpha :: \psi_1 \vdash M : \psi^\gamma \qquad \alpha \neq \gamma}{\Gamma \vdash M : \psi^\gamma}$$

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. We could also have used this other equivalent but less readable rule:

$$\frac{\Gamma, \gamma :: \zeta_0 \to \zeta, x : \zeta_0 \vdash M : \zeta}{\Gamma \vdash \lambda(x) M : (\zeta_0 \to \zeta)^\gamma}$$

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. Since well-formedness ensures that there is at most one arrow type in an ambivalent type (Rule WF-TYPE-AMBIVALENT), ψ can only be composed of rigid type variables. So the application rule remains deterministic. Rule M-LET is as usual.

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 $(\tau)\, M$, also written $(M : \tau)$, 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

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

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).

Example

We now illustrate the typing rules through 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 the following derivation:

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

where M is $\lambda(y)\,\mathsf{if\ true\ then\ 1\ else}\ y$. Rule M-BIND is used for variables γ_b and γ_0 in Δ' that are no longer used (we omitted the other premises), 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 $\Gamma_a \vdash (a \to \mathtt{int})\ M : \forall(\alpha,\alpha',\gamma)\ (a^\alpha \to \mathtt{int}^{\alpha'})^\gamma$.

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. Indeed, $\mathsf{use}\ x : \mathsf{eq}(a,\mathtt{int})\ \mathsf{in}\ M$ would be rejected, since an instance of the type of M always will contain an ambivalent type of the from $(\mathtt{int} \approx a \approx \ldots)^\alpha$ which cannot be well-formed in $a, a \doteq \mathtt{int}$. Hence, we have:

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

$$\langle \forall(\bar\alpha)\ \zeta \rangle = \forall(\bar\alpha)\ \langle\zeta\rangle \qquad \langle \mathtt{int} \approx \bar a^\alpha \rangle = \mathtt{int}$$
$$\langle \varepsilon^\alpha \rangle = \alpha \qquad \langle \zeta_1 \to \zeta_2 \approx \bar a^\alpha \rangle = \langle\zeta_1\rangle \to \langle\zeta_2\rangle$$
$$\langle \bar a^\alpha \rangle = \min \bar a \qquad \langle \mathsf{eq}(\zeta_1,\zeta_2) \approx \bar a^\alpha \rangle = \mathsf{eq}(\langle\zeta_1\rangle, \langle\zeta_2\rangle)$$

Fig. 4. Canonical types

$$\langle x \rangle = x \qquad \langle \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 \rangle = \mathsf{let}\ x = \langle M_1\rangle\ \mathsf{in}\ \langle M_2\rangle$$
$$\langle c \rangle = c \qquad \langle \mathsf{use}\ M_1 : \tau \doteq \tau\ \mathsf{in}\ M_2 \rangle = (\lambda(\mathsf{Eq})\,\langle M_2\rangle)\ \langle M_1\rangle$$
$$\langle M_1\ M_2 \rangle = \langle M_1\rangle\ \langle M_2\rangle \qquad \langle \nu(a)\,M \rangle = \langle M\rangle$$
$$\langle \lambda(x)\,M \rangle = \lambda(x)\,\langle M\rangle \qquad \langle (\tau) \rangle = \lambda(x)\,x$$
$$\langle \mathsf{Eq} \rangle = \mathsf{Eq}$$

Fig. 5. Elaboration of terms

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.

3.6 Type soundness

Type soundness is established by seeing our system as a subset of HMG(X) [**?**].

For this purpose, we exhibit a translation from our language to HMG(X) that preserves well-typedness. Types and typing contexts are translated as well so that the translation of typing judgments in our language is a valid typing judgment in HMG(X).

The translation emphasizes an interesting aspects of our system: ambivalent types are only a gadget for type inference and can be dropped in the translation. The key is that well-formed ambivalent types are such that all simple types in their interpretation are provably equivalent in the current context, *i.e.* under the equality assumptions that have been introduced by use expressions.

The type system HMG(X) is designed for expressiveness rather than type inference and expressions contain no explicit type information at all. Moreover, the language is quite rich, and assume arbitrary GADT type-definitions while we only use one predefined datatype Eq with a single data-constructor Eq of type $\forall(\alpha)\ \alpha \to \alpha \to \mathsf{eq}(\alpha,\alpha)$. In particular, the expression $\mathsf{use}\ M_1 : \mathsf{eq}(\tau_1,\tau_2)\ \mathsf{in}\ M_2$ that destructs a GADT stands for the immediate application $(\lambda(\mathsf{Eq})M_2)\ M_1$ in HMG(X). Notice that we only use HMG(X) with equality constraints—and with general subtyping constraints.

The translation of ambivalent types into simple types is given in Figure 4. The translation $\langle\zeta\rangle$ of a well-formed type ζ always picks a simple type that is in the interpretation $[\![\zeta]\!]$ of ζ. To simplify, we make the translation deterministic. For that purpose, we assume given an ordering of rigid variables (which may coincide with the order in which variables are introduced in the context). The translation is only defined for well-formed types, for which every raw type contains at most one type that is not a rigid variable, *i.e.* a structured type. Leaves ε^α are translated into regular type variables; Inner nodes $\rho \approx \psi^\alpha$ are recursively translated into $\langle\rho\rangle$, where ρ is a structure type or a rigid type variable smaller than all rigid type variables of ψ.

The translation of contexts is the point-wise translation of bindings, except for variable bindings (both rigid-variables and node descriptions), which are simply dropped.

The translation of terms is given on Figure 5. It is an homorphism, except for two things: on the one hand, it drops all type information; on the other hand, it translates use $M_1 : \tau_1 \doteq \tau_2$ in M_2 into the immediate application $(\lambda(\mathsf{Eq})\, M_2)\, M_1$ as described above.

Equality constraints and our two constructs Use and Eq are just a particular instance of a GADT, with a single data-constructor Eq of type $\forall(\alpha)\ \alpha \to \alpha \to \mathsf{eq}(\alpha, \alpha)$.

Theorem 1. *If* $\Gamma \vdash M : \sigma$ *then* $\langle\Gamma\rangle \vdash \langle M\rangle : \langle\sigma\rangle$.

The proof is by structural induction on the derivation of the typing judgment in our system. The key is that well-typedness implies well-formedness of ambivalent types, which implies that all types in the semantics of a well-formed ambivalent types are provably equal under the equalities in the typing context. This is used in particular in the translation of instantiation, which changes ambivalence in our system and relies on constraint entailment in HMG(X).

4 Properties

In this section we will prove the following properties.

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. Although further developments in this section will use the split form, described below, we first formulate this important result in the mixed form used up to now. 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 other 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 subsection 4.5) 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 $\mathscr{J}$ [**?**].

4.1 Split view

The grammar of types we have presented so far is quite convenient for the user to read or write types, since it is presented as a usual tree-structure with additional label decorations on every node. However, while close to our usual representation of types, there is redundancy: the description of nodes is repeated on every node but also in the typing context. While this eases the reading of types, it makes them harder to manipulate technical developments and proofs.

To solve this tension, we introduce an alternative view, call the *split view*, which is strictly equivalent to the mixed view, but avoids redundancy by representing internal nodes only by their labels. To avoid confusion, we will refer to the previous definition as types in the *mixed view*.

Types The grammar of ambivalent types ς in split form is defined as:

$$\begin{array}{ll} \rho ::= a \mid \alpha \to \alpha \mid \mathsf{eq}(\alpha,\alpha) \mid \mathtt{int} & \Delta ::= \emptyset \mid \Delta, \alpha :: \psi \\ \psi ::= \varepsilon \mid \rho \approx \psi & \varsigma ::= \Delta \triangleright \alpha \end{array}$$

We reuse the same non-terminals for raw types ρ—as both forms are have the same meaning, indeed. However, we write ς instead of ζ for types in split form so as to avoid the confusion. By contrast with mixed types ζ, the syntactic definition of split types ς is not recursive, as we just write a labels α instead of nodes ψ^α and recover ψ from α using a *label context* Δ mapping labels to their content ψ. As a result, an ambivalent type is now a pair $\Delta \triangleright \alpha$ of a type context and a flexible type variable. In practice, however, the label context is often part of a larger context, which is either left implicit or fixed explicitly.

