Typechecking of Overloading in Programming Languages and Mechanized Mathematics

Arthur Charguéraud[1], Martin Bodin[2], and Louis Riboulet[3]

[1]Inria & Université de Strasbourg, CNRS, ICube, France
[2]Inria, France
[3]ENS Lyon, France

Overloading consists of using a same symbol (or name) to refer to several functions (or 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 [Str84, DRS85]. PVS [Sha96] and ADA [WWF87] 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.

```
(* 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

[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.

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.

1.3 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]Our prototype can be tested online: `https://charguer aud.org/research/2025/overloading/proto.php`.

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.

```
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

[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.

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.

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 Typechecking the Core λ-calculus

3.1 Unification on Types

As usual in traditional ML typecheckers, types may be refined by side-effects during the traversals of the AST. A type could be a fresh unification variable (written `?A` in Coq syntax), or a partially known type (e.g. `list ?A`), or a fully resolved type (e.g. `list int`). Types may be unified, in which case they share the same representation. A Union-Find style data structure is used to keep track of such sharing.

In addition to ML types, we introduce a special type of the form `Unresolved(ty,candidates)`. This type is used during typechecking only, to represent the type associated with a term for which more than one candidate instance could match. The type `ty` is a type variable that represents the type of the term at hand, as viewed from the context. The candidates consist of a list of pairs (v_i,S_i), where each value v_i represents an instance whose (possibly polymorphic) type is S_i.

During the first typechecking pass on the AST, the type `ty` could be refined by unification, until it only unifies with one of the type scheme S_i. At this point, the unification of `ty` with S_i is effectively performed, and the type of the term becomes simply `ty`.

3.2 Typechecking of Constants

We next explain how to typecheck an overloaded constant c with expected type T. During the first pass, the result type T may remain `Unresolved`. At the end of the second pass, however, typechecking would fail if the type T remains `Unresolved`. For both passes, the key steps are as follows.

1. Consider the set of instances associated with c. Each instance is a value v_i of type S_i.
2. For each S_i, test whether T could unify with S_i. Keep the boolean result of the test, but undo side-effects that might have been performed during the unification process.
3. Count how many unifications have succeeded.
   - If none of the instance unify, raise a typechecking error.
   - If exactly one instance S_i can unify with T, then the constant c is resolved to be the value v_i. Unify S_i and T, and assign this type to be the type of c.
   - If several instances could unify, unify T with the type `Unresolved`, and assign this type to be the type of c.

Optimization: during the first pass, we can attach to the type `Unresolved` the list of the instances that could unify, saving work for the second pass. During the second pass, we can do the same, but this time for the purpose of reporting the list of candidates in a potential error message associated with unresolved instances.

3.3 Typechecking of Function Calls

In what follows, we explain how to typecheck a term of the form $t_0(t_1, .., t_n)$. As explained earlier, the typechecking process consists of two recursive traversals of the AST. Each of the two passes takes as argument a term and an expected type.

First pass. Assume the term $t_0(t_1, .., t_n)$ is typechecked with expected type T_r.

1. Introduce fresh type variables named T_1, ... T_n and T_r.
2. Recursively process the term t_0 with expected type $T_1 \to .. \to T_n \to T_r$. (If T_r is known, this might suffice to resolve the function—this is typically the case when t_0 is a data constructor.)
3. Recursively process the terms t_i with expected type T_i.
4. If t_0 has a `Unresolved` type, try to resolve it by typechecking this constant again with expected type $T_1 \to .. \to T_n \to T_r$. (This type may be partially known.)
5. Save the type T_r to be the current type of the term $t_0(t_1, .., t_n)$.
   (This step is common to every language construct during the second pass.)

Second pass on an application. Assume the term $t_0(t_1, .., t_n)$ is typechecked with expected type T_r. Let T be the type saved for this term at the end of the first pass. The second pass proceeds as follows.

1. Unify the type T with the type T_r.
   (This step is common to every language construct during the second pass.)
2. Retrieve the types T_i stored for each of the arguments t_i during the first pass.
3. Recursively process the function t_0 with expected type $T_1 \to .. \to T_n \to T_r$.
4. If t_0 is still not resolved, report failure to disambiguate t_0.
5. Recursively process each of the arguments with expected type T_i.

Note that the typechecking at step (3) may result in the types T_i being refined. Hence, step (5) may propagate downward additional type information obtained during the resolution of t_0.

3.4 Typechecking of Let-Bindings

Consider the typechecking of a term of the form $\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2$ against an expected type T. For simplicity, we describe here the case of monomorphic bindings, and ignore the details associated with ML-style generalization to polymorphism.

First pass.

1. Let T_1 be a fresh type variable.
2. Recursively typecheck t_1 with expected type T_1.
3. In an environment extended with a binding from x to T_1, recursively typecheck t_2 with expected type T. This means that, when traversing t_2, if we reach an occurrence of x with expected type T', then we unify this type T' with T_1.

Second pass. When executing the second pass of our typing algorithm on $\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2$, we process t_2 before t_1. The reason is that the second pass over t_2 may provide further type information about occurrences of x, and we wish to gather as much information as possible before propagating it into t_1.

1. In an environment extended with a binding from x to T_1, recursively typecheck t_2 with expected type T.
2. Recursively typecheck t_1 with expected type T_1.

At the end of this second pass, any instance that remains unresolved triggers a typechecking error.

4 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 *)
```

5 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.

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

[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.

7 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.

8 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.

9 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`.

10 Future Work and Conclusion

In future work, we plan to present formal rules for describing our algorithms, and to formalize the properties of our algorithm. These including several properties that are generally desirable for bidirectional typechecking algorithms [DK21]. Besides, we would like to polish the error messages, following the ideas from previous work [Cha15]. 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

[Cha15] Arthur Charguéraud. Improving type error messages in OCaml. *Electronic Proceedings in Theoretical Computer Science*, 198:80–97, dec 2015.

[DK21] Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Comput. Surv.*, 54(5), may 2021.

[DRS85] Gabriel Dos Reis and Bjarne Stroustrup. A formalism for c++. Technical report, Technical Report, 1985.

[Sha96] Natarajan Shankar. Pvs: Combining specification, proof checking, and model checking. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design*, pages 257–264, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[Str84] Bjarne Stroustrup. The c++ programming language: reference manual. Technical report, Bell Lab., 1984.

[WWF87] David A. Watt, Brian A. Wichmann, and William Findlay. *Ada language and methodology*. Prentice Hall International (UK) Ltd., GBR, 1987.