Typechecking of Overloading

ARTHUR CHARGUÉRAUD, Inria & Université de Strasbourg, CNRS, ICube, France
MARTIN BODIN, Inria, France
JANA DUNFIELD, Queen's University, Canada
LOUIS RIBOULET, ENS Lyon, France, France

Overloading consists of using a same symbol to refer to several functions, or a same same to refer to several constants. Overloading is ubiquitous in mathematics. It also appears in numerous programming languages that resolve overloading statically, as opposed to languages that rely on dynamic dispatch during program execution. Thus, a key question is how to determine, for every occurrence of an overloaded symbol, which function it refers to. Static resolution of overloading is intrinsically intertwined with typechecking. Indeed, overloading resolution depends on types, but the types of the overloaded symbols depend on how they are resolved. This work presents the first typechecking algorithm for static resolution of overloading that: (1) guides resolution not only by function arguments but also by expected result type, and (2) supports polymorphic types. Moreover, our algorithm supports type inference like traditional ML typecheckers—we only exclude inference of polymorphism. We illustrate the practicality of our algorithm for typechecking conventional mathematical formulae, as well as for typechecking ML code with overloading of literals, functions, constructors, and record field names.

1 INTRODUCTION

1.1 Overloading in Programming Languages

In programming languages, overloading enables a programmer to reuse, at different types, the same mathematical operators, function names, method fields, and data constructor names. Arguably, the use of overloading can obfuscate the code slightly, because the programmer needs to resolve the symbols to know what they actually stand for. However, overloading greatly improves the conciseness and the readability of the code. For these reasons, many programming languages exploit overloading.

There are two main approaches to resolving overloading: dynamic resolution and static resolution. Consider an addition of two expressions, for example. With the dynamic approach, the runtime system first evaluates the two expressions to values, then, depending on the shape of these values, decide which addition operator is applicable. In contrast, this paper focuses on static resolution of overloading: the aim is to be able to tell, before the execution, just by inspecting the types, to which function every overloaded symbol corresponds to.

Several languages support static resolution of overloading. For example, C++ features function overloading [Dos Reis and Stroustrup 1985; Stroustrup 1984]. PVS [Shankar 1996] and ADA [Watt et al. 1987] support overloading not only of functions but also constants. OCaml does not support overloading for functions or constructors; it partially supports overloading of record and constructor names, yet their resolution is very fragile. Haskell provides a form of overloading via typeclasses, however typeclasses induce runtime overheads—one motivation for static resolution is precisely to avoid overheads and to enable further optimizations. The benefits of overloading in terms of conciseness can be visualized via the example shown below.

Authors' addresses: Arthur Charguéraud, Inria & Université de Strasbourg, CNRS, ICube, France, arthur.chargueraud@inria.fr; Martin Bodin, Inria, France, martin.bodin@inria.fr; Jana Dunfield, Queen's University, Kingston, Ontario, Canada, jd169@queensu.ca; Louis Riboulet, ENS Lyon, France, France, louis.riboulet@ens-lyon.fr.

2025. XXXX-XXXX/2025/1-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
(* Without overloading *)
(float_of_int (n + 1)) *. (3.0 *. pi / 4.0)
(* With overloading *)
(float_of_int (n + 1)) * (3 * pi / 4)

(* Without overloading *)
Array.iteri f (Array.map succ (Array.concat t (Array.of_list [2;3])))
(* With overloading *)
iteri f (map succ (concat t (to_array [2;3])))
```

Further in the paper, we also show examples highlighting the benefits of overloading data constructors and record field names.

1.2 Overloading in Mathematics

The practice of overloading has not been invented for programming languages. Indeed, mathematicians have exploited overloading essentially forever. For example, mathematicians use the symbol + to denote the addition operation regardless of the type of the addition. Only in case of high ambiguity is a type annotation used, e.g., $x +_{\mathbb{Z}} y$. The resolution of the type of a mathematical operator can be guided, in most cases, by the type of the arguments that the operator is applied to. For example, if x and y denote variables in $\mathbb{Z}$, then $x + y$ *resolves* to the addition operator from the mathematical structure $\mathbb{Z}$. Yet, in more complex examples, symbol resolution can be slightly less obvious.

For example, consider the following formula, assuming M and N to be two matrices over complex numbers

$$\sum_{d\in\{i,2i\}} \sum_{k\in[-6;7]} 3 \cdot e^{\frac{d\cdot\pi}{8}} \cdot M^{2\cdot k^2} \cdot N$$

Can you deduce the type of every operator and constant involved in the following formula? Can you present your reasoning steps in the form of an algorithm? Can you describe an efficient algorithm for resolving all the symbols in the formula?

It appears that every mathematician and, more generally, every user of mathematics implicitly have some form of algorithm for being able to resolve overloaded symbols. However, as surprizing as this might be given the importance of mathematics, the algorithm at play does not appear to have ever been made explicit! Proposing an algorithm able to resolve mathematical formulae as mathematicians conventionally write them is crucial for at least two applications.

The first application is mechanized mathematics, typically carried out in a proof assistant. There, overloading resolution would enable users to write formulae that follow standard mathematical practice. There have been attempts at supporting overloading by means of typeclasses or canonical structures. However, such encodings introduce a logical indirection that complicates proofs and gets in the way of rewriting operations.[1]

The second application—or rather, an important particular case of the first application—-is for writing program specifications, in the context of formal verification. Program specifications typically involve a fair number of mathematical facts. Such facts need to be parsed and processed in a way that leaves no ambiguity whatsoever. At the same time, specifications are meant to be readable by people who are not expert in formal methods. Hence, in the context of verifying a program in a particular domain of application, it is crucial for statements to be as close as possible to standard mathematics and to the standard notations used in that application domain.

[1]Technically, an instance `add (inst:=Z_add) x y` is convertible in Coq to `Z.add x y` but these terms are not syntactically equal, causing difficulties for matching and printing formulae.

1.3 Related Work and Contribution

The overloading resolution algorithm of C++ is probably the most well-known, and the most widely used. One fundamental limitation of this algorithm, though, is that functions are resolved based solely on the type of arguments. Resolution never depends on the expected result type. As a result, C++ does not support overloading resolution for constants. Yet, overloading constants is very useful: for example $\varnothing$ denotes the empty set, but also the empty multiset, the empty map, etc. Likewise, with overloading of constants, a literal such as `3` could be interpreted either as an integer or as a floating-point value depending on the context.

The programming language ADA and the prover PVS have both addressed the issue of resolving overloaded constants. They do so by means of a bidirectional typechecking algorithm. A bidirectional algorithm propagates type information both downwards—from the context to the subterms—and upwards—in the opposite direction. Concretely, an overloaded constant is resolved by the type expected by the context; and a function can be resolved based on both the type of its arguments and its expected return type.

The algorithms from ADA and PVS have two important limitations. First, they do not support polymorphism—they only support a form of functor construction. Second, they do not support local inference—all variables must be explicitly typed. Our work removes these two limitations.

Concretely, this paper presents the first typechecking algorithm that resolves overloaded symbols in the presence of polymorphism and local type inference. We have implemented our algorithm in an ML-style programming language, simply ruling out partial applications, which generally introduces too many ambiguities. We also leave aside the inference of polymorphism for the moment—maybe this feature can be added in the future, yet one might argue that explicit type quantifiers (like in Coq) make the code easier to read. Beyond programming languages, our algorithm can be applied to resolve overloaded symbols in formulae appearing in the context of mechanized mathematics or formal specifications.

Our prototype typechecker is implemented in OCaml.[2] The language it processes uses a syntax that closely resembles that of OCaml. Our typechecker can produce as output a program decorated with types, with every overloaded symbol decorated with the definition it refers to. Moreover, our prototype can produce as output an OCaml source file, obtained by replacing all overloaded symbols by the definitions they resolve to. This output file can be compiled and executed using the OCaml standard toolchain.

One feature that we do not yet support is the treatment of implicit coercions, which are supported for builtin types in C++, and supported for user-defined types in Coq. We leave their treatment to future work.

The paper starts by presenting the key ideas, then explain the typing rules for the core λ-calculus with overloaded symbols, and finally presents extensions to records, to data constructors and pattern matching, and to *derived* instances. A derived instance can be used to assert, e.g., that a `sum` operation is available for any data structure that features a `fold` operation and whose elements have a type that supports a `zero` and a `plus` operation.

2 OVERVIEW

2.1 Need for a Bidirectional Typechecker

Throughout the paper, we assume a context where two addition functions are available, one of type `int -> int -> int` and another of type `float -> float -> float`. Let us assume the underlying functions are built-in.

[2]Our prototype can be tested online: https://chargueraud.org/research/2025/overloading/proto.php.

```
external int_add : int -> int -> int = "%addint"
external float_add : float -> float -> float = "%addfloat"
```

In our prototype, we can register these two functions as *instances* of the plus symbol, via the following syntax.

```
let (+) = __instance int_add
let (+) = __instance float_add
```

We show below two occurrences of the addition operator that are resolved based on the type of the arguments.

```
let ex1 (x:int) (y:int) = x + y (* [+] resolves to [int_add] *)
let ex2 (x:float) (y:float) = x + y (* [+] resolves to [float_add] *)
```

Likewise, we would like to overload *constants*. We wish be able to write a constant, say 1, in the same way regardless of whether it is the unit value in $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ or $\mathbb{C}$. Unlike the resolution of an operator, whose resolution may be guided by the type of its arguments, the resolution of a constant must be guided by the type expected by its context.

```
let ex3 : int = 1 (* resolves [1:int] *)
let ex4 (x:int) = x + 1 (* resolves [1:int] *)
let ex5 (x:float) = 1 + x (* resolves [1:float] *)
```

When combining the use of overloaded operators and overloaded constants, one encounters situation where resolution requires propagation of type information in depth. The following example shows how an expected return type needs to be propagated downwards through operations until reaching the constants at the leaves.

```
let ex5 : float = (3 + 4) + (1 + (0 + 2)) (* resolves [1:float] *)
```

In general, propagation of type information actually needs to be *bidirectional*. In the example shown below, to realize that the operations at hand concern integer values, one needs to first investigate the subexpression `x + (0 + 2)`, exploit the fact that its left-hand side involves a value of type `int`, then needs to propagate this type information in depth in the subexpression `3 + 4`.