The mixed form of $\Delta \triangleright \alpha$ can be obtained by starting from α and recursively recovering contents from Δ. For instance, the split form $\alpha :: \mathtt{int}, \alpha_0 :: \alpha \to \alpha \triangleright \alpha_0$ corresponds to the mixed form $(\mathtt{int}^\alpha \to \mathtt{int}^\alpha)^{\alpha_0}$. Here is the formal translation:

$$\begin{array}{ll} \lceil \Delta \triangleright \alpha \rceil = \lceil \alpha \rceil_\Delta & \lceil a \rceil_\Delta = a \\ \lceil \alpha \rceil_\Delta = (\lceil \Delta(\alpha) \rceil_\Delta)^\alpha & \lceil \alpha_1 \to \alpha_2 \rceil_\Delta = \lceil \alpha_1 \rceil_\Delta \to \lceil \alpha_2 \rceil_\Delta \\ \lceil \varepsilon \rceil_\Delta = \varepsilon & \lceil \mathsf{eq}(\alpha_1,\alpha_2) \rceil_\Delta = \mathsf{eq}(\lceil \alpha_1 \rceil_\Delta, \lceil \alpha_2 \rceil_\Delta) \\ \lceil \rho \approx \psi \rceil_\Delta = \lceil \rho \rceil_\Delta \approx \lceil \psi \rceil_\Delta & \lceil \mathtt{int} \rceil_\Delta = \mathtt{int} \end{array}$$

Well-formed types have both a mixed form and a split form. Let ζ be a well-formed type in mixed form. Then, the split form of ζ is $\Delta \triangleright \alpha$ where Δ is mapping every flexible variable α of ζ to the unique set ψ in ζ that α labels. Reciprocally, a well-formed type $\Delta \triangleright \alpha$ in split form is such that $\alpha \in \mathsf{dom}(\Delta)$, and there is a strict partial order $\prec_\Delta$ on $\mathsf{dom}(\Delta)$ such that for each $\gamma \in \mathsf{dom}(\Delta)$, the variables of $\Delta(\gamma)$ precede γ. This guarantees that the above translation function is well-defined and terminates.

Given a type ψ^α in mixed form, we write $|\psi^\alpha|$ for the *minimal context* of ψ^α, *i.e.* the smallest well-formed Δ such that $\lceil \alpha \rceil_\Delta = \psi^\alpha$.

Free variables Free variables of a type in mixed form are defined as follows:

$$\begin{aligned}
\mathsf{ffv}(\Delta \triangleright \alpha) &= \mathsf{ffv}_\Delta(\alpha) & \mathsf{ffv}_\Delta(a) &= \emptyset \\
\mathsf{ffv}_\Delta(\alpha) &= \{\alpha\} \cup \mathsf{ffv}_\Delta(\Delta(\alpha)) & \mathsf{ffv}_\Delta(\mathtt{int}) &= \emptyset \\
\mathsf{ffv}_\Delta(\varepsilon) &= \emptyset & \mathsf{ffv}_\Delta(\alpha_1 \to \alpha_2) &= \mathsf{ffv}_\Delta(\alpha_1) \cup \mathsf{ffv}_\Delta(\alpha_2) \\
\mathsf{ffv}_\Delta(\rho \approx \psi) &= \mathsf{ffv}_\Delta(\rho) \cup \mathsf{ffv}_\Delta(\psi) & \mathsf{ffv}_\Delta(\mathsf{eq}(\alpha_1, \alpha_2)) &= \mathsf{ffv}_\Delta(\alpha_1) \cup \mathsf{ffv}_\Delta(\alpha_2)
\end{aligned}$$

Definitions for $\mathsf{frv}_\Delta(\alpha)$ are similar, with $\mathsf{frv}_\Delta(\alpha)$ and $\mathsf{frv}_\Delta(a)$ becoming equal to $\mathsf{frv}_\Delta(\Delta(\alpha))$ and $\{a\}$, respectively.

Type schemes In the split view, we need to include a label context inside type schemes, since the description of nodes is not inlined anymore: The translation from split form to mixed form is then extended to type schemes.

$$\sigma ::= \forall(\Delta)\ \alpha \qquad\qquad \lceil \forall(\Delta_1)\ \alpha \rceil_\Delta = \forall(\mathsf{dom}(\Delta_1))\ \lceil \alpha \rceil_{\Delta,\Delta_1}$$

For instance, $\forall(\gamma_b,\gamma_1,\gamma_b,\gamma_0)\ \forall(\alpha)\ (\mathtt{bool}^{\gamma_b} \to (\alpha \to (\alpha \to \alpha)^{\gamma_2})^{\gamma_1})^{\gamma}$ becomes $\forall(\Delta)\ \gamma$ where Δ is $\alpha, \gamma_b :: \mathtt{bool}, \gamma_2 :: \alpha \to \alpha, \gamma_1 :: \alpha \to \gamma_2, \gamma_0 :: \mathtt{bool}^{\gamma_b} \to \gamma_1$.

Substitution In the split view, node descriptions must always preexist in the typing context (during type inference, the typing context will be extended as needed with new node descriptions). Thus, it is sufficient for substitutions to substitute variables for variables—and extend these structurally to substitutions on raw types,

Preservation of ambivalence is slightly generalized:

Definition 3. *A variable substitution θ* preserves ambivalence between label contexts *Δ and Δ', written $\Delta \vdash \theta : \Delta'$, if and only if, for any $\alpha :: \psi$ in Δ, $\theta(\alpha') :: \psi'$ is in Δ' and $\theta(\psi) \subseteq \psi'$.*

The two definitions coincide in the following way: the substitution θ *preserves ambivalence in a type* ζ if and only if $|\zeta| \vdash \hat{\theta} : |\zeta\theta|$, where $\hat{\theta}(\alpha)$ is either α' if $\alpha \in \mathsf{dom}(\theta)$ and $\theta(\alpha) = \psi^{\alpha'}$ or α otherwise.

Typing rules Typing rules in split form are given in figure 6. Most of them can be obtained by replacing mixed types of the form $\zeta = \psi^\gamma$ by simply γ. When Γ already contains $\gamma :: \psi$, this alone is sufficient.

But we need to be a bit more careful with type schemes and substitution. An advantage of split form schemes is that they contain the specification of quantified nodes inside their binders. This simplifies rule S-INST and S-NEW. However, a side-effect of this specification is that even variables that do not occur in the body of the scheme are constrained. As a result, some uses of M-INST, which ignored the constraint as it was not kept in the type scheme, are no longer valid with S-INST.

To assist the weaker S-INST we add a new rule S-BIND, which mimics the derived rule M-BIND, allowing to discard node descriptions that are not referenced anymore. Since $\alpha :: \psi$ is rightmost in the context of the premise, and γ is not α, it can safely be discarded in the conclusion. To replace a use of M-INST on an absent variable, one just needs the following two steps, when there is only one quantifier.

$$\textsc{s-Bind}\ \frac{\textsc{s-Inst}\ \dfrac{\Gamma, \alpha :: \psi \vdash M : \forall(\alpha :: \psi)\ \gamma}{\Gamma, \alpha :: \psi \vdash M : \gamma[\alpha \leftarrow \alpha]} \qquad \alpha \neq \gamma}{\Gamma \vdash M : \gamma}$$

$$\textsc{s-Var}\ \frac{\vdash \Gamma \qquad x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \textsc{s-Inst}\ \frac{\Gamma \vdash M : \forall(\alpha :: \psi)\ \sigma \qquad \gamma :: \psi_2 \in \Gamma \qquad \psi \subseteq \psi_2}{\Gamma \vdash M : \sigma[\alpha \leftarrow \gamma]}$$

$$\textsc{s-Gen}\ \frac{\Gamma, \alpha :: \psi \vdash M : \sigma}{\Gamma \vdash M : \forall(\alpha :: \psi)\ \sigma} \qquad \textsc{s-Bind}\ \frac{\Gamma, \alpha :: \psi \vdash M : \gamma \qquad \alpha \neq \gamma}{\Gamma \vdash M : \gamma}$$

$$\textsc{s-New}\ \frac{\Gamma, a, \alpha :: a \vdash M : \sigma \qquad \Gamma \vdash \forall(\alpha :: \varepsilon)\ \sigma}{\Gamma \vdash \nu(a) M : \forall(\alpha :: \varepsilon)\ \sigma} \qquad \textsc{s-Fun}\ \frac{\Gamma, x : \gamma_0 \vdash M : \gamma_1}{\Gamma \vdash \lambda(x) M : \forall(\gamma :: \gamma_0 \to \gamma_1)\ \gamma}$$

$$\textsc{s-App}\ \frac{\Gamma \vdash M_1 : \gamma_1 \qquad \Gamma \vdash M_2 : \gamma_2 \qquad \gamma_1 :: \gamma_2 \to \gamma \approx \psi \in \Gamma}{\Gamma \vdash M_1\ M_2 : \gamma}$$

$$\textsc{s-Let}\ \frac{\Gamma \vdash M_1 : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash M_2 : \gamma_2}{\Gamma \vdash \mathsf{let}\, x = M_1\, \mathsf{in}\, M_2 : \gamma_2} \qquad \textsc{s-Ann}\ \frac{\Gamma \vdash \forall(\mathsf{ftv}(\tau))\ \tau}{\Gamma \vdash (\tau) : \forall(\mathsf{ftv}(\tau))\ \lceil \tau \to \tau \rceil}$$

$$\textsc{s-Witness}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Eq} : \forall(\alpha, \gamma :: \mathsf{eq}(\alpha, \alpha))\ \gamma} \qquad \textsc{s-Use}\ \frac{\Gamma \vdash (\mathsf{eq}(\tau_1, \tau_2))\ M_1 : \gamma_1 \qquad \Gamma, \tau_1 \doteq \tau_2 \vdash M_2 : \gamma_2}{\Gamma \vdash \mathsf{use}\, M_1 : \mathsf{eq}(\tau_1, \tau_2)\, \mathsf{in}\, M_2 : \gamma_2}$$

Fig. 6. Typing rules in split form

If there are several quantifiers, they must first all be instantiated, then S-BIND is applied for unused variables, which must be on the right of the context, then S-GEN is used repeatedly to reconstruct the type scheme.

