The Simple Essence of Overloading

Making Ad-Hoc Polymorphism More Algebraic with Flow-Based Variational Type-Checking

JIŘÍ BENEŠ, University of Tübingen, Germany
JONATHAN IMMANUEL BRACHTHÄUSER, University of Tübingen, Germany

Type-directed overload resolution allows programmers to reuse the same name, offloading disambiguation to the type checker. Since many programming languages implement overload resolution by performing backtracking in the type checker, it is commonly believed to be incompatible with Hindley-Milner-style type systems. In this paper, we present an approach to overload resolution that combines insights from variational type checking and algebraic subtyping. We formalize and discuss our flow-based variational framework that captures the essence of overloads by representing them as *choices*. This cleanly separates constraint collection, constraint solving, and overload resolution. We believe our framework not only gives rise to more modular and efficient implementations of type checkers, but also serves as a simpler