```
let ex6 (x:int) = (3 + 4) + (x + (0 + 2)) (* resolves [4:int] *)
```

Another example illustrating the need for bidirectional propagation appears next. In this example, the resolution of the constant `0` that appears in the then-branch exploits the type information inferred from deep in the else-branch.

```
let ex7 (x:float) =
  if x < 0 then 0 + 1 else 2 * x (* resolves [0:float] *)
```

2.2 A Bidirectional Typechecking Algorithm

The bidirectional algorithm that we propose makes two passes over the AST. The first pass consists of a recursive function that propagates the expected type, if it is available, downwards into the subterms. Moreover, via the result of the recursive calls, type information from the subterms is propagated upwards.[3] The second pass propagates expected type downwards into the subterms, a second time. However, this time the expected type could be more refined than in the first pass, thanks to information synthesized from other subterms during the first pass.

[3]Technically, the expected type provided as arguments is unified with the type of the term at hand, hence there is no need for the recursive function to return a type as result.

The types manipulated by the algorithm consist either of a conventional ML type, possibly a partially resolved type such as `list ?A`, or of a special type, written `Unresolved`. For example, if `x` has type `int`, then the expression `x` and the expression `x + 2` resolve to type `int`. On the contrary, the expression `0` and the expression `0 + 2`, when their expected return type is unspecified, are associated with the type `Unresolved`.

The type information acquired during the first pass may be exploited, during a second pass, to infer the type of subterms that were `Unresolved` after the first pass. For example, assume `x` has type `int`, and consider the typing of the expression `(0 + 2)+ x`. On the one hand, the subterm `0 + 2` has type `Unresolved`. On the other hand, `x` has type `int`. Thus, the addition at hand must be the one of type `int -> int -> int`. We deduce that `0 + 2` should be of type `int`. In the second pass, we propagate this information downwards into the subterms of `0 + 2`. We conclude that the constants `0` and `2` have type `int`.

After the 2 passes, we expect every subterm to be labelled with a type. In particular, we expect all overloaded functions (including constants) to be resolved. If an overloaded function remains unresolved, we reject the program.

The reader may ask why 2 passes and not 3 or more. Our rational is as follows. First, the practical code patterns that we have considered appear to all successfully typecheck using 2 passes. Second, as we illustrate in this paper, example programs that require more than 2 passes to typecheck appear to have an intristic complexity that makes them challenging for a programmer to mentally typecheck. Third, a smaller number of passes is beneficial for the efficiency of typechecking. That said, one possibility that we would like to explore is to execute, in case of remaining unresolved symbols after the second pass, additional passes. This way, in case of the typechecking ends up succeeding, we could report to the user a message indicating that the program provided is not ill-typed yet is missing a few type annotations to allow for faster typechecking.

We next focus in more details on two critical aspects of the algorithms. First, we explain which constructs make this expected return type available, and which constructs introduce subterms with an unknown expected return type. Second, we explain how to retain the ability to perform a significant amount of local type inference for local variables in the presence of overloaded functions.

2.3 Availability of the Expected Return Type

The first pass propagates downward the type expected for the term. There are essentially three ways by which the expected type can be determined.

- First, it may come from an explicit type annotation.
- Second, it may come from the control structure. For example, in a conditional of the form `if t0 then t1 else t2`, the term `t0` must have type `bool`.
- Third, the type of function arguments may be deduced from the type of the function being applied, when this function is resolved. For example, if there is a unique instance of `f` of type `int -> int`, then when typing the application `f 0`, the subterm `0` is known to be of type `int`. Similarly, the addition `((0 + 2): int)` can be resolved from the expected type `int`, thus the subterms `0` and `2` are resolved to be of type `int` during the first pass of the algorithm.

In contrast, there are several constructs for which the type of the subterms cannot be guessed immediately.

- First, consider a let-binding of the form `let x = t1 in t2`. No expected type is available for the first-pass typing of `t1`. As we shall see, the type of `x` may be inferred from the typing of `t1`. Alternatively, it may be inferred from the occurrences of `x` inside `t2`, in which case the type inferred for `x` is propagated into `t1` during the second pass. More generally, the type of t1 may also be inferred as a combination of the type of t1 and of the types imposed by the contexts associated with the occurrences of x.

- Second, consider a function call of the form `f t1`, and assume that the expected return type is not known. The resolution of the argument t1 must proceed in the first pass without an expected return type. The type inferred for t1 might help resolve f. If it does not suffice to resolve f in the call `f t1`, then the type returned for that call is `Unresolved`. In that case, an expected return type propagated during the second pass may help resolve the call.
- Third, consider a function call of the form `f t1`, and assume that the expected return type is known, but that there are several instances of f that are compatible with that return type. In this case again, t1 is typed without an expected return type. The type inferred for t1 should suffice to resolve f, otherwise f cannot be resolved and our algorithm rejects the program.

2.4 Local Type Inference in the Presence of Overloading

ML type inference offers a strong form of local type inference that, in particular, can infer the type of a local variable either based on its definition or based on its occurrences. Inferring the type of a variable from its occurrences is exploited in ML for example in a term of the form `fun x -> t1`, or in a term of the form `let x = ref [] in t1`. One strong benefit of local type inference is that it saves the need for most, if not all, type annotations. Yet, preserving a strong form of local inference in the presence of overloading can involve nontrivial flow of type propagation.

Consider the following example.

```
let ex8 =
  let x = 0 in
  let y = 1 in
  let z = x + y in
  (2 + x) + (3:int)
```

At first, the type of the variables x, y and z is `Unresolved`. On the last line, one can deduce via bidirectional typing that x admits type `int`. It follows the definition of x, i.e., the occurrence of 0, has type `int`. Moreover, it follows that the addition `x + y` has type `int`, because the first argument of this addition has type `int`. Hence, its second argument, namely the variable y, also admits type `int`. Finally, we deduce that the definition of y, i.e., the occurrence of 1, has type `int`.

Another interesting example involves a local function definition. Consider the definition exlet1 shown below, followed with exlet2 where the "plus 42" operator has been named as a local function.

```
let exlet1 (f:int->int) (g:int->int) (x:int) : int =
  f (x + 42) + g (2*x + 42)

let exlet2 (f:int->int) (g:int->int) (x:int) : int =
  let op = (fun n -> n + 42) in
  f (op x) + g (op (2*x))
```

Our algorithm is able to deduce, based on the calls to op that the + operation in the definition of op is an integer operation. More generally, our typing algorithm features the ability to type a local variable using information coming either from its definition or from its occurrences.

In summary, the interplay between overloading resolution and local type inference requires particular care with respect to the treatment of bound variables. Our bidirectional algorithm propagates type information in a specific manner during the two-pass process. Indirectly, via the type unifications performed, our algorithm gathers type constraints associated with the occurrences of variables. Thereby, our typing algorithm is able to handle idiomatic ML programming patterns without the need for type annotations on variables. Such a mixture of local inference and overloading resolution is not achieved by the existing algorithms implemented in PVS, ADA, and C++.

3 COMPLEXITY OF OVERLOADING RESOLUTION

Let us show that overloading resolution is NP-hard. We begin by presesnting the simplest proof, which consists of a reduction to a problem known as *positive one-in-three 3-SAT*. We then present a slighly more complex proof, which reduces to the more standard problem *3-SAT*. Both encodings involve two types, `int` and `float`, which are used to encode *true* and *false*, and involve a number of variables `xi`, each defined as the overloaded integer 0. This integer 0 can be interpreted at type `int` or `float`. For each occurence of 0, the choice of its type reflects a decision to interpret the variable `xi` as *true* or as *false*.

3.1 Reduction to Positive One-in-Three 3-SAT

Assume `0` to have two instances, of type `int` and `float`. Assume `f` be a unit function of three arguments, with three instances, each instance accepting one argument of type `int` and two other arguments of type `float`, in the three possible order.

```
__instance 0 : int
__instance 0 : float
__instance f : int -> float -> float -> unit
__instance f : float -> int -> float -> unit
__instance f : float -> float -> int -> unit
```

Now, consider the example program:

```
let x1 = 0 in
let x2 = 0 in
let x3 = 0 in
let x4 = 0 in
let x5 = 0 in
let x6 = 0 in
f x1 x3 x4;
f x1 x4 x5;
f x2 x3 x5;
f x2 x3 x6;
```

This program admits a resolution of overloaded symbol if and only if the boolean formula:

$$(x_1 \lor x_3 \lor x_4) \;\; \land \;\; (x_1 \lor x_4 \lor x_5) \;\; \land \;\; (x_2 \lor x_3 \lor x_5) \;\; \land \;\; (x_2 \lor x_3 \lor x_6)$$

admits an instantiation of the boolean variables x_i such that each of the conjuncts includes exactly one *true* variable (and thus two *false* variables). In general, this problem is known as *positive one-in-three 3-SAT*, which is known to be NP-hard.

Following the scheme presented above, it is clear that any formula of *positive one-in-three 3-SAT* can be encoded into a program that admits at one (or more) resolution of overloaded symbol if and only if the formula considered is satisfiable. Hence, overloading resolution is NP-hard.

3.2 Reduction to 3-SAT

By introducing just a few more instances, we can encode instances of *3-SAT*. We need a negation function, which converts `float` to `int` and vice-versa. We also need additional instances of our previous function `f`, to account for the cases where more than one disjunct is *true*.