As we mentioned before, in the mixed form M-BIND is derivable, so we can translate back from split form to mixed form:

$$\begin{array}{l}\textsc{m-Gen} \\ \textsc{m-Inst}\end{array}\ \frac{\dfrac{\Gamma, \alpha :: \psi_1 \vdash M : \psi^\gamma}{\Gamma \vdash M : \forall(\alpha)\ \psi^\gamma[\alpha \leftarrow \psi_1^\alpha]} \qquad \Gamma \vdash \psi^\gamma}{\Gamma \vdash M : \psi^\gamma[\alpha \leftarrow \psi^\gamma]}$$

4.2 Substitutivity

As we just mentioned at the beginning of this section, an essential property of type systems is substitutivity of types. We restate it formally in split form, as the presence of node descriptions in the typing context makes it differ from other type systems.

To simplify the handling of the context, we first define an interleaving operation $\Gamma \mid \Delta$, which merges a *pure typing context* Γ, which term variables, rigid type variables, and equations —but no node descriptions—, with a label context Δ, composed only of node descriptions $\alpha :: \psi$. The non-deterministic interleaving operation $\Gamma \mid \Delta$ is defined by reordering the components of Δ, and inserting them between the components of Γ. $\vdash \Gamma \mid \Delta$ expresses that Γ and Δ can be merged to obtain a well-formed typing context. Checking the well-formedness is decidable: one just has to try inserting at all possible

positions, and check whether the result is well-formed. The judgment $\Gamma \mid \Delta \vdash M : \zeta$ implies $\vdash \Gamma \mid \Delta$.

Now the intuition is that apply a substitution $\Delta \vdash \theta : \Delta'$ to a typing context $\Gamma \mid \Delta'$ turns it into the typing context $\theta(\Gamma) \mid \Delta'$.

Our first lemma checks that well-formedness is preserved.

Lemma 1 (Substitution-WF). *If $\Gamma \mid \Delta \vdash \sigma$ and $\Delta \vdash \theta : \Delta'$, and $\vdash \theta(\Gamma) \mid \Delta'$, then $\theta(\Gamma) \mid \Delta' \vdash \theta(\sigma)$.*

We can then state and prove the substitution lemma.

Lemma 2 (Substitution). *If $\Gamma \mid \Delta \vdash M : \sigma$ and $\Delta \vdash \theta : \Delta'$, and $\vdash \theta(\Gamma) \mid \Delta'$, then $\theta(\Gamma) \mid \Delta' \vdash M : \theta(\sigma)$, using the same derivation.*

Proof. By induction on $\Gamma \mid \Delta \vdash M : \sigma$.

Case S-GEN: by induction hypothesis, we know that $\vdash \theta(\Gamma) \mid \Delta'$. Since $\Gamma \mid \Delta \vdash \psi$, if we choose α outside of $\mathsf{dom}(\Delta')$, by lemma 1, $\vdash (\theta(\Gamma) \mid \Delta'), \alpha :: \psi$, and as a result $(\theta(\Gamma) \mid \Delta'), \alpha :: \psi \vdash M : \theta(\sigma)$ and we conclude.

Case S-BIND is similar.

Case S-INST: By induction hypothesis we have $\theta(\Gamma) \mid \Delta' \vdash M : \forall(\alpha :: \psi)\ \sigma$. We assume that $\theta(\gamma) = \gamma'$. From $\Delta \vdash \theta : \Delta'$, we know that $\gamma' :: \psi_3 \in \Delta'$ and $\psi_2 \subseteq \psi_3$. As a result we also get $\psi \subseteq \psi_3$, and we conclude by rule S-INST.

Other cases are easy.

4.3 Generic instance

Generic instance in ML is defined as the iteration of type instantiation, followed by generalization of freshly introduced type variables. Namely, σ_1 is more general than σ_2, written $\sigma_1 \prec \sigma_2$ if and only if σ_1 and σ_2 are of the form $\forall(\bar{\alpha}_1)\ \tau$ and $\forall(\bar{\alpha}_2)\ \tau[\bar{\alpha}_1 \leftarrow \tau_1]$, respectively, where α_2 are not free in σ_1.

While $\prec$ is a binary relation in ML, it also depends on the typing context in our setting, because type instantiation must preserve well-formedness, which depends on the typing context.

Definition 4. *We say that σ_2 is an instance of σ_1 under Γ, which we write $\Gamma \vdash \sigma_1 \prec \sigma_2$, if and only if σ_1 and σ_2 are of the form $\forall(\Delta_1)\ \alpha$ and $\forall(\Delta_2)\ \theta(\alpha)$ such that $\Delta_1 \vdash \theta : (\Gamma, \Delta_2)$, $\mathsf{dom}(\theta) \subseteq \mathsf{dom}(\Delta_1)$, $\mathsf{dom}(\Delta_2) \cap \mathsf{dom}(\Gamma) = \emptyset$, and $\vdash \Gamma, \Delta_2$.*

If the judgment $\Gamma \vdash M : \sigma$ holds and $\Gamma \vdash \sigma \prec \sigma'$, then the judgment $\Gamma \vdash M : \sigma'$ also holds—it is in fact derivable from the former by a sequence of S-INST, S-BIND, and S-GEN rules.

The generic instance relation between type schemes may be lifted into a relation between contexts. We write $\Gamma \prec \Gamma'$ to mean that Γ and Γ' coincide except on program variables where the images are in a point-wise generic instance relation.

Strengthening of the typing context is the opposite of generic instance. We prove monotonicity with respect to strengthening.

Lemma 3 (Monotonicity). *If $\Gamma \vdash M : \sigma$ and $\Gamma' \prec \Gamma$ then $\Gamma' \vdash M : \sigma$.*

Proof. The proof goes by induction on the derivation.

– Rule S-VAR: we have $x : \sigma \in \Gamma$ and $x : \sigma' \in \Gamma'$, with $\Gamma \vdash \sigma' \prec \sigma$. We can replace $\Gamma \vdash x : \sigma$ by $\Gamma \vdash x : \sigma'$ followed by instantiation and generalization steps leading to $\Gamma \mid \Delta \vdash x : \sigma$.

– All other cases are trivial induction steps.

4.4 Canonical derivations

The rules in Figure 6 are not syntax directed. Indeed, rules S-INST, S-GEN and S-BIND may be used anywhere in a derivation. However, we may put derivations in canonical forms—which helps reasoning by cases on derivation in proofs. We will also replace S-NEW by the following S-NEW', which is trivially derivable by the addition S-GEN steps before S-NEW, and S-INST steps after.

$$\text{S-NEW'}\quad \frac{\Gamma, a, \alpha :: a, \Delta \vdash M : \gamma \qquad \Gamma, \alpha :: \varepsilon, \Delta \vdash \gamma}{\Gamma, \alpha :: \varepsilon, \Delta \vdash \nu(a) M : \gamma}$$

Notice that α is the unique flexible type variable that labels a.

Definition 5. *A derivation is canonical if:*

- *It uses* S-NEW' *in place of* S-NEW.
- *Rule* S-INST *is only used immediately below another rule* S-INST*, or one of rules* S-VAR*,* S-FUN*,* S-ANN *or* S-WITNESS*.*
- *Rule* S-GEN *is only used immediately above another rule* S-GEN *or the left premise of rule* S-LET*, or at the end of the derivation.*
- *Rule* S-BIND *is only used immediately above another rule* S-BIND*, the left premise of rule* S-LET*, the right premise of* S-USE*, or rules* S-GEN *and* S-NEW'*, or at the end of the derivation.*

Lemma 4. *Any judgment has a canonical derivation.*

Proof. We prove that S-INST can be pushed up until it reaches a legal position, and S-BIND and S-GEN can be pushed down similarly.

For rule S-BIND we need to see all the rules which do not allow S-BIND as a premise.

– Rule S-INST is impossible, since its premise is not a monotype.

– In rule S-FUN, we move S-BIND to the conclusion, applying permutaiton to the premise, to put $x : \gamma_0$ last.

– In rule S-APP, from either of the premise, we move S-BIND to the conclusion, weakening the other premise (*i.e.* adding $\alpha :: \psi$ to Γ.)

– Similarly for the second premise of rule S-LET and the first premise of S-USE.

The only remaining case is rule S-GEN above rule S-INST. This amounts to the derivation:

$$\text{S-INST}\ \dfrac{\text{S-GEN}\ \dfrac{\Gamma, \alpha :: \psi \vdash M : \sigma}{\Gamma \vdash M : \forall(\alpha :: \psi)\, \sigma} \qquad \gamma :: \psi_2 \in \Gamma \qquad \psi \subseteq \psi_2}{\Gamma \vdash M : \sigma[\alpha \leftarrow \gamma]}$$

We split Γ as $\Gamma_0 \mid \Delta$. We can apply the substitution lemma to $\Gamma_0 \mid \Delta, \alpha :: \psi \vdash M : \sigma$, with $\Delta, \alpha :: \psi \vdash [\alpha \leftarrow \gamma] : \Delta$, which gives $\Gamma_0 \mid \Delta \vdash M : \sigma[\alpha \leftarrow \gamma]$, with a derivation two steps shorter than the original one.

$$\textsc{Subst}\quad \frac{\alpha_1 \in \mathsf{ffv}(\varphi) \qquad \alpha_1 \neq \alpha_2}{\varphi \wedge \alpha_1 \mapsto \alpha_2 \Rrightarrow \varphi[\alpha_1 \mapsto \alpha_2] \wedge \alpha_1 \mapsto \alpha_2} \qquad \textsc{Delete}\quad \varphi \wedge \alpha \mapsto \alpha \Rrightarrow \varphi$$

$$\textsc{Merge}\quad \varphi \wedge \alpha :: \psi_1 \wedge \alpha :: \psi_2 \Rrightarrow \varphi \wedge \alpha :: \psi_1 \approx \psi_2$$

$$\textsc{Decomp-Arr}\quad \varphi \wedge \alpha :: (\alpha_1 \to \alpha_2) \approx (\alpha_1{}' \to \alpha_2{}') \approx \psi \Rrightarrow \varphi \wedge \alpha :: (\alpha_1 \to \alpha_2) \approx \psi \wedge \alpha_1{}' \mapsto \alpha_1 \wedge \alpha_2{}' \mapsto \alpha_2$$

$$\textsc{Decomp-Eq}\quad \varphi \wedge \alpha :: \mathsf{eq}(\alpha_1, \alpha_2) \approx \mathsf{eq}(\alpha_1{}', \alpha_2{}') \approx \psi \Rrightarrow \varphi \wedge \alpha :: \mathsf{eq}(\alpha_1, \alpha_2) \approx \psi \wedge \alpha_1{}' \mapsto \alpha_1 \wedge \alpha_2{}' \mapsto \alpha_2$$

$$\textsc{Context}\quad \frac{\varphi \Rrightarrow \varphi'}{\exists \alpha.\varphi \Rrightarrow \exists \alpha.\varphi'} \qquad \textsc{Extrude}\quad \frac{\alpha \notin \mathsf{ffv}(\varphi_1)}{\varphi_1 \wedge (\exists \alpha.\varphi_2) \Rrightarrow \exists \alpha.(\varphi_1 \wedge \varphi_2)}$$

Fig. 7. Unification algorithm

4.5 Existence of principal derivations

In this section, we prove the existence of principal derivations by providing an algorithm that either finds one, or fails when the input program admits no typing derivation.

Our algorithm is a constraint-based variant of W. The unification part takes inspiration from a structural polymorphism framework developed for variants and records [**?**], and the inference part reproduces the style used for semi-explicit first-class polymorphism [**?**].

We first describe unification on ambivalent types.

Definition 6. *A* unification problem φ *is a formula of the form:*

$$\varphi ::= \emptyset \mid \exists(\alpha)\ \varphi \mid \varphi \wedge \varphi \mid \alpha :: \psi \mid \alpha \mapsto \alpha$$

Definition 7. *A* unifier *(also called a solution) of a unification problem φ is a pair of a label context Δ and a variable substitution θ such that θ maps both sides of every elementary substitution $\alpha_1 \mapsto \alpha_2$ in φ to the same variable, and for any equation $\alpha :: \psi$ in φ, there is a ψ_0 containing at most one proper raw type, such that $\theta(\alpha) :: \psi_0 \in \Delta$ and $\theta(\psi) \subseteq \psi_0$.*

The pair (Δ, θ) corresponds to a substitution in the mixed view.

The resolution of a unification problem is a sequence of transformations that preserve the set of unifiers. These transformations are defined on Figure 7. They are of three kinds.

- we may apply a substitution (or, as a special case, discard it if meaningless);
- we may merge two constraints;
- or we may decompose a constraint, introducing new substitutions, when the same constructor $\to$ or eq appears twice in a constraint.

A unifier (Δ, θ) is more general than (Δ', θ'), noted $(\Delta, \theta) \sqsubseteq (\Delta', \theta')$, if there exists a substitution θ_1 such that $\theta_1 \circ \theta = \theta'$ and (Δ', θ_1) is a unifier of Δ, or equivalently $\Delta \vdash \theta_1 : \Delta'$.

A unification problem (Δ, θ) is solved when no applicable substitution is left, and the merging and decomposition rules do not apply anymore. At that point we can directly read a unifier from the solved problem, collecting the substitutions in θ, and the constraints in Δ.

Lemma 5. *Unification of ambivalent types admits a most general unifier, and there exists an algorithm returning such an unifier.*

Proof. We first prove termination. We say that a variable is solved if it appears at most once, and only on the left hand side of a substitution. We take as measure $\langle$*number of unsolved variables, total size of node descriptions, number of substitutions*$\rangle$. DELETE decreases the number of substitutions, SUBST decreases the number of unsolved variables, remaining ones decrease the total size.

Next we prove that if $\varphi \Rrightarrow \varphi'$, and $\exists\bar{\alpha}.\Delta \wedge \theta$ is a solution of φ, then it is also a solution of φ', and reciprocally.

In rule SUBST, any solution of the premise or the conclusion must satisfy $\theta(\alpha_1) = \theta(\alpha_2)$, so that $\theta(\varphi) = \theta(\varphi[\alpha_1 \leftarrow \alpha_2])$.

In rule DELETE, $\alpha \mapsto \alpha$ is redundant, so that the two constraints are equivalent.

In rule MERGE, a solution of either the premise or the conclusion must satisfy $\theta(\alpha) = \alpha'$ and $\theta(\psi_1 \approx \psi_2) = \theta(\psi_1) \approx \theta(\psi_2) \subset \Delta(\alpha')$, so they are equivalent.

In rule DECOMP-ARR, if (Δ, θ) is a solution of the premise then $\theta(\alpha_1 \to \alpha_2 \approx \alpha_1{}' \to \alpha_2{}' \approx \psi) \subseteq \Delta(\alpha)$, which, combined with the above restriction on Δ, means that $\Delta(\alpha_1) = \Delta(\alpha_1{}')$ and $\Delta(\alpha_2) = \Delta(\alpha_2{}')$, and as a result this is also a solution of the conclusion. Reciprocally, as solution of the conclusion must satisfy the same equalities on type variables, so the it is also a solution of the premise. The correctness of rule DECOMP-EQ can be proved similarly.

Our type inference algorithm, given in figure 8, is based on constraint generation and solving. For each rewriting rule of the form $\Gamma \triangleright M : \gamma \Rrightarrow \exists\bar{\alpha}.\varphi$ we assume implicitly the premise $\bar{\alpha} \cap (\mathsf{ffv}(\Gamma) \cup \{\gamma\}) = \emptyset$. In these rules, Γ does not contain node descriptions.

To solve the type inference problem of whether $\Gamma' \vdash M : \gamma$ is true for some instance Γ' of Γ, we first split Γ into Γ_0 and Δ_0 such that $\Gamma = \Gamma_0 \mid \Delta_0$, and then apply the rewriting rules of figures 7 and 8 starting from the inference constraint $\Gamma_0; \Delta_0 \triangleright M : \gamma$. If we can rewrite it to $\exists\bar{\alpha}.\Delta_1 \wedge \theta$, we obtain a principal derivation $\theta(\Gamma_0) \wedge \Delta_1 \vdash M : \theta(\gamma)$. Otherwise, since the unification rules can never fail, rewriting must have been blocked by one of the premises. But again, premises about variable names never fail: we can always choose some variable names to satisfy them. The remaining premises are variable access for $\Gamma \triangleright x : \gamma$, which may fail if x is not defined in Γ, and the well-formedness checks $\vdash \Gamma \mid \Delta$, in the last 3 rules. The well-formedness checks ensure simultaneously coherence, proper scoping of rigid variables, and the absence of cycles inside ambivalent types.

We define the restriction of a unifier to a set of variables as $(\Delta, \theta)|_{\bar{\alpha}} = (\Delta|_{\mathsf{ffv}_\Delta(\theta(\bar{\alpha}))}, \theta|_{\bar{\alpha}})$. This is used in rules I-NEW, I-USE and I-CONCL.

The following lemma justifies our delayed well-formedness checks.

$$\textsc{I-Var}\quad \frac{\Gamma(x) = \forall(\Delta)\ \alpha \qquad \mathsf{dom}(\Delta) = \bar{\alpha}}{\Gamma \triangleright x : \gamma \Rrightarrow \exists \bar{\alpha}.\Delta \wedge \gamma \mapsto \alpha}$$

$$\textsc{I-New}\quad \frac{\Gamma, a \triangleright M_1 : \gamma \Rrightarrow \exists \bar{\alpha}.\Delta \wedge \theta \qquad \vdash \theta(\Gamma), a \mid \Delta \qquad \Delta' = \Delta \setminus \{\alpha :: a \in \Delta\}}{\Gamma \triangleright \nu(a) M_1 : \gamma \Rrightarrow \exists \bar{\alpha}\alpha_1.((\Delta'[\alpha \leftarrow \alpha_1]^{\alpha::a\in\Delta}, \alpha_1 :: \varepsilon), \theta[\alpha \leftarrow \alpha_1]^{\alpha::a\in\Delta})|_{\mathsf{ffv}(\Gamma),\gamma}}$$

$$\textsc{I-Fun}\quad \Gamma \triangleright \lambda(x) M_1 : \gamma \Rrightarrow \exists \alpha_1 \alpha_2.(\Gamma, x : \alpha_1 \triangleright M_1 : \alpha_2) \wedge \alpha_1 :: \varepsilon \wedge \alpha_2 :: \varepsilon \wedge \gamma :: \alpha_1 \to \alpha_2$$

$$\textsc{I-App}\quad \Gamma \triangleright M_1\ M_2 : \gamma \Rrightarrow \exists \alpha_1 \alpha_2.(\Gamma \triangleright M_1 : \alpha_1) \wedge (\Gamma \triangleright M_2 : \alpha_2) \wedge \alpha_1 :: \alpha_2 \to \gamma$$

$$\textsc{I-Let}\quad \frac{\Gamma \triangleright M_1 : \gamma_1 \Rrightarrow \exists \bar{\alpha}.\Delta \wedge \theta \qquad \gamma_1 \notin \bar{\alpha} \qquad \sigma = \mathsf{Gen}(\theta(\gamma_1), \theta(\Gamma), \Delta)}{\Gamma \triangleright \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 : \gamma \Rrightarrow \exists \bar{\alpha}, \gamma_1.\Delta \wedge \theta \wedge (\theta(\Gamma), x : \sigma \triangleright M_2 : \theta(\gamma))}$$

$$\textsc{I-Ann}\quad \frac{\wr \tau \to \tau \int = \forall(\Delta)\ \alpha \qquad \mathsf{ftv}_\Delta(\alpha) = \bar{\alpha}}{\Gamma \triangleright (\tau) : \gamma \Rrightarrow \exists \bar{\alpha}, \mathsf{dom}(\Delta).\Delta \wedge \gamma \mapsto \alpha}$$

$$\textsc{I-Witness}\quad \Gamma \triangleright \mathsf{Eq} : \gamma \Rrightarrow \exists \alpha.\gamma :: \mathsf{eq}(\alpha, \alpha)$$

$$\textsc{I-Use}\quad \frac{\Gamma, \tau_1 = \tau_2 \triangleright M_2 : \gamma \Rrightarrow \exists \bar{\alpha}.\Delta \wedge \theta \qquad \vdash \theta(\Gamma), \tau_1 = \tau_2 \mid \Delta}{\Gamma \triangleright \mathsf{use}\ M_1 : \mathsf{eq}(\tau_1, \tau_2)\ \mathsf{in}\ M_2 : \gamma \Rrightarrow \exists \bar{\alpha}, \alpha_1.(\Gamma \triangleright (\mathsf{eq}(\tau_1, \tau_2)) M_1 : \alpha_1) \wedge (\Delta, \theta)|_{\mathsf{ffv}(\Gamma),\gamma}}$$

$$\textsc{I-Concl}\quad \frac{\Delta_0 \wedge (\Gamma_0 \triangleright M : \gamma) \Rrightarrow \exists \bar{\alpha}.\Delta \wedge \theta \qquad \vdash \theta(\Gamma_0) \mid \Delta}{\Gamma_0; \Delta_0 \triangleright M : \gamma \Rrightarrow \exists \bar{\alpha}.(\Delta, \theta)|_{\mathsf{ffv}(\Gamma_0, \Delta_0),\gamma}}$$

Fig. 8. Constraint generation

Lemma 6 (WF-postponement). *If $\Delta \vdash \theta : \Delta'$ and $\vdash \theta(\Gamma) \mid \Delta'$, then $\vdash \Gamma \mid \Delta$.*

Proof. An environment is well-formed if

1. term and type variable are introduced only once (rules WF-CTX-EXPR, WF-CTX-RIGID and WF-CTX-FLEX),
2. type variables (both rigid and flexible) are introduced before they are used (rules WF-TYPE-$\star$ and WF-SCHEME),
3. all ambivalent types are coherent (rule WF-TYPE-AMBIVALENT).

The first point is independent of substitution and merging (Δ and Δ' are already functions). The second one requires that a node constraint $\alpha :: \psi$ appears after the variables in $\mathsf{ffv}(\psi)$ and $\mathsf{frv}(\psi)$ are introduced, and before α is used. The last one requires that this node constraint appears after the equations required for $\Gamma \Vdash \psi$; the requirement on function and eq types is already fulfilled since Δ and Δ' are parts of solutions. To summarize, for obtaining $\Gamma \mid \Delta$ to be well-formed when $\theta(\Gamma) \mid \Delta'$ is well-formed, we only need the order of node constraints to satisfy the above ordering requirements.

Let $\Gamma' = \theta(\Gamma) \mid \Delta'$ such that $\vdash \Gamma'$. We construct $\Gamma_0 = \Gamma \mid \Delta$ from Γ' in the following way:

- we replace each $x : \theta(\sigma)$ in Γ' by $x : \sigma$ from Γ;
- we replace each $\alpha' :: \psi'$ in Γ', by the set of node constraints $\{\alpha :: \psi \in \Delta \mid \theta(\alpha) = \alpha'\}$. Since $\bigcup\{\theta(\Delta(\alpha)) \mid \theta(\alpha) = \alpha'\} \subset \Delta'(\alpha')$, and $\alpha' \notin \mathsf{ffv}(\Delta'(\alpha'))$ (by well-formedness), the order of between these node constraints doesn't matter,
- the other components (rigid variables and equations) are left unchanged.

Since we have preserved the ordering of components in Γ', all ordering constraints are satisfied, and Γ_0 is well-formed.

Theorem 2. *If* $\Gamma \triangleright M : \gamma \Rrightarrow \exists\bar{\alpha}.\Delta \wedge \theta$ *and* $\vdash \theta(\Gamma) \mid \Delta$ *then* $\theta(\Gamma) \mid \Delta \vdash M : \theta(\gamma)$.

Proof. Note first that since the rewriting rules for typing problems convert them to unification problems, in a way that does not depend on other constraints, when it succeeds the type inference algorithm of figure 8 provides a unique solution modulo a renaming between equivalent type variables. This allows us to choose whatever resolution strategy we wish.

The proof is by induction on M.

If $M = x$, the constraint Δ ensures that α is bound to an instance of $\forall(\Delta)\ \alpha$, so that in the resulting $\Delta, \alpha \mapsto \gamma$, $\theta(\gamma)$ is bound to such an instance. The corresponding derivation is obtained with S-VAR followed by repeated uses of S-INST.

If $M = \mathsf{Eq}$ or $M = (\psi)$, the constraint is already solved and satisfies the requirements. For $M = (\psi)$ we need to apply repeatedly S-INST after S-ANN.

If $M = \lambda(x)M_1$, since we assumed $\Gamma \triangleright \lambda(x)M_1 : \gamma \Rrightarrow \exists\alpha_1\alpha_2\bar{\alpha}.\Delta \wedge \theta$, we have also $\Gamma, x : \alpha_1 \triangleright M_1 : \alpha_2 \Rrightarrow \exists\bar{\alpha}.\Delta_1 \wedge \theta_1$, and $(\Delta_1, \theta_1) \sqsubseteq (\Delta, \theta)$. This means there is a substitution $\Delta_1 \vdash \theta_2 : \Delta$ such that $\theta_2 \circ \theta_1 = \theta$. By lemma 6 $\vdash \theta_1(\Gamma) \mid \Delta_1$, which implies $\vdash \theta_1(\Gamma), x : \theta_1(\alpha_1) \mid \Delta_1$. By induction hypothesis we get $\theta_1(\Gamma), x : \theta_1(\alpha_1) \mid \Delta_1 \vdash M_1 : \theta_1(\alpha_2)$. By the substitution lemma, $\theta(\Gamma), x : \theta(\alpha_1) \mid \Delta \vdash M_1 : \theta(\alpha_2)$. Since we also have $\theta(\alpha_1) \to \theta(\alpha_2) \in \Delta(\theta(\gamma))$, we can conclude that $\theta(\Gamma) \mid \Delta \vdash \lambda(x)M_1 : \theta(\gamma)$, using rule S-FUN.

If $M = M_1\ M_2$, then $\Gamma \triangleright M_1 : \alpha_1 \Rrightarrow \exists\bar{\alpha}_1.\Delta_1 \wedge \theta_1$ and $\Gamma \triangleright M_2 : \alpha_2 \Rrightarrow \exists\bar{\alpha}_2.\Delta_2 \wedge \theta_2$, with $\Delta_1 \vdash \theta_1' : \Delta$ and $\Delta_2 \vdash \theta_2' : \Delta$ such that $\theta = \theta_1' \circ \theta_1 = \theta_2' \circ \theta_2$. By lemma 6 we have $\vdash \theta_1(\Gamma) \mid \Delta_1$ and $\vdash \theta_2(\Gamma) \mid \Delta_2$. By induction hypothesis and the substitution lemma, we obtain $\theta(\Gamma) \mid \Delta \vdash M_1 : \theta(\alpha_1)$ and $\theta(\Gamma) \mid \Delta \vdash M_2 : \theta(\alpha_2)$. Thanks to $\alpha_1 :: \alpha_2 \to \gamma$, we also have $\theta(\alpha_2) \to \theta(\gamma) \in \Delta(\theta(\alpha_1))$, so that we can conclude by S-APP.

If $M = \mathsf{let}\, x = M_1\, \mathsf{in}\, M_2$, then $\Gamma \triangleright M_1 : \gamma_1 \Rrightarrow \exists\bar{\alpha}_1.\Delta_1 \wedge \theta_1$ and $\Gamma, x : \forall(\bar{\alpha}_1)\ \gamma_1 \triangleright M_2 : \gamma \Rrightarrow \exists\bar{\alpha}_2.\Delta_2 \wedge \theta_2$ with $\forall(\bar{\alpha}_1)\ \gamma_1 = \mathsf{Gen}(\theta_1(\gamma_1), \theta_1(\Gamma), \Delta_1)$, $\Delta_1 \vdash \theta_1' : \Delta$ and $\Delta_2 \vdash \theta_2' : \Delta$ such that $\theta = \theta_1' \circ \theta_1 = \theta_2' \circ \theta_2$. As in previous cases, we obtain $\theta(\Gamma) \mid \Delta \vdash M_1 : \theta(\gamma_1)$ and $\theta(G), x : \theta_1'(\forall(\bar{\alpha}_1)\ \gamma_1) \mid \Delta \vdash M_2 : \theta(\gamma)$. Note that the variables $\bar{\alpha}_1$ are a subset of $\bar{\alpha}$ not accessible from other constraints (they are neither free in Γ and θ), and as a result they are not touched by subsequent constraint solving. That is, if $\alpha \in \bar{\alpha}_1$, then $\theta(\alpha) = \alpha$ and $\Delta(\alpha) = \theta_1'(\Delta_1(\alpha))$. We can build a derivation by first applying S-GEN for each variable in $\bar{\alpha}_1$, obtaining $\theta(\Gamma) \mid \Delta \setminus \bar{\alpha}_1 \vdash M_1 : \forall(\bar{\alpha}_1)\ \gamma_1$, then applying the weakening lemma for node descriptions in order to reintroduce in Γ the nodes that S-GEN removed. Then we can apply S-LET.

If $M = \Gamma \triangleright \mathsf{use}\ M_1 : \mathsf{eq}(\tau_1,\tau_2)\ \mathsf{in}\ M_2 : \gamma$, then $\Gamma \triangleright (\mathsf{eq}(\tau_1,\tau_2))\,M_1 : \gamma_1 \Rrightarrow \exists\bar{\alpha}_1.\Delta_1 \wedge \theta_1$ and $\Gamma, \tau_1 = \tau_2 \triangleright M_2 : \gamma \Rrightarrow \exists\bar{\alpha}_2.\Delta_2 \wedge \theta_2$ with $\Delta_1 \vdash \theta_1' : \Delta$ and $\Delta_2|_{\mathsf{ffv}_{\Delta_2}(\theta_2(\Gamma),\theta_2(\gamma))} \vdash \theta_2' : \Delta$ such that $\theta = \theta_1' \circ \theta_1 = \theta_2' \circ \theta_2|_{\mathsf{ffv}(\Gamma),\gamma}$. We obtain $\theta(\Gamma) \mid \Delta \vdash (\mathsf{eq}(\tau_1,\tau_2))\,M_1 : \theta(\gamma_1)$ and $\theta(G), \tau_1 \doteq \tau_2 \mid \Delta \vdash M_2 : \theta(\gamma)$ (after applying S-BIND to remove useless variables). We conclude by rule S-USE.

If $M = \Gamma \triangleright \nu(a)\,M_1 : \gamma$, then $\Gamma, a \triangleright M_1 : \gamma \Rrightarrow \exists\bar{\alpha}.\Delta_1 \wedge \theta_1$ and $\vdash \theta_1(\Gamma), a \mid \Delta_1$, which gives $\theta_1(\Gamma), a \mid \Delta_1 \vdash M_1 : \theta_1(\gamma)$ by induction hypothesis. We can split Δ_1 into 4 parts, $\Delta_2, \Delta_3, \Delta_4, \Delta_5$, such that $\mathsf{dom}(\Delta_2) \subset \mathsf{ffv}_{\Delta_1}(\theta_1(\Gamma))$, $\Delta_3 = \{\alpha :: a \in \Delta_1\}$, $\mathsf{dom}(\Delta_4) \subset \mathsf{ffv}_{\Delta_1}(\theta_1(\gamma)) \setminus \mathsf{ffv}_{\Delta_1}(\theta_1(\Gamma))$, and $\mathsf{dom}(\Delta_5) \cap \mathsf{ffv}_{\Delta_1}(\theta_1(\Gamma), \theta_1(\gamma)) = \emptyset$. We first discard Δ_5 using S-BIND to obtain $\theta_1(\Gamma) \mid \Delta_2, a, \Delta_3, \Delta_4 \vdash M_1 : \theta_1(\gamma)$. We then merge all variables in Δ_3, since they all represent the same type, using $\theta_3 = [\alpha \leftarrow \alpha_1]^{\alpha::a\in\Delta_1}$, and obtain $\theta_1(\Gamma) \mid \Delta_2, a, \alpha_1 :: a, \theta_3(\Delta_4) \vdash \theta_3(\theta_1(\gamma))$. We can then apply S-NEW', to obtain $\theta_1(\Gamma) \mid \Delta_2, \alpha :: \varepsilon, \theta_3(\Delta_4) \vdash M_1 : \theta_3(\theta_1(\gamma))$, which is actually identical to $\theta_3(\theta_1(\Gamma)) \mid \Delta'|_{\mathsf{ffv}_{\Delta'}(\theta_3(su_1(\Gamma)),\theta_3(\theta_1(\gamma)))} \vdash M_1 : \theta_3(\theta_1(\gamma))$ where $\Delta' = \theta_3(\Delta_1 \setminus \Delta_3) \cup \{\alpha :: \varepsilon\}$, and lets us conclude.

Corollary 1 (Soundness). *If $\Gamma_0; \Delta_0 \triangleright M : \gamma \Rrightarrow \exists\bar{\alpha}.\Delta \wedge \theta$ then $\theta(\Gamma_0) \mid \Delta \vdash M : \theta(\gamma)$ and $\Delta_0 \vdash \theta : \Delta$.*

Proof. From the hypothesis, we must have $\Delta_0 \wedge (\Gamma_0 \triangleright M : \gamma) \Rrightarrow \exists\bar{\alpha}.\Delta' \wedge \theta'$ and $\Delta_0 \vdash \theta' : \Delta'$ with $(\Delta, \theta) = (\Delta', \theta')|_{\mathsf{ffv}(\Gamma_0,\Delta_0),\gamma}$. In turn this means that we have $\Gamma_0 \triangleright M : \gamma \Rrightarrow \exists\bar{\alpha}.\Delta_1 \wedge \theta_1$ with $\Delta_1 \vdash \theta_2 : \Delta'$ and $\theta_2 \circ \theta_1 = \theta'$. Since $\vdash \theta'(\Gamma_0) \mid \Delta'$, by lemma 6 $\vdash \theta_1(\Gamma_0) \mid \Delta_1$. By theorem 2, we have $\theta_1(\Gamma_0) \mid \Delta_1 \vdash M : \theta_1(\gamma)$. By the substitution lemma we have $\theta(\Gamma_0) \mid \Delta' \vdash M : \theta(\gamma)$. By repeated uses of S-BIND we obtain $\theta(\Gamma_0) \mid \Delta \vdash M : \theta(\gamma)$. We also have $\Delta_0 \vdash \theta : \Delta$, since $\theta(\mathsf{dom}(\Delta_0)) \subset \mathsf{dom}(\Delta)$.