```
__instance 0 : int
__instance 0 : float
__instance neg : float -> int
__instance neg : int -> float
__instance f : int -> float -> float -> unit
__instance f : float -> int -> float -> unit
__instance f : float -> float -> int -> unit
__instance f : int -> int -> float -> unit
__instance f : int -> float -> int -> unit
__instance f : float -> int -> int -> unit
__instance f : int -> int -> int -> unit
```

Now, consider the example program:

```
let x1 = 0 in
let x2 = 0 in
let x3 = 0 in
let x4 = 0 in
let x5 = 0 in
f x1 x3 (neg x4);
f x1 x4 (neg x5);
f x2 (neg x3) x5;
f (neg x2) x3 (neg x6);
```

This program admits a resolution of overloaded symbol if and only if the boolean formula:

$$(x_1 \lor x_3 \lor \neg x_4) \ \land \ (x_1 \lor x_4 \lor \neg x_5) \ \land \ (x_2 \lor \neg x_3 \lor x_5) \ \land \ (\neg x_2 \lor x_3 \lor \neg x_5)$$

admits an instantiation of the boolean variables x_i such that each conjunct evaluates to *true* (that is, each conjunct includes at least one *true* variable).

Following the scheme presented above, it is clear that any formula of *3-SAT* can be encoded into a program that admits at least one resolution of overloaded symbol if and only if the formula considered is satisfiable. This gives us a second proof that overloading resolution is NP-hard.

4 TYPECHECKING RULES

The *typechecking and symbol-resolution* process is thereafter refered to as *typechecking* for short. This typechecking process leverages *unification* steps. These steps are implemented using a mutable data structure for representing all the types involved, together with a recursive function called unify. This structure is standard to STLC and ML typechecking; We recall how it works in Section 4.1.

Our typechecking process then involves two phases. In the first phase, we perfom all the unification that captures the constraints of ML typechecking. For symbols that are not yet resolved, we simply introduce a not-yet-constrainted type. Then, in a second phase, we try to resolve overloaded symbols iteratively. If there exists one symbol for which exactly one of its possible instances would unify with type expected for this symbol by its context, then we can assign this symbol to this one instance.

The resolution of one overloaded symbol may introduce additional constraints, which materialize by additional type unifications. Hence, the resolution of one symbol may allow for the resolution of other symbols that could not be previously resolved. We iterate the resolution process until it converges. Two cases are possible. If all symbols are resolved, then the program is fully resolved and typechecked. Otherwise, the program is either ambiguous (i.e., several instantiations are possible),

```
type id = unit ref (* unique identifier for type variables and constructors *)

type typ = desc ref (* type representations are mutable *)
and desc =
  | Flexible (* type is not yet constrained *)
  | Unified of typ (* type is unified with another one *)
  | Constr of id * typ list (* structured type; e.g.:
    [Constr(id_arrow,[T1;T2])] represents the type [T1 -> T2];
    [Constr(id_int,[])] represents the constant type [int];
    [Constr(c,[])] represents a polymorphic type variable ['a] in the context of a
        type scheme where [c] is the identifier for that variable. *)

type sch = list id * typ (* type scheme, e.g. [forall 'a. 'a -> 'a] *)
(* where the [typ] in a [sch] does not contain any [Unified] or [Flexible]. *)
```

Fig. 1. Internal representation of types and type schemes. Our prototype includes additional field to store human-readable names for type variables, and to store marks used during cycle detection.

```
let rec unify (t1:typ) (t2:typ) : unit = (* may raise the [Failure] exception *)
  if !t1 != !t2 then
  match !t1, !t2 with
  | Unified t1', _ -> unify t1' t2
  | _, Unified t2' -> unify t1 t2'
  | Flexible, _ -> t1 := Unified t2
  | _, Flexible -> t2 := Unified t1
  | Constr(c1,ts1), Constr(c2,ts2) ->
      if c1 != c2 || List.length ts1 <> List.length ts2 then raise Failure;
      List.iter2 unify ts1 ts2
```

Fig. 2. Implementation of unification. Our implementation includes additional operations for path compression, cycle detection, support for backtracking, and support for informative error messages.

or the symbol resolution for this program is too complex to be achieved by simple deduction steps (i.e., additional type annotations would be needed).

4.1 Representation of Types

We begin by recalling the standard technique for representing types in a unification-based typechecker. We describe the core ideas by means of presenting a simplified OCaml implementation. We refer to our prototype implementation for the ingredients that are not described here. They include:

- **Path compression**: for improve performance, the unification function ought to compress the paths that it follows (like in the Union-Find data structure); our prototype uses a `get_repr` function to achieve this.
- **Efficient backtracking**: our typechecking algorithm involves *unification attempts*, which needs to be undone in case several instances successfully unify; our prototype records in a stack *checkpoints* and *update descriptions* to allow for efficiently reverting to a previous state.

```
(* Substitute bindings of [m] in a copy of [t] *)
let rec subst (m:(id*typ)list) (t:typ) : typ =
  match !t with
  | Flexible | Unified _ -> assert false
  | Constr(x,[]) when List.mem_assq x rho -> List.assq id m
  | Constr(c,ts) -> ref (Constr(c, List.map (subst m) ts))

(* Scheme instantiation, e.g. [forall 'a. 'a -> 'a] becomes
   [?t -> ?t] for a fresh flexible type [?t] *)
let instantiateScheme (s:sch) : unit =
  let (xs,t) = s in
  let m = List.map (fun x -> (x, ref Flexible)) xs in
  subst m t

let unifyScheme (s:sch) (t':typ) : unit =
  let t = instantiateScheme s in
  unify t t'

let checkScheme (s:sch) (t':typ) : unit =
  let (xs,t) = s in
  unify t t'
```

Fig. 3. Definitions of checkScheme(S, T') and unifyScheme(S, T'). The function checkScheme(S, T') checks that the type T' unifies with the type scheme S without constraining the polymorphic variables quantified by that type scheme. The function unifyScheme(S, T') unifies a type T with an *instantiation* of a type scheme S of the form $\forall \vec{X}. T$, that is, it invokes unify$(([\vec{A}/\vec{X}]\, T), T')$, where the types $\vec{A}$ denote fresh flexible types. The OCaml operations `List.mem_assq` and `List.assq` implement search in an association list by using physical equality for comparison—recall the identifiers have type "`unit ref`".

- **Cycle detection**: unless one is interested in supporting recursive types (as with OCaml's `-rectypes` compiler flag), one needs to ensure that no cycles are created in the graph-based representation of types.
- **Error reporting**: additional code is needed to report informative error messages to the user in case of unification failure; in particular, type variables need to preserve and introduce human-readable names for type variables.

Definition 4.1 (Grammar of ML types). ML typechecking involves types and type-schemes, characterized by the following grammar.

$$\begin{array}{llll} \text{ML types} & \tau & ::= & \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{int} \mid \mathsf{float} \mid \alpha \mid \tau \to \tau \mid C(\vec{\tau}) \\ \text{ML type schemes} & \sigma & ::= & \forall \vec{\alpha}.\, \tau \end{array}$$

Above, the form $C(\vec{\tau})$ corresponds to the application of a type constructor, e.g., `bool list` in OCaml. As we will see shortly afterwards, one can view the constant constructors (e.g. int) and type variables (written α above) as applications of constructors to zero arguments. Likewise, one can view the arrow type $\tau_1 \to \tau_2$ as a particular type constructor applied to the list made of τ_1 and τ_2.

Fig. 1 gives the internal representation of types that we use for typechecking. A type (type `typ`) may be refined during the unification process, hence it is described as a reference over a type description (type `desc`). The descriptions are as follows. A constant type, `int` is described as `Constr(id_int,[])`, where `id_int` is the unique identifier for the type `int`. Identifiers are here represented at type `unit ref`, which is a trivial implementation of unique identifiers. A type of the form $\tau_1 \longrightarrow \tau_2$ is described in the form `Constr(id_arrow,[T1;T2])`, that is, as the identifier for the arrow constructor paired with the list of two types `T1` and `T2`, which internalize τ_1 and τ_2, respectively.

If a type associated with a subterm is not yet constrained to any particular shape, then this type is described as `ref Flexible`, for a special *flexible* constructor. If a type `T1` has already been unified with another type `T2`, then the representation of `T1` is set to `ref (Unified T2)`.

A ML type scheme consists of a type with universally quantified variables. For example, the type of the identity function is $\forall A.\, A \longrightarrow A$, which is written `forall 'a. 'a -> 'a` in OCaml. The type `sch` denotes a type scheme. It consists of a list of identifiers and a type. In this type, a polymorphic variables with identifier `x` is described as a constant constructor `Constr(x,[])`. For example, the representation type $\forall A.\, A \longrightarrow A$ can be constructed as shown below.

```
(* internalization of [forall 'a. 'a -> 'a] *)
let a : id = ref () in
let a1 : typ = ref(Constr(a,[])) in
let a2 : typ = ref(Constr(a,[])) in (* variant: [let a2 = a1 in] *)
([a], ref (Constr(id_arrow,[a1; a2])))
```

More generally, any user-provided type annotation needs to be *interalized* as an object of type `typ`, so that this type annotation can be exploited during the typecheck process. The details of internalization may be found in our prototype.

Fig. 2 shows the implementation of the unification function, named `unify`, which takes two types `T1` and `T2`. The purpose of this function is: (1) to ensure that the representation of these two types do not conflict; and (2) to mutate the structure in such a way that any future refinement applied to `T1` will be also (implicitly) applied to `T2`, and vice-versa. The function is implemented recursively. All the `Unified` constructors are traversed. If a `Flexible` type is found, it is turned into a `Unified`. If two structured types are found, the function checks that the constructors match, then unify recursively the lists of type arguments. For example, to unify the types `T11->T12` with `T21->T22`, one recursively unifies `T11` with `T21` and `T12` with `T22`.

Fig. 3 gives the implementation of three functions that are meant to deal with ML type schemes.

The function `instantiateScheme` applies to a type scheme, and specializes it on a fresh types. For example, for the type scheme $\forall A.\, A \longrightarrow A$ the function `instantiateScheme` produces $T \longrightarrow T$ for an unconstrained type T.

The function `unifyScheme` is a shorthand that applies to a type scheme S and to a type T, and that unifies T with an instance of S. For example, given the type scheme $\forall A.\, A \longrightarrow A$ and a type $\text{int} \longrightarrow U$ for some unconstrained type U, the function `unifyScheme` unifies U as int. This function `unifyScheme` is exploited in particular for typechecking occurrence of variables, and for testing candidate instances of an overloaded symbol.

The function `checkScheme` also applies to a type scheme S and to a type T. Its purpose is to check that T is as general as S. For example, consider an identity function annotated with the type scheme $\forall A.\, A \longrightarrow A$. The type inferred for the implementation of this identity function is $T \longrightarrow T$, for an unconstrainted type T. In order to check that the implementation is as polymorphic as the type annotation claims, one needs not only to unify $T \longrightarrow T$ with $A \longrightarrow A$, but also needs to check that T was not previously unified with any other type. The function `checkScheme` implements this check by treating the type variable A as a (zero-arity) type contructor. Sucn a constructor only unifies with itself.

4.2 Declarative Typechecking

Definition 4.2 (Initial labelling with fresh flexible types). Given a program, we assign to every node and every binder of the AST a fresh type, that is, a value of type `typ` allocated as `ref Flexible`. As an exception, if a binder is already annotated with a type or a type scheme, we re-use the (internalized representation) of this type. Moreover, every variable occurrence is labelled with a unique identifier, written x_{id}. Identifiers is used for keep track of the values to which overloaded symbols are resolved. Programs labelled in such a way correspond to the following grammar.

Terms labelled with a type	t	$::=$	$u^{:T}$
Contents of a term	u	$::=$	$x_{\mathsf{id}} \mid v \mid t_1(t_2) \mid \mathsf{let}\, x^{:S} = t_1 \,\mathsf{in}\, t_2 \mid \lambda x^{:T}.\, t_1$
Literals	v	$::=$	$tt \mid b \mid n \mid d$

For literals, we here include the unit value, boolean values, integers, and floating-point numbers, which are used in our examples. Note that in a source programs, functions appear as terms; it is only during the runtime evaluation of a program that functions evaluate to closures, which belong to the rammar of runtime values.

Remark: a program may contain user-guided type annotations on arbitrary subterms, written in OCaml syntax as `(t : T)`. From the perspective of typechecking, this construct can be processed exactly as `let x : T = t in x`, which does fit the above grammar. Our implementation nevertheless features direct support for user-guided type annotations, in order to allow for more informative error messages.

Definition 4.3 (ML-Typechecking phase). Given a program where every AST node and every binder is initially annotated with a `Flexible` type, apply the rules given the table below to every AST node, in any order.

Subterm labelled with its type	Operations to apply
$(\mathsf{let}\, x^{:S} = {u_1}^{:T_1} \,\mathsf{in}\, {u_2}^{:T_2})^{:T}$	$\mathsf{checkScheme}(S, T_1)\,;\, \mathsf{unify}(T_2, T)$
$({u_0}^{:T_0}({u_1}^{:T_1}))^{:T}$	$\mathsf{unify}(T_0,\, T_1 \rightarrow T)$
$(\lambda x^{:T_0}.\, {u_1}^{:T_1})^{:T}$	$\mathsf{unify}(T,\, T_0 \rightarrow T_1)$
$v^{:T}$ where v is a literal of type T'	$\mathsf{unify}(T', T)$
$x_{\mathsf{id}}^{:T}$ if x is bound in scope to S	$\mathsf{unifyScheme}(S, T)$
$x_{\mathsf{id}}^{:T}$ if x is an overloaded symbol	*do nothing*

Definition 4.4 (Symbol resolution phase). Consider a program on which the ML-typechecking phase has already been applied. The occurrence of overloaded symbols may be resolved, in any order, by repeatedly applying the following rule as many times as possible. The rule captures the fact that exactly one of the possible instance has a type that would unify succesfully.

Subterm with its type	Conditions to satisfy	Operations to apply
$x_{\mathsf{id}}^{:T}$ where x_{id} is an occurrence of a not-yet-resolved overloaded symbol with possible instances $(v_1 : S_1), \dots, (v_n : S_n)$	$\mathsf{unifyScheme}(S_i, T)$ would succed $\wedge\, \forall j \neq i.\ \mathsf{unifyScheme}(S_j, T)$ would fail	record x_{id} resolved as v_i; $\mathsf{unifyScheme}(S_i, T)$

Definition 4.5 (Declarative typeckecking). To typecheck a program and resolves its symbols, apply the following steps:

(1) Decorate every node of the AST with a fresh flexible type, as descrribed in Definition 4.2.

(2) Perform unifications associated with ML-typechecking, as described in Definition 4.3.
(3) Iteratively apply the symbol resolution rule from Definition 4.4 as many times as possible.
(4) If all symbols are resolved, then typechecking is successful.

Actually, the substeps of (2) and (3) can be interleaved in any order.

4.3 Properties of Declarative Typechecking

We need to model the mutable state, which is implemented by means of references in Fig. 1.

Definition 4.6 (Mutable state for the typechecker). The state binds a type identifier T (a location of type `typ`) to its description D (a value of type `desc`). The state also maps certain variable identifiers to the values they resolve to, e.g. the identifier id of a variable x_{id} may be mapped to a value v.

$$\begin{array}{lrcl} \text{Type representation} & T & ::= & \textit{unique identifiers} \\ \text{Type description} & D & ::= & \mathsf{Flexible} \mid \mathsf{Unified}\, T \mid \mathsf{Constr}(C, \vec{T}) \\ \text{Type scheme} & S & ::= & \forall \vec{C}.\ T \\ \text{Mutable state} & s & ::= & \varnothing \mid s[T := D] \mid s[\mathrm{id} := v] \end{array}$$

After typechecking a program, certain type may remain unconstrained. For example, when typechking the program `let x = None in true`, `None` is assigned the type `T option`, where the type `T` remains unconstrained. At the end of the typechecking process, these unconstrained can be freely interpreted as any type. To simplify proofs, we choose to assign unconstrained type to the `unit` type.

Definition 4.7 (Interpretation of type representations). In a given state s, and in a context V that maps type constructor identifiers (written C) to ML type variable (written α), a type T is interpreted as an ML type $[\![T]\!]_s^V$, defined recursively as follows, using the operation $[\![D]\!]_s^V$ to interpret a type description. (The result is undefined in case of cycles.)

$$\begin{array}{lcl} [\![T]\!]_s^V & := & [\![s[T]]\!]_s^V \\ [\![\mathsf{Flexible}]\!]_s^V & := & \mathsf{unit} \\ [\![\mathsf{Unified}\, T']\!]_s^V & := & [\![T']\!]_s^V \\ [\![\mathsf{Constr}(C, [\,])]\!]_s^V & := & \alpha \qquad \text{if } V(C) = \alpha \\ [\![\mathsf{Constr}(C, [T_1, ..., T_n])]\!]_s^V & := & C([\,[\![T_1]\!]_s^V, ..., [\![T_n]\!]_s^V\,]) \\ [\![\forall \vec{C}.\ T]\!]_s^V & := & \forall \vec{\alpha}.\ [\![T]\!]_s^{(V[\vec{C}:=\vec{\alpha}])} \end{array}$$

Definition 4.8 (Interpretation of states). A state s that binds type identifiers (written T) to type descriptions (written D) can be interpreted as a map written $[\![s]\!]$ that maps the binds the same type identifiers to ML types (written τ).

$$[\![s]\!][T] \quad := \quad [\![s[T]]\!]_s^{\emptyset}$$

Definition 4.9 (Equivalent states). Two states s and s' are equivalent if and only if they have the same interpretation, that is, if and only if: $[\![s]\!] = [\![s']\!]$.

We next introduce judgments for every operation.

Definition 4.10 (Typing judgment for literals). $\boxed{\vdash_{\mathrm{lit}} v : T}$ assets that v is a literal of type T.

Using our formalism with explicit state, we revisit the unification and checking operations defined in Fig. 2 and Fig. 3.

Definition 4.11 (Unification operation).
$\boxed{s_1 \vdash \mathsf{unify}(T, T') \dashv s_2}$ describes the operation of executing $\mathsf{unify}(T, T')$ defined in Fig. 2 with the state evolving from s_1 to s_2.

Definition 4.12 (Scheme checking operation).

$\boxed{s_1 \vdash \mathsf{checkScheme}(S, T) \dashv s_2}$ describes the operation of executing $\mathsf{checkScheme}(S, T)$ defined in Fig. 3, with the state evolving from s_1 to s_2.

Definition 4.13 (Scheme unification operation).

$\boxed{s_1 \vdash \mathsf{unifyScheme}(S, T) \dashv s_2}$ describes the operation of executing $\mathsf{unifyScheme}(S, T)$ defined in Fig. 3, with the state evolving from s_1 to s_2.

We also revisit the symbol resolution operation that was previously described in Definition 4.4.

Definition 4.14 (Symbol resolution).

$\boxed{s_1 \vdash_{\mathsf{res}} (x_{\mathsf{id}}^{:T}) \in I \dashv s_2}$ The resolution of the symbol x_{id} of type T against a set of candidate instances I makes the state evolve from s_1 to s_2.

The judgment is defined by a rule that handles the resolution (a single candidate instance unifies), and a rule that reduces the set of candidate instances (more than one candidate instances unify). If zero candidate instances unify, the algorithm is stuck—in the implementation, an exception is raised.

$$\frac{\begin{array}{c} I = (v_1 : S_1), \ldots, (v_n : S_n) \qquad s \vdash \mathsf{unifyScheme}(S_i, T) \dashv s_i \\ \forall j \neq i.\ \neg \left(\exists s_j.\ s \vdash \mathsf{unifyScheme}(S_j, T) \dashv s_j\right) \end{array}}{s \vdash_{\mathsf{res}} (x_{\mathsf{id}}^{:T}) \in I \dashv (s_i[\mathsf{id} := v_i])} \ \textsc{ResUnique}$$

$$\frac{\begin{array}{c} I = (v_1 : S_1), \ldots, (v_n : S_n) \\ I' = \{(v_i : S_i) \in I \ \mid\ \exists s_i.\ s \vdash \mathsf{unifyScheme}(S_i, T) \dashv s_i\} \qquad |I'| \geq 2 \end{array}}{s \vdash_{\mathsf{res}} (x_{\mathsf{id}}^{:T}) \in I \dashv (s[\mathsf{id} := I'])} \ \textsc{ResMultiple}$$

The commutativity lemma justifies that all the ML typechecking steps (substeps of (2) in Definition 4.5) can be performed in any order.

Lemma 4.15 (Commutativity of unifications). *If two unification operations succeed, then swapping their execution leads to an equivalent state.*

TODO. □

The monotocity lemmas, together with commutativity, justifies the ability to interleave resolution steps (substeps of (3) in Definition 4.5) with the ML typechecking steps (substeps of (2) in Definition 4.5).

Lemma 4.16 (Monotonicity of unification).

- **Monotonicity of unification successes***: if a unification succeeds when performed before another unification, it would also succeed if performed after that other unification.*
- **Monotonicity of unification failures***: if a unification fails when performed before another unification, it would also fail if performed after that other unification.*
- **Successful unifications can be anticipated***: if a unification succeeds when performed after another unification, it would also succeed if performed before that other unification.*

TODO. □

Note that if two unifications are incompatible, the second unification being performed would fail whereas the first one being performed could succeed.

Lemma 4.17 (Monotonicity of resolution).

- **Monotonicity of resolutions**: *if a successful resolution is followed by a successful unification, then this unification can be performed before the resolution, leading to equivalent states.*
- **Resolutions can be anticipated if they succeed**: *if a resolution succeeds after a successful unification, and if the resolution succeeds when executed before that unification, then the two operations can be swapped, leading to equivalent states.*

TODO. □

Definition 4.18 (Definition of ML typing rules). TODO: recall the standard typing rules of ML, on our grammar. Use Γ to denote a ML typing environment, bindings x to σ.

Theorem 4.19 (Type soundness). *If a program successfully typechecks against our rules, then the extracted program where overloaded symbols are replaced with the values they resolved to is well-typed against the standard ML typing rules.*

Proof. The unifications that are performed (and not backtracked) during our full typechecking process are the same as those performed during ML typechecking. Overloaded symbols are replaced with values that are typechecked after the other nodes, but this has no incidence. Indeed, recall from standard typechecking theory that the order in which unifications are performed does not matter. □

Theorem 4.20 (Non-ambiguity). *If a program successfully typechecks against our rules, then no other instantiation of the set of overloaded symbols could lead to a well-typed ML program.*

Proof. Assume the contrary. Consider the set of symbols that are resolved differently in the two instantiations. Consider the first of these symbols that has been resolved by our algorithm. For this symbol, more than one instantiation is possible, hence the rule of Definition 4.4 does not apply. Contradiction. □

Theorem 4.21 (Characterization of ill-typed programs). *If a program does not typecheck against our rules, then:*

- *either no instantiation of overloaded symbols make the program well-typed in ML;*
- *or there are more than one instantiation that makes the program well-typed in ML; in this case, disambiguation can be achieved by adding type annotations;*
- *or there is exactly only instantiation that makes the program well-typed in ML, yet this instantiation cannot be deduced by applying a series of simple deduction steps (in the sense of applying a step of Definition 4.4 to one of the remaining unresolved symbols); in this case, resolution can be guided by adding type annotations.*

Proof. Let N be the number of instantiations of the set of overloaded symbols that make the program well-typed. The three cases correspond to $N = 0$, $N > 1$ and $N = 1$, respectively. □

5 TYPECHECKING ALGORITHM

The previous section describes rules for typechecking a program, but does not specify the order in which to apply the rules. Each succesful resolution of a symbol may unlock the ability to resolve on other symbol. Hence, in the worst case, we may need a quadratic number of resolution attempts.

In this section, we present an algorithm for applying rules in a specific order. This algorithm attempts the resolution of a given symbol at most three times. Moreover, it requires only two traversals of the AST describing the program.

The proposed algorithm is incomplete: it typechecks fewer program than the set of rules presented earlier. However, as we have argued, a practical solution is necessarily incomplete because overloading resolution is NP-hard. Besides, as we will argue by means of examples, the 2-pass algorithm that we propose suffices to resolve overloaded symbols in many common programming patterns.

We begin with an informal presentation, which specifies our algorithm in an accessible manner, following the presentation style of certain language standards, e.g., that ECMAScript for the JavaScript language. We then formalize the algorithm through a set of inference rules, making the mutable state explicit.

5.1 Informal Presentation

We assume the input program to have its node labelled as described in Section 4.2. We describe how to process the AST in two passes, including the work associated with structural ML typechecking as described in 4.3.

The algorithm is described as a recursive function. This function threads an *environment*, which binds variables to a type scheme, or bind them to a list of instances in case of an overloaded symbol.

Typechecking of Literals. To typecheck a literal v labelled with a type T, written $v^{:T}$, it suffices to unify T with the type of that literal. For example, for the boolean constant `true`, unify T with bool.

Typechecking of Regular Variables. To typecheck a variable x labelled with a type T, written $x^{:T}$, the first step is to look up the type S associated to x in the environment. The, the type T is unified with a fresh instance of S, by means of executing $\mathsf{unifyScheme}(S, T)$.

Typechecking of Overloaded Symbols. We next explain how to typecheck an overloaded symbol $x_{\mathrm{id}}^{:T}$. During the first pass, the symbol may remain unresolved, however after the second pass it must be resolved. A resolution attempts proceed as follows.

(1) Consider the set of instances associated with the overloaded symbol x. The i-th instance consists of a value v_i of type S_i.
(2) For each S_i, test whether T could unify with S_i, by evaluating $\mathsf{unifyScheme}(S_i, T)$. Keep the boolean result of the test, but undo all the side-effects performed during this process.
(3) Count for how many indices i the unifications have succeeded.
    - If none of the instance unify, raise the exception `NoInstanceMatch`.
    - If exactly one instance S_i unifies with T, then record x_{id} as resolved to the value v_i. Evaluate again $\mathsf{unifyScheme}(S_i, T)$, this time keeping the side-effects.
    - If several instances could unify, then they are two cases. If on the first pass, then do nothing. If on the second pass, then raise the exception `MultipleInstancesMatch`.

Optimization: during the first pass, if more than one instance unifies, then we can store the subset of instances that could unify. It suffices to consider this subset during the second pass. In other work, we may save work by definitely ruling out instances that already do not unify during the first pass.

Typechecking of Function Calls. In what follows, we explain how to typecheck a term of the form $t_0(t_1, .., t_n)$, labelled with a type T. Let T_i denote the type of the subterm t_i, for each i.

First pass.

(1) Unify the type T_0 with $T_1 \rightarrow .. \rightarrow T_n \rightarrow T$.
(2) Recursively typecheck the function t_0.
(3) Recursively typecheck the arguments $t_1, ..., t_n$.
(4) If t_0 is an unresolved overload symbol, recursively typecheck the term t_0 once again.

Second pass.

(1) Recursively typecheck the function t_0. If t_0 is an unresolved overload symbol, then at this point it must have been resolved.
(2) Recursively typecheck the arguments $t_1, \ldots, t_n$.

Typechecking of Let-Bindings. Consider the typechecking of a term of the form $\mathsf{let}\ x^{:S} = t_1\ \mathsf{in}\ t_2$ labelled with type T.

First pass.

(1) Unify the type T_1 of t_1 with a fresh instance of S.
(2) Unify the type T_2 of t_2 with the type T of the whole let-binding.
(3) Recursively typecheck t_1.
(4) In an environment extended with a binding from x to S, recursively typecheck t_2.

Second pass.

(1) In an environment extended with a binding from x to S, recursively typecheck t_2.
(2) Recursively typecheck t_1.

5.2 Formal Presentation of Algorithmic Typechecking

First, we model the typechecking environment, which keeps track of what variables are bound to—previously, the environment was left implicit.

Definition 5.1 (Typing environments). An *environment* binds variables as either *regular variables* that admit a type scheme S, or as *overloaded symbols* for which there is a set of possible instances I.

Set of instances	$I ::= \epsilon \mid I, (v : S)$
Typing environment	$E ::= \emptyset \mid E,\ x : \mathsf{VarRegular}(S) \mid E,\ x : \mathsf{VarOverloaded}(I)$

Definition 5.2 (Mutable state for the algorithmic typechecker). Recall that the state map variable identifiers to the values they resolve to, e.g. the identifier id of a variable x_{id} may be mapped to a value v. For algorithmic typechecking, a variable identifier may also be bound to a subset I describing *remaining possible instances.*

$$\text{Mutable state, extended} \quad s ::= \varnothing \mid s[T := D] \mid s[\mathsf{id} := v] \mid s[\mathsf{id} := I]$$

Let us summarize the possible cases for a variable x_{id} occurring in the program being typechecked.

- The variable is bound as a regular variable with an entry of the form $x : \mathsf{VarRegular}(S)$ in the environment in which the variable x_{id} is typechecked.
- The variable is bound as an overloaded symbol with an entry of the form $x : \mathsf{VarOverloaded}(I)$ in the environment in which the variable x_{id} is typechecked. In that case, the current state s provides information about the status of the resolution of x_{id}.
  - If not resolution attempt has yet been performed on x_{id}, then the identifier id is not bound in the state s.
  - If x_{id} has been successfully resolved to a value v among a set of possible instances I, then the state s binds id that this value v.
  - If a resolution attempt has reduced from I to I' the set of candidate instances for x_{id}, then the state s binds id to this subset I', whose cardinality is necessarily greater than one.

Definition 5.3 (First pass).

$\boxed{E; s_1 \vdash_{\mathsf{fst}} t \dashv s_2}$ Under environment E and input state s_1, our first typechecking pass over t makes the state evolves to s_2.

The judgment is defined by the rules from Fig. 4.

$$\frac{\vdash_{\mathsf{lit}} v : T' \qquad s_1 \vdash \mathsf{unify}(T', T) \dashv s_2}{E; s_1 \vdash_{\mathsf{fst}} v^{:T} \dashv s_2}\ \textsc{FstLiteral}$$

$$\frac{E(x) = \mathsf{VarRegular}(S) \qquad s_1 \vdash \mathsf{unifyScheme}(S, T) \dashv s_2}{E; s_1 \vdash_{\mathsf{fst}} x_{\mathsf{id}}^{:T} \dashv s_2}\ \textsc{FstVarRegular}$$

$$\frac{E(x) = \mathsf{VarOverloaded}(I) \qquad s_1 \vdash_{\mathsf{res}} (x_{\mathsf{id}}^{:T}) \in I \dashv s_2}{E; s_1 \vdash_{\mathsf{fst}} x_{\mathsf{id}}^{:T} \dashv s_2}\ \textsc{FstVarOverloaded}$$

$$\frac{\begin{array}{c} s_1 \vdash \mathsf{checkScheme}(S, T_1) \dashv s_2 \qquad s_2 \vdash \mathsf{unify}(T_2, T) \dashv s_3 \\ E; s_3 \vdash_{\mathsf{fst}} ({u_1}^{:T_1}) \dashv s_4 \qquad E, x : \mathsf{VarRegular}(T_1);\ s_4 \vdash_{\mathsf{fst}} ({u_2}^{:T_2}) \dashv s_5 \end{array}}{E; s_1 \vdash_{\mathsf{fst}} (\mathsf{let}\, x^{:S} = ({u_1}^{:T_1})\ \mathsf{in}\ ({u_2}^{:T_2}))^{:T} \dashv s_5}\ \textsc{FstLet}$$

$$\frac{s_1 \vdash \mathsf{unify}(T,\ T_0 \to T_1) \dashv s_2 \qquad E, x : \mathsf{VarRegular}(T_0);\ s_2 \vdash_{\mathsf{fst}} ({u_1}^{:T_1}) \dashv s_3}{E; s_1 \vdash_{\mathsf{fst}} (\lambda x^{:T_0}.\, {u_1}^{:T_1})^{:T} \dashv s_3}\ \textsc{FstLam}$$

$$\frac{\begin{array}{c} s_1 \vdash \mathsf{unify}(T_0,\ T_1 \to T) \dashv s_2 \qquad E; s_2 \vdash_{\mathsf{fst}} ({u_0}^{:T_0}) \dashv s_3 \qquad E; s_3 \vdash_{\mathsf{fst}} ({u_1}^{:T_1}) \dashv s_4 \\ \forall x. \forall \mathsf{id}. \forall I'.\ \neg \left(u_0 = x_{\mathsf{id}}\ \wedge\ s_4[\mathsf{id}] = I'\right) \end{array}}{E; s_1 \vdash_{\mathsf{fst}} ({u_0}^{:T_0}({u_1}^{:T_1}))^{:T} \dashv s_4}\ \textsc{FstApp1}$$

$$\frac{\begin{array}{c} s_1 \vdash \mathsf{unify}(T_0,\ T_1 \to T) \dashv s_2 \qquad E; s_2 \vdash_{\mathsf{fst}} ({u_0}^{:T_0}) \dashv s_3 \qquad E; s_3 \vdash_{\mathsf{fst}} ({u_1}^{:T_1}) \dashv s_4 \\ u_0 = x_{\mathsf{id}} \qquad s_4[\mathsf{id}] = I' \qquad s_4 \vdash_{\mathsf{res}} (x_{\mathsf{id}}^{:T_0}) \in I' \dashv s_5 \end{array}}{E; s_1 \vdash_{\mathsf{fst}} ({u_0}^{:T_0}({u_1}^{:T_1}))^{:T} \dashv s_5}\ \textsc{FstApp2}$$

Fig. 4. Rules for the first typechecking pass.

These rules combine the operations from the ML-typechecking phase (Definition 4.3) and the rules from declarative symbol resolution (Definition 4.4). Resolutions are first performed on the way down through the AST. Then, for unresolved functions, an additional resolution attempt is performed on the way back up from the recursion.

Definition 5.4 (Second pass).

$\boxed{s_1 \vdash_{\mathsf{snd}} t \dashv s_2}$ In input state s_1, our second typechecking pass over t makes the state evolves to s_2.

The judgment is defined by the rules from Fig. 5.

There are two key aspects to this second pass. First, in the rule SndLet, the subterm t_2 is processed before t_1. Second, in the rule SndVarForceResolve, the resolution is performed for variables that were previously unresolved, using the set of remaining candidates that was stored during the first pass. For a overloaded function name, whose resolution may be attempted twice during the first phase, it means that a third and last attempt may be performed during the second phase.

Definition 5.5 (Two-pass algorithm). A closed program t is typechecked in an empty environment,and in a state initialized with instances for overloaded symbols. The initial mutable state s_0 is obtained by internalizing all the types at hand: built-in types, types associated with instances, and types from user-provided annotatons.

$$\frac{}{s \vdash_{\mathsf{snd}} v^{:T} \dashv s}\ \textsc{SndLiteral} \qquad \frac{\mathsf{id} \notin \mathrm{dom}\, s}{s \vdash_{\mathsf{snd}} x_{\mathsf{id}}^{:T} \dashv s}\ \textsc{SndVarRegular} \qquad \frac{s[\mathsf{id}] = v}{s \vdash_{\mathsf{snd}} x_{\mathsf{id}}^{:T} \dashv s}\ \textsc{SndVarResolved}$$

$$\frac{s_1[\mathsf{id}] = I' \qquad s_1 \vdash_{\mathsf{res}} (x_{\mathsf{id}}^{:T}) \in I' \dashv s_2 \qquad s_2[\mathsf{id}] = v}{s_1 \vdash_{\mathsf{snd}} x_{\mathsf{id}}^{:T} \dashv s_2}\ \textsc{SndVarForceResolve}$$

$$\frac{s_1 \vdash_{\mathsf{snd}} t_2 \dashv s_2 \qquad s_2 \vdash_{\mathsf{snd}} t_1 \dashv s_3}{s_1 \vdash_{\mathsf{snd}} (\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2)^{:T} \dashv s_3}\ \textsc{SndLet} \qquad \frac{s_1 \vdash_{\mathsf{snd}} t_1 \dashv s_2}{s_1 \vdash_{\mathsf{snd}} (\lambda x^{:T_0}.\, t_1)^{:T} \dashv s_2}\ \textsc{SndLam}$$

$$\frac{s_1 \vdash_{\mathsf{snd}} t_0 \dashv s_2 \qquad s_2 \vdash_{\mathsf{snd}} t_1 \dashv s_3}{s_1 \vdash_{\mathsf{snd}} (t_0(t_1))^{:T} \dashv s_3}\ \textsc{SndApp}$$

Fig. 5. Rules for the second typechecking pass.

In that environment and in that state, the AST of the program is traversed by the first pass, described by the judgment $\emptyset; s_0 \vdash_{\mathsf{fst}} t \dashv s_1$. Then, it is traversed by the second pass, described by the judgment $s_1 \vdash_{\mathsf{snd}} t \dashv s_2$. If these two judgments hold, typechecking is successful.

A variable occurrence with unique identifier `id` such that $s_2[\mathsf{id}] = v$ is resolved to the value v. A variable occurrence with unique identifier `id` such that $\mathsf{id} \notin \mathrm{dom}\, s_2$ corresponds to a regular variable, i.e., bound by a let-binding or a lambda-abstraction.

If the algorithm is interrupted due to the last premise of the rule SndVarForceResolve, namely $s_2[\mathsf{id}] = v$, being not satisfied, then we can report a lack of information for the resolution of the variable with identifier id.

Theorem 5.6 (Inclusion of algorithmic typechecking into declarative typechecking). *If a program typechecks against our two-pass algorithm 5.5, then it typechecks against our declarative typechecking from Definition 4.5.*

Proof. All unifications performed by the two-pass algorithm can be performed as steps of the declarative typechecking. □

Theorem 5.7 (Soundness of algorithmic typechecking into declarative typechecking). *If a program typechecks against our two-pass algorithm 5.5, then it typechecks against our declarative typechecking from Definition 4.5.*

Proof. Follows from 5.6 and ??. □

Definition 5.8 (Two-pass algorithm with error-reporting). A variant algorithm for improved error-reporting is defined as follows.

(1) Execute a modified version of the two-pass algorithm where the rule SndVarForceResolve is replaced with a variant that does not include its third premise, that is, where resolution is not forced to succeed. Let s' be the final state obtained.
(2) Consider the set of unresolved symbols. A unique identifier id is unresolved in the final state s' if and only if $s'[\mathsf{id}] = I'$ for some I'. If all symbols are resolved, then typechecking is successful. Otherwise, apply the following steps.
(3) For every unresolved symbol, test whether it could be resolved using the symbol-resolution rule of Definition 4.4. (Testing can be interrupted after the first successful resolution.)

- If resolution succeeds for at least one symbol, let id be the identifier of the first (or any) of these symbols. Report the message: *The symbol id could be resolved but only with more than two passes, so please provide additional type annotations.*
- Othewise, if resolution does not succeed for any symbol, let id be the identifier of the first (or any) of these symbols. Report the message: *The symbol id could not be resolved, so please provide additional type annotations.*

6 OVERLOADED RECORD FIELDS

From the perspective of typechecking, we view all record operations as function calls—like it is done in Coq for example. Overloaded record fields thus give rise to overloaded functions. In what follows, we present our encodings, first at a high level, then in more details.

Summary of our encodings. Consider the following type definition. We assume that fields are sorted alphabetically before typechecking begins.

```
type t = { mutable f : int; mutable g : int }
```

Our encodings can be summarized as follows.

```
r.f                          __get_f r
r.f <- 3                     __set_f r 3
{ f = 3; g = 4 }             __make_f_g 3 4
{ r with f = 3 }             __with_f r 3
{ r with f = 3; g = 4 }      __with_g (__with_f r 3) 4
```

These encodings are exploited for the purpose of the typechecking only. After resolution of overloaded fields, our prototype output an OCaml source code where record fields are renamed in an unambiguous manner. For example, the field `f` of the type `t` is renamed into `t_f`, and an access of the form `r.f` with an expression `r` of type `t` becomes `r.t_f`.

In what follows, we explain the details of the encodings.

Encoding of get operations. For each field, we introduce an overloaded getter function. Consider for example the field `f` of type `int` in the type `t`. We introduce a special function `__get_f` and provide an instance of `__get_f` of type `t -> int`. We then encode the expression `r.f` as the function call `__get_f r`.

Encoding of set operations. Similarly, we introduce a special function `__set_f`, and introduce an instance of `__set_f` of type `t -> int -> unit`. We then encode the expression `r.f <- v` as the function call `__set_f r v`.

Encoding of record construction. For a record type featuring two fields named `f` and `g`, we introduce an overloaded function named `__make_f_g`. Recall that we assume field names to be sorted alphabetically. The expression `{f = 3; g = 4}` is interpreted as `__make_f_g 3 4`. In practice, an instance of `__make_f_g` can be resolved in different ways: the type expected by the context might disambiguate; else there might be only one record definition featuring exactly the two fields `f` and `g`; else, there might be only one matches the types of the arguments provided; else, the user would need to add a type annotation to force an expected result type.

Encoding of record update. The `with`-construct with multiple updated fields is treated by the typechecker as nested unary `with`-constructs. For example:

```
{ r with f = 3; g = 4 } (* is encoded as *) { { r with f = 3 } with g = 4 }.
```

There remains to explain the encoding for updating one field. Consider the field `f` of the type `t`. We introduce a function named `__with_f` of type `t -> int -> t`. Then, we encode `r with f = 3` as the function call `__with_f r 3`.

Encoding for polymorphic record types. For a polymorphic record type definition, the instances introduced simply consist of polymorphic functions. For example, consider the definition.

```
type 'a cell = { hd : 'a; tl : 'a list }
```

The instance of `__get_hd` provided for reading the first field has type `'a cell -> 'a`.

Advanced examples. Consider the following examples.

```
type t = { f : int; mutable g : int }
type u = { f : int; mutable g : float }
type v = { f : int; mutable g : float; h : bool }
```

The code snippets shown below illustrate the resolution at play on several examples exploiting the types `t`, `u` and `v` defined above.

```
let r1 (r:t) = r.f (* resolves [f] to be a field of [t] *)
let r2 : t = { f = 3; g = 2 } (* [2] resolves as [int] *)
let r3 = { f = 3; g = (2:float) } (* resolves [r3] to [u] *)
let r4 = { f = 3; g = 2; h = true } (* resolves [r4] to [v] *)
let r5 = r2.g <- 2 (* [r2] has type [t], thus [2] resolves to [int] *)
let r6 = { r2 with g = 2 } (* [r2] has type [t], thus [2] resolves to [int] *)
let r7 = { f = 2; g = 3 } (* rejected: ambiguous *)
```

7 OVERLOADED DATA CONSTRUCTORS

Motivating example. To illustrate the interest of overloaded constructors, consider the following definitions describing the grammar of two toy programming languages.

```
type t = Var of var | Let of var * t * t | Load of t
type u = Var of var | Let of var * u * u | Load of var
```

The function shown below takes a program from the grammar `t` into one from the grammar `u`, by assigning a name via a let-binding to sub-expressions that appear in load instructions. Observe how the same constructor names can be used in both embedded languages.

```
let rec norm (e:t) : u =
  match e with
  | Var x -> Var x
  | Let (x, t1, t2) -> Let (x, norm t1, norm t2)
  | Load t1 ->
      match t1 with
      | Var x -> Load x
      | _ -> let x = generate_var_fresh_from t1 in
              Let (x, norm t1, Load x)
```

The point of this example is to illustrate how constructors are resolved. The pattern matching construct filters an argument `e` of type `t`, thus the constructors are resolved at type `t`. In the branches, whose expected return type is annotated to be `u`, the constructors are resolved at type `u`.

Typechecking of pattern matching. Here again, we view constructors are functions, and resolve overloading of constructor applications using the standard mechanism for functions—like Coq does. Moreover, we perform disambiguation of constructors inside patterns.

Among other properties, we wish the following simple pattern matching expression:

```
match t0 with x -> t1
```

to be typechecked in a totally equivalent manner as the corresponding let-binding (as it is the case in OCaml):

```
let x = t0 in t1
```

To achieve this equivalence, we need to typecheck first `t0` then `t1` in the first pass, then typecheck `t1` then `t0` in the second pass. (Recall Section 2.2.) The above observation gives the skeleton of the typechecking process in case there is a single branch with a trivial pattern. We handle the general case as described next.

To help the description, consider a representative example.

```
match t0 with
| p1 -> t1
| p2 -> t2
```

To typecheck such an expression with an expected result type `T`, we proceed as follows.

(1) Typecheck the scrutiny `t0`, obtain a type `T_0`.
(2) Typecheck the patterns `p1` and `p2`, with expected type `T_0`.
(3) Typecheck the continuations `t1` and `t2`, with expected type `T`.
(4) Typecheck again the continuations `t1` and `t2`, with expected type `T`.
(5) Typecheck again the patterns `p1` and `p2`, with expected type `T_0`.
(6) Typecheck again the scrutiny `t0`.

There is an interesting bidirectional flow of type information. The scrutiny may propagate information, through the patterns, to the variables that are bound in the branches. Reciprocally, the branches may refine the types of the variables, which may help resolve the pattern constructors, and ultimately refine the type of the scrutiny. Besides, type information may flow across the various branches, both for the type `T_0` and for the type `T`. For `T_0`, the resolution of any of the patterns during the first pass generally suffices to resolve the constructors in all the other patterns. For `T`, the resolution of the type of any of the branches during the first pass suffices to provide information for typechecking all the other branches with an expected return type.

Advanced examples. To illustrate complex flows of type information, consider the following examples.

```
type t = A of t | B of int | C of int
type u = A of u | B of float

let f v =
  match v with
  | A _ -> ()
  | B _ -> ()
  | C _ -> () (* the 1st traversal of this pattern forces [v:t] *)

let g v =
  match v with
  | A (B x) -> ()
  | A (B x) -> ignore (x:int) (* the 2nd traversal of this pattern gives [v:t] *)
  | _ -> ()
```

Examples requiring more than 2 passes. Here again, there exists pattern matching that resolve to exactly one type, yet for which more than two passes would be necessary to propagate sufficient information. We make a deliberate choice of limiting the number of phases, both to ensure efficiency and to allow predictability by the programmer.

The following counter-example reuses the above type definitions of `t` and `u`. It does not typecheck in our system. Indeed, 3 passes would be needed to resolve the type of the constructor `A`: a first pass to propagate the type of x from the branch into the pattern, a second pass to propagate the type of x from the pattern variable to the type of `P`, and a third pass to propagate the type of `P` down onto the constructor `A`.

```
type 'a p = P of 'a * 'a

let h v =
  match v with
  | P (A y, B x) -> (x:int)
```

Arguably, in the example above, the mental work involved for resolving the type of the constructor `A` is nontrivial. In practice, it is generally not hard for the programmer to add just one type annotation is the right place to ease resolution significantly. For example, if we remove the type annotation `(x : int)` that appears in the continuation, and if we add a type annotation `(v : int p)` on the argument `v` of the function `h`, we would allow all constructors to be trivially resolved on the first pass.

8 TREATMENT OF POLYMORPHIC HIGHER-ORDER ITERATORS

Consider the mathematical expression $\sum_{x \in E}(x+1)$. If the variable E denotes a set of real numbers, then the variable x obviously stands for a real number, hence the $x+1$ operation is on real numbers. More generally, when we have a container data structure at hand (e.g., a list, a set, a map, etc.), we expect to know the type of its elements. If we iterate over that container, the iteration operation over this container should be resolved guided by the type of the container, and the type of the variable that denotes an element should be deduced from the type of the elements of that container. Our aim is to translate this intuitive recipe into our typing algorithm.

Assume two instances of the `map` function, one for lists and one for arrays.

```
val List.map : 'a list -> ('a -> 'b) -> 'b list
val Array.map : 'a array -> ('a -> 'b) -> 'b array
```

In our prototype implementation, the syntax for registering instances is as follows.

```
let map = __instance Array.map
let map = __instance List.map
```

Consider a container `d` defined as a list of floating-point values, and an operation that invokes `map` over `d` to add one unit to every value in the list.

```
let d : float list = [3.2; 4.5]
let ex12 = map (fun x -> 2 * x + 1) d
```

Intuitively, a programmer typechecks this code as follows. Firstly, because `map` is applied to `d` of type `list float`, it must be the instance of `map` that operates over lists. Secondly, because the list contains elements of type `float`, the variable `x` should be of type `float`. It should follow that `2 * x + 1` is typechecked as involving operations over `float` values.

Yet, our algorithm, without additional feature, would fail to typecheck the above example. The function `map` is initially unresolved. The first pass of typing on the function `(fun x -> 2 * x + 1)`, performed without knowledge of `x`, provides no information whatsoever. Then, the function `map` is resolved based on its argument `d`. During the second pass of typing, we propagate the information that `(fun x -> 2 * x + 1)` has a type of the form `float -> ?t`. Thus, we learn that `x` is of type `float` through the resolution of the call to `map`. This second pass propagates downward, without any information at hand about the expected return type `?t` for the body `x + 1`. Thus, the second pass of the resolution is unable to resolve the type of the addition operator. One could handle this example with more than two passes, but we would like to avoid more than two passes to keep the time complexity low and the predictability high. In conclusion, the example `ex12`, without additional annotation, cannot be typed by the two-pass algorithm presented so far.

A simple yet unsatisfying work-around would consist in requiring a type annotation of the argument of the local function, that is, to write `(fun (x:float)-> 2 * x + 1)`. Yet, doing so would be frustrating because `d` is a `float list`, hence its elements are *obviously* of type `float`.

To capture this intuition, we introduce a general mechanism for overloaded functions, to distinguish arguments treated as *input* for typing from those treated as *output* for typing. Arguments in input-mode guide the resolution. Arguments in output-mode are not processed by the algorithm until the function call is resolved; at this point, the type expected for every argument is available.

Coming back to our motivating example, the first argument of `map` should be treated as an *output*, whereas the second one should be treated as an *input* by the typing algorithm. In our prototype implementation, the syntax for registering the input-ouput modes is by providing a list, as illustrated below.

```
let map = __overload [Out; In] (* input-output modes for arguments *)
```

Unless specified otherwise, all arguments are in input mode. A command such as the above is only required for overloaded functions that need arguments in output-mode. Typically, all higher-order iterators over containers would benefit from it. Note that the modes must be the same for all instances of a same symbol.

When the typing algorithm resolves a function call, it performs the first pass on the input-mode arguments only, and totally ignores the output-mode arguments. Then, it attempts to resolve the symbol based on the arguments. If the resolution succeeds, then the first-pass is performed on the output-mode arguments, and the result type of the function is returned. Else, if there are several matching instances, the output-mode arguments are ignored, and the type `Unresolved` is returned. During the second pass, the context may bring additional information by means of an expected return type. If the function was previously unresolved, the expected type must suffice to discriminate between the instances—that is, if the function is not resolved at this stage, the program is rejected. Otherwise, if the function is resolved with help of the return type, the first pass is performed on the output-mode arguments.[4] At that point, regardless of whether the function resolution took place in the first or the second pass, there remains to execute the second pass on the arguments, to complete the typing process.

The input-output mode mechanism may seem a little technical at first, but it appears necessary to mimic the intuitive process involved when typing mathematical expressions, without the need for additional type annotations, and without imposing a specific order to the arguments of a function.

[4]It is important not to skip the first-pass of the typing algorithm, even if the expected type is available, because there may be subterms that do not have an expected return type available, for which the first-pass is essential in order to infer all the types associated with these subterms.

This mechanism brings minor complications to the algorithm, yet provides a general solution to the case of higher-order iterators on containers.

9 DERIVED INSTANCES

The notion of derived instances can be used, for example, to express that as soon as an addition operator is associated with a type A, then an addition operator is available on matrices of elements of type A. Another example is that of reductions: for any container data structure equipped with a `fold` operator, and for any type equipped with a zero constant and an addition operator, one can derive a `sum` operator for instances of the container storing values of that type.

In what follows, we present our syntax for derived instances. We also describe the possibility for packing several instances. For example, for defining the `sum` operation, we can use a monoid structure to pack a zero constant and an addition operator into a single `addmonoid` instance. Then, we explain how we resolve instances: unlike for traditional typeclasses, our algorithm does not backtrack during resolution.

A simple derived instance. As first example, assume a type of matrices `'a matrix`, and assume an operation `matrix_add` that takes as argument an addition operator and two matrices. We can register an instance for matrix addition as follows.

```
val matrix_add : ('a -> 'a -> 'a) -> 'a matrix -> 'a matrix -> 'a matrix

(* Register an instance for [+] on the type ['a matrix], for every type
   ['a] for which there exists an instance of [+] on the type ['a]. *)
let (+) (type a) ((+) : a -> a -> a) : a matrix -> a matrix -> a matrix =
  __instance (fun m1 m2 -> matrix_add (+) m1 m2)
```

Instances with two arguments. As second example, let us show how to define a `sum` operator on arrays whose elements have a type that supports a zero constant and an addition operation.

```
(* Register an instance of [sum] for arrays with [+] and [zero]. *)
let sum (type a) ((+) : a -> a -> a) (zero : a) : a array -> a =
  __instance (fun s -> Array.fold (fun acc v -> acc + v) zero s)
```

Instances with packaged arguments. Let us next revisit the above example by introducing an additive monoid structure that carries both `zero` and `+`. First, we define a record to represent monoids.

```
  (* Structure to respresent monoids *)
  type 'a monoid = { op : 'a -> 'a -> 'a ; neutral : 'a }
```

Then we introduce an instance for the additive monoid on `int`. In the definition shown below, note that the symbols `(+)` and `0` are resolved, thanks to the type annotation `int monoid`, to be of type `int->int->int` and `int`, respectively.

```
  (* Register an instance of the additive monoid on [int] *)
  let addmonoid : int monoid = __instance { op = (+); neutral = 0 }
```

We can then revisit our definition on `sum` to depend on an additive monoid.

```
  (* Register an instance of [sum] for arrays whose elements are equipped
    with the additive monoid. *)
  let sum (type a) (m : a monoid) : a array -> a =
    __instance (fun s -> Array.fold (fun acc v -> m.op acc v) m.neutral s)
```

```
(* Example usage *)
let result1 = sum ([| 4; 5; 6 |] : int array)
```

Derived instances for monoids. In fact, we can state that for every type equipped with a zero and a sum operator, an additive monoid can be derived.

```
(* Register an instance of [addmonoid] for types with a [(+)] and [zero]. *)
let addmonoid (type a) ((+) : a -> a -> a) (zero : a) : a monoid =
  __instance ({ op = (+); neutral = zero })
```

With such an instance, we can remove our previous instance specific to `[int monoid]` and the example of `result1` would still successfully typecheck: the resolution `sum` on an `int array` triggers the resolution of `int monoid`, which in terms triggers the resolution of `(+)` and `zero` for the type `int`.

A more advanced example: fold and map-reduce. Let us generalize the definition of the `sum` function to all structures that exhibit a `fold` operator. The construction goes through the intermediate definition of a `mapreduce` operator.

```
(* Example instances of fold operators *)
let fold : ('a -> 'x -> 'a) -> 'a -> 'x array -> 'a = Array.fold_left
let fold : ('a -> 'x -> 'a) -> 'a -> 'x list -> 'a = List.fold_left

(** Register an instance of [mapreduce] derived from [fold] *)
let mapreduce (type t) (type a) (type x)
  (fold : (a -> x -> a) -> a -> t -> a)
  : (x -> a) -> a monoid -> t -> a =
  __instance (fun f m s -> fold (fun acc x -> m.op acc (f x)) m.neutral s)

(* Register an instance of [sum] derived from [fold] and [addmonoid] *)
let sum (type t) (type a)
  (addmonoid : a monoid)
  (mapreduce : (a -> a) -> a monoid -> t -> a)
  : t -> a =
  __instance (fun s -> mapreduce (fun x -> x) addmonoid s)

(* Example usage *)
let result2 = sum ([| 4; 5; 6 |] : int array)
```

An example mathematical formula. Recall our motivating example.

$$\sum_{d\in\{i,2i\}} \sum_{k\in[-6,7]} 3 \cdot e^{\frac{d\cdot\pi}{8}} \cdot M^{2\cdot k^2} \cdot N$$

Assuming instances of additions, products and exponent operators on integers, complex numbers and matrices, as well as instance for the integer range constructor, we can typecheck the formula, without the need for any type annotation inside the formula.

```
let demo (m:complex matrix) (n:complex matrix) =
  bigsum [i; 2*i] (fun d ->
    bigsum (range (-6) 7) (fun k ->
      3 * (e ^ (d * pi / 8)) * (m ^ (2*k^2)) * n))
```

Resolution policy for derived instance. In general, a derived instance takes the form: $\forall A_1..A_k.\ D_1 \Longrightarrow .. \Longrightarrow D_n \Longrightarrow T$, where A_i are type variables, where D_j represent the *premises*—that is, the instances to be resolved for the conclusion to hold—and where T denote the type of the instance that can be constructed.

Consider a type `Unresolved(ty,candidates)`, where the candidates are derived instances with conclusions T_1, .. T_n, and where `ty` is the type T_r that guides the resolution. The resolution process, which may be triggered during both passes of our typechecking algorithm, is as follows.

- If more than one type T_i unifies with T_r, no resolution takes place.
  (In particular, no backtracking is involved.)
- If exactly one type T_i unifies with T_r, resolution continues as follows.

(1) The types A_i are instantiated during the unification of T_i with T_r.
(2) Let D_j be the premises associated with T_i.
(3) The typechecker attempts to resolve the premises D_i, for these types A_i.
  - If all premises D_i can be resolved, the resolution is complete.
  - Else, the type remains `Unresolved(ty,candidates)`.

As an optimization, we can trim the list of instances to `Unresolved(ty,[candidate])`, where `candidate` was the unique remaining candidate. Furthermore, we can specialize the type of this single candidate to: $D_1 \Longrightarrow .. \Longrightarrow D_{n'} \Longrightarrow T_r$, where the D_j are instantiated with the aforementioned types A_i, and where only the D_j that were unresolved are kept. During the second typechecking pass, the types A_i may be further refined, allowing the remaining instances D_j to be resolved.

10 NON-TREATMENT OF PARTIAL APPLICATIONS

In this section, we explain what problems would arise if we wanted to support overloading and partial applications at the same time. In the languages C++, ADA, and PVS, which support static overloading resolution, the syntax of function calls takes the form `f(x,y)`, hence does not allow for partial applications. In contrast, in a traditional ML-style syntax, the syntax for a function call takes the form `f x y`, and the expression `f x` refers to the partial application of `f` to a first argument `x`. Now, what happens if we overload the name `f`?

For example, assume `sum x1 x2` and `sum x1 x2 x3` to be two overloaded functions—both functions are named `sum`, the first one expects 2 arguments, whereas the second one expects 3 arguments. If the programmer writes `let y = sum 3 4`, there are good chances that the intent is *not* a partial application. Yet, without further annotation, there is no way for the typechecker to know the programmer's intention and to rule out the possibility that `y` could be a partial application of the `sum` function that expects 3 arguments.

More generally, the experience from other languages, in particular C++, is that programmers routinely rely on the number of arguments to distinguish between several functions. Hence, if we were to allow for partial applications, we would significantly decrease the benefits of overloading, because we would impose on the programmer the writing of additional type annotations for disambiguation.

For this reason, we decided to not support the traditional ML-syntax for partial applications. Manual η-expansions, for example `fun z -> sum 3 4 z`, always remains possible, albeit syntactically heavy. To mitigate the syntactic overhead, we suggest introducing a new syntactic construct, with placeholders in the place of non-provided arguments. For example, `#(sum 3 4 _)` would be syntax for `fun z -> sum 3 4 z`. Likewise, `#(sum _ 4 5)` would be syntax for `fun x -> sum x 4 5` and `#(sum _ 4 _)` syntax for `fun x z -> sum x 4 z`. Note that in other scenarios, type annotations might be required to disambiguate between several overloaded functions, e.g., `#(f 2 (_:int))`.

In summary, our proposal is to make partial applications explicit. This way, `let y = sum 3 4` resolves to an application of the 2-argument `sum` function, without need for any annotation; and `let g = #(sum 3 4 _)` resolves to the partial application of the 3-argument `sum` function, at the cost of a very lightweight syntactic overhead—lighter than a type annotation.

11 OPAQUE VS TRANSPARENT TYPES IN RESOLUTION

When a type `t` is defined as an alias for another type `u`, it is not obvious whether the overloading resolution process should treat `t` and `u` as identical types, or as distinct types. This issue is well-known in the context of typeclasses, e.g., Coq provides *Typeclasses Transparent* and *Typeclasses Opaque* commands to control whether a given definition should be transparent or not with respect to typeclass resolution.

To handle the matter, we choose to follow ML-style practice. If `t` is defined as `u`, then `t` and `u` are unifiable and hence interchangeable throughout the scope of `t`. For example, two instances of respective type `u -> int` and `t -> int` will always overlap, hence the programmer should introduce only one of the two instances.

If, however, a type `t` is introduced as an *abstract type*, that is, as a type whose implementation is not revealed (e.g., hidden behind a module type), then `t` is not unifiable with any other type. In particular, overloading resolution may discriminate between instances by exploiting the fact that `t` and `u` are different types—even though the type `t` might have been once realized as `u`.

12 FUTURE WORK AND CONCLUSION

In future work, we would like to formalize the properties of our algorithm.

Besides, we would like to polish the error messages, following the ideas from previous work [Charguéraud 2015]. Last but not least, we would like to try using overloading at scale, in the context of ML programming as well as in the context of typechecking common mathematical formulae parsed using Coq's support for custom syntax.

REFERENCES

Arthur Charguéraud. 2015. Improving Type Error Messages in OCaml. *Electronic Proceedings in Theoretical Computer Science* 198 (dec 2015), 80–97. https://doi.org/10.4204/eptcs.198.4

Gabriel Dos Reis and Bjarne Stroustrup. 1985. *A Formalism for C++*. Technical Report. Technical Report.

Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional Typing. *ACM Comput. Surv.* 54, 5, Article 98 (may 2021), 38 pages. https://doi.org/10.1145/3450952

N. Shankar. 1996. PVS: Combining specification, proof checking, and model checking. In *Formal Methods in Computer-Aided Design*, Mandayam Srivas and Albert Camilleri (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 257–264.

Bjarne Stroustrup. 1984. *The C++ programming language: reference manual.* Technical Report. Bell Lab.

David A. Watt, Brian A. Wichmann, and William Findlay. 1987. *Ada language and methodology.* Prentice Hall International (UK) Ltd., GBR.