Theorem 3. *For any Γ, M, γ, if there exists a substitution $\emptyset \vdash \theta : \Delta$ such that $\theta(\Gamma) \mid \Delta \vdash M : \theta(\gamma)$ has a canonical derivation not finishing with* S-BIND*, then $\Gamma \triangleright M : \gamma \Rrightarrow \exists\bar{\alpha}.\Delta' \wedge \theta'$ and $(\Delta', \theta')|_{\mathsf{ffv}(\Gamma),\gamma} \sqsubseteq (\Delta, \theta)$ and $\vdash \theta'(\Gamma) \mid \Delta'$.*

Proof. We prove the theorem by induction on M. Concerning $\vdash \theta'(\Gamma) \mid \Delta'$, it is a consequence of $(\Delta', \theta')|_{\mathsf{ffv}(\Gamma),\gamma} \sqsubseteq (\Delta, \theta)$ and $\vdash \theta(\Gamma) \mid \Delta$ by lemma 6 for the part of Δ' which is not filtered out, and this is also true for the remnant as long as equations in Γ are not removed, as it is no longer changed by unification: all variables discarded from Δ' by the restriction are included in $\bar{\alpha}$. So we do not mention it in those cases.

If $M = x$, then rule S-VAR was applied, followed by S-INST. Assume $\Gamma(x) = \forall(\Delta_0)\ \alpha$, with $\mathsf{dom}(\Delta_0)$ fresh. The S-INST steps mean that θ is extended into $\Delta_0 \vdash \theta' : \Delta$, such that $\theta'(\alpha) = \theta'(\gamma)$. As a result $(\Delta_0, [\gamma \mapsto \alpha]) \sqsubseteq (\Delta, \theta')$, and we conclude.

Similarly, if $M = \mathsf{Eq}$ or $M = (a)$, then the inferred type is the most general one.

If $M = \lambda(x)\,M_1$, then rule S-FUN was applied, followed by S-INST. That is, we have $\theta(\Gamma) \mid \Delta \vdash M : \forall(\alpha :: \gamma_0 \to \gamma_1)\ \alpha$ and $\Delta(\theta(\gamma)) = \gamma_0 \to \gamma_1 \approx \psi$. By inversion $\theta(\Gamma) \mid \Delta, x : \gamma_0 \vdash M_1 : \gamma_1$, and $\theta(\Gamma) \mid \Delta \vdash \gamma_0$. By induction hypothesis, $\Gamma, x : \gamma_0 \triangleright M_1 : \gamma_1 \Rrightarrow \exists\bar{\alpha}.\Delta_1 \wedge \theta_1$ and $(\Delta_1, \theta_1)|_{\mathsf{ffv}(\Gamma),\gamma_0,\gamma_1} \sqsubseteq (\Delta, \theta)$. This means that $\theta_1(\Gamma) \mid \Delta_1 \vdash M : \forall(\alpha :: \theta_1(\gamma_0 \to \gamma_1))\ \alpha$. By applying S-INST once we obtain $\Gamma \mid \Delta_1, \alpha :: \theta_1(\gamma_0 \to \gamma_1) \vdash M : \alpha$, we set $\theta' = \theta_1[\gamma \leftarrow \alpha]$, $\Delta' = \Delta_1, \alpha :: \theta_1(\gamma_0 \to \gamma_1)$, and we conclude with $(\Delta', \theta')|_{\mathsf{ffv}(\Gamma),\gamma} \sqsubseteq (\Delta, \theta)$.

If $M = M_1 M_2$, then rule S-APP was applied. That is, we have $\theta(\Gamma) \mid \Delta \vdash M_1 : \gamma_1$, $\theta(\Gamma) \mid \Delta \vdash M_2 : \gamma_2$ and $\gamma_2 \to \theta(\gamma) \in \Delta(\gamma_1)$. By induction hypothesis, $\Gamma \triangleright M_1 :$

$\gamma_1 \Rrightarrow \exists\bar{\alpha}.\Delta_1 \wedge \theta_1$ and $\Gamma \triangleright M_2 : \gamma_2 \Rrightarrow \exists\bar{\alpha}.\Delta_2 \wedge \theta_2$, with $(\Delta_1,\theta_1)|_{\mathsf{ffv}(\Gamma),\gamma_1} \sqsubseteq (\Delta,\theta)$ and $(\Delta_2,\theta_2)|_{\mathsf{ffv}(\Gamma),\gamma_2} \sqsubseteq (\Delta,\theta)$. Note that here Δ is unchanged by S-APP, and θ already correctly handles γ_1 and γ_2, since we had $\gamma_1 :: \gamma_2 \to \theta(\gamma) \approx \psi \in \Delta$. For the same reason the unification problem $(\exists\bar{\alpha}.\Delta_1 \wedge \theta_1) \wedge (\exists\bar{\alpha}.\Delta_2 \wedge \theta_2) \wedge \gamma_1 :: \gamma_2 \to \gamma$ has a solution $\exists\bar{\alpha}.(\Delta',\theta')$, and $(\Delta',\theta')|_{\mathsf{ffv}(\Gamma),\gamma,\gamma_1,\gamma_2} \sqsubseteq (\Delta,\theta)$, since it is a conjunction of unification problems for which (Δ,θ) is a solution. We obtain $\vdash \theta'(\Gamma) \mid \Delta'$ by lemma 6, and the fact other variables of Δ' are not shared between Δ_1 and Δ_2. And we also have $(\Delta',\theta')|_{\mathsf{ffv}(\Gamma),\gamma} \sqsubseteq (\Delta',\theta')|_{\mathsf{ffv}(\Gamma),\gamma,\gamma_1,\gamma_2} \sqsubseteq (\Delta,\theta)$, since this is only a further restriction.

If $M = \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2$, then rule S-LET was applied. That is, we have $\theta(\Gamma) \mid \Delta \vdash M_1 : \sigma_1$ and $\theta(\Gamma), x : \sigma_1 \mid \Delta \vdash M_2 : \gamma_2$. For the left branch, there may be S-BIND and S-GEN steps. That is, there are α_1, Δ_1 and Δ_2 such that $\sigma_1 = \forall(\Delta_1)\ \alpha_1$, and $(\theta(G) \mid \Delta), \Delta_1, \Delta_2 \vdash M_1 : \alpha_1$. By induction hypothesis, $\Gamma \triangleright M_1 : \alpha_1 \Rrightarrow \exists\bar{\alpha}.\Delta_1' \wedge \theta_1'$ and $(\Delta_1',\theta_1')|_{\mathsf{ffv}(\Gamma),\alpha_1} \sqsubseteq (\Delta,\Delta_1,\Delta_2,\theta)$. As a consequence, if we pose $\sigma_1{}' = \mathsf{Gen}(\theta(\gamma_1),\theta_1'(\Gamma),\Delta_1')$ where $\theta = \theta_1 \circ \theta_1'$, we have that any instance of σ_1 is an instance of σ_1'. We can use the monotonicity lemma to obtain $\theta_1(\theta_1'(\Gamma), x : \sigma_1') \mid \Delta \vdash M_2 : \theta_1(\theta_1'(\gamma_2))$, which gives by induction hypothesis $\theta_1'(\Gamma), x : \sigma_1' \triangleright M_2 : \theta_1'(\gamma_2) \Rrightarrow \exists\bar{\alpha}.\Delta_2' \wedge \theta_2'$ and $(\Delta_2',\theta_2')|_{\mathsf{ffv}(\theta_1'(\Gamma),x:\sigma_1'),\theta_1'(\gamma_2)} \sqsubseteq (\Delta,\theta)$. Again we conclude by combining those two results.

If $M = \mathsf{use}\ M_1 : \mathsf{eq}(\tau_1,\tau_2)\ \mathsf{in}\ M_2$, then rule S-USE was applied. For the left premise, by combining the induction hypothesis for M_1 with the case for S-ANN and S-APP, we have $\Gamma \triangleright (\mathsf{eq}(\tau_1,\tau_2))\, M_1 : \gamma_1 \Rrightarrow \exists\bar{\alpha}.\Delta_1' \wedge \theta_1'$ with $(\Delta_1',\theta_1')|_{\mathsf{ffv}(\Gamma),\gamma_1} \sqsubseteq (\Delta,\theta)$. For the right premise, some S-BIND steps may have been involved, and the real premise becomes $(\theta(\Gamma) \mid \Delta), \Delta_2 \vdash M_2 : \theta(\gamma)$, so that $\vdash (\theta(\Gamma) \mid \Delta), \Delta_2$. By induction hypothesis we have $\Gamma, \tau_1 \doteq \tau_2 \triangleright M_2 : \gamma \Rrightarrow \exists\bar{\alpha}.\Delta_2' \wedge \theta_2'$ with $(\Delta_2',\theta_2')|_{\mathsf{ffv}(\Gamma),\gamma} \sqsubseteq ((\Delta,\Delta_2),\theta)$ and $\vdash \theta_2'(\Gamma), \tau_1 \doteq \tau_2 \mid \Delta_2'$. The last hypothesis satisfies the second premise of rule I-USE. Since $\mathsf{dom}(\Delta_2) \cap \mathsf{ffv}_\Delta(\theta(\Gamma),\gamma) = \emptyset$, the second hypothesis gives $(\Delta_2',\theta_2')|_{\mathsf{ffv}(\Gamma),\gamma} \sqsubseteq (\Delta,\theta)$. Combined with the first premise we obtain $(\Delta',\theta')|_{\mathsf{ffv}(\Gamma),\gamma} \sqsubseteq (\Delta,\theta)$, since (Δ_1',θ_1') may only instantiate γ if $\gamma \in \mathsf{ffv}(\Gamma)$. Finally, the outer Γ does not contain $\tau_1 \doteq \tau_2$, but $(\Delta_2'|_{\mathsf{ffv}_{\Delta_2'}(\theta_2'(\Gamma),\theta_2'(\gamma))},\theta_2') = (\Delta_2',\theta_2')|_{\mathsf{ffv}(\Gamma),\gamma} \sqsubseteq (\Delta,\theta)$ so that we obtain $\vdash \theta_2'(\Gamma) \mid \Delta_2'|_{\mathsf{ffv}_{\Delta_2'}(\theta_2'(\Gamma),\theta_2'(\gamma))}$ by lemma 6, and conclude that $\vdash \theta'(\Gamma) \mid \Delta'$ as usual.

If $M = \Gamma \triangleright \nu(a)\, M_1 : \gamma$, then rule S-NEW' was applied. Δ is of the form $\Delta_1, \alpha_0 :: \varepsilon, \Delta_2$, the conclusion being $(\theta(\Gamma) \mid \Delta_1), \alpha_0 :: \varepsilon, \Delta_2 \vdash M_1 : \theta(\gamma)$, and the premise, $(\theta(\Gamma) \mid \Delta_1), \alpha_0 :: a, \Delta_2 \vdash M_1 : \theta(\gamma)$. Some S-BIND steps may have been used in the premise, so that our starting point for induction is $(\theta(\Gamma) \mid \Delta_1), a, \alpha_0 :: a, \Delta_2, \Delta_3 \vdash M_1 : \theta(\gamma)$, with $\theta(\gamma) \notin \mathsf{dom}(\Delta_3)$. The induction hypothesis gives $\Gamma, a \triangleright M_1 : \gamma \Rrightarrow \exists\bar{\alpha}.\Delta_1' \wedge \theta_1'$ with $(\Delta_1',\theta_1')|_{\mathsf{ffv}(\Gamma),\gamma} \sqsubseteq ((\Delta_1,\alpha_0 :: a,\Delta_2,\Delta_3),\theta)$ and $\vdash \theta_1'(\Gamma), a \mid \Delta'$. Since we have restricted ourselves to $\mathsf{ffv}(\Gamma),\gamma$, the variables in Δ_3 are irrelevant, and we have $(\Delta_1',\theta_2')|_{\mathsf{ffv}(\Gamma),\gamma} \sqsubseteq ((\Delta_1,\alpha_0 :: a,\Delta_2),\theta)$. The well-formedness of $(\theta(\Gamma) \mid \Delta_1), \alpha_0 :: \varepsilon, \Delta_2$ means that neither Δ_1 nor Δ_2 contain a, so that all descriptions containing a in $\Delta_1'|_{\mathsf{ffv}_{\Delta_1'}(\theta_1'(\Gamma),\theta_1'(\gamma))}$ must be of the form $\alpha :: a$, so that we can map them to α_0. We pose $\Delta_2' = \Delta_1' \setminus \{\alpha :: a \in \Delta_1'\}$, $\theta_2' = [\alpha \leftarrow \alpha_0]^{\alpha::a\in\Delta_1'}$ and $(\theta',\Delta') = ((\theta_2'(\Delta_2'),\alpha_0 :: \varepsilon),\theta_2' \circ \theta_1')|_{\mathsf{ffv}(\Gamma),\gamma}$. We obtain $(\theta',\Delta') \sqsubseteq ((\Delta_1,\alpha_0 :: \varepsilon,\Delta_2),\theta)$, and we get $\vdash \theta'(\Gamma) \mid \Delta'$ by lemma 6.

Corollary 2 (Principality). *For any Γ, Δ_0, M, γ, if there exists a substitution $\Delta_0 \vdash \theta : \Delta$ such that $\theta(\Gamma) \mid \Delta \vdash M : \theta(\gamma)$, then $\Gamma; \Delta_0 \triangleright M : \gamma \Rrightarrow \exists\bar{\alpha}.\Delta' \wedge \theta'$ and $(\Delta',\theta') \sqsubseteq (\Delta,\theta)$.*

Proof. We first remark that we can assume $(\mathsf{dom}(\Delta_0) \cup \mathsf{ffv}(\Delta_0)) \subseteq \mathsf{ffv}_{\Delta_0}(\Gamma, \gamma)$, *i.e.* if Δ_0 contained variables that are not accessible from Γ and γ, then we could just rename them outside of $\mathsf{dom}(\theta)$ and $\mathsf{dom}(\Delta)$.

Since the canonical derivation may finish with S-BIND steps, we first remove them, and obtain $\theta(\Gamma) \mid \Delta, \Delta_1 \vdash M : \theta(\gamma)$. Since $\Delta_0 \vdash \theta : \Delta$ implies $\emptyset \vdash \theta : \Delta, \Delta_1$, we can use theorem to obtain $\Gamma \triangleright M : \gamma \Rrightarrow \exists \bar{\alpha}.\Delta_1' \wedge \theta_1'$ with $(\Delta_1', \theta_1')|_{\mathsf{ffv}(\Gamma),\gamma} \sqsubseteq ((\Delta, \Delta_1), \theta)$ and $\vdash \theta_1'(\Gamma) \mid \Delta_1'$. Due to the restriction, we can strengthen the second property into $(\Delta_1', \theta_1')|_{\mathsf{ffv}(\Gamma),\gamma} \sqsubseteq (\Delta, \theta)$, *i.e.* variables in $\mathsf{ffv}_{\Delta_1'}(\theta_1'(\Gamma), \theta_1'(\gamma))$, being accessible from Γ, γ, cannot be mapped into variables of Δ_1, which are not accessible from Γ, γ. Moreover $\Delta_0 \vdash \theta : \Delta$ implies $(\Delta_0, id) \sqsubseteq (\Delta, \theta)$, and from the remark above the non-filtered part of (Δ_1', θ_1') has no common variables with Δ_0, so we can conclude that $\Delta_0 \wedge (G \triangleright M : \gamma) \Rrightarrow \exists \bar{\alpha}.\Delta_2' \wedge \theta_2'$ with $(\Delta_2', \theta_2') \sqsubseteq (\Delta, \theta)$. The returned substitution is $(\Delta', \theta') = (\Delta_2', \theta_2')|_{\mathsf{ffv}(\Gamma,\Delta_0),\gamma}$. Again, this amounts to weakening the substitution, and we obtain $(\Delta', \theta') \sqsubseteq (\Delta, \theta)$.

The statement in mixed form that we presented at the beginning of this section can be obtained in the following way: for the mixed form typing problem $\Gamma \triangleright M : \zeta$, let $\Delta_0 = |\Gamma|$, then if $\lceil \Gamma \rceil_{\Delta_0}^{-1}; \Delta_0 \triangleright M : \lceil \zeta \rceil_{\Delta_0}^{-1}$ has a solution (Δ, θ), then there exists a principal solution.

5 Related works

While GADTs have been an active research area for about 10 years, early works on GADTs 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.

5.1 Comparison

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 [**?**] 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 [**?**].

Program	`f`	$\mathtt{f}_1$	$\mathtt{f}_2$	`g`	$\mathtt{g}_1$	$\mathtt{g}_2$	`p`	$\mathtt{p}_1$
Ambivalent	$\surd$	$\surd$	$\surd$	$-$	$\surd$	$\surd$	$\surd$	$-$
OutsideIn	$-$	$-$	$-$	$-$	$-$	$\surd$	$\surd$	$\surd$
Stratified	$-$	$\surd$	$\surd$	$-$	$-$	$\surd$	$-$	$-$

The results for `f` are unsurprising: this is the motivating example for introducing ambivalence, and it 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 bidirectional 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 corresponding type annotations large enough, so that we prefer to keep local polymorphism in OCaml [**?**].

[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.

Moreover, local polymorphism is critical to the annotation propagation mechanism used by OCaml, originally for polymorphic methods, and now for GADTs too.

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 [**?**], 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`.

5.2 Formalization techniques

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

Acknowledgments 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. P. Johann and N. Ghani. Foundations for structured programming with gadts. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 297–308, New York, NY, USA, 2008. ACM.
8. D. Le Botlan and D. Rémy. Recasting MLF. *Information and Computation*, 207(6):726–785, 2009.
9. 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.
10. 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.
11. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
12. 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.
13. 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.
14. 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.

15. 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.
16. 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.
17. 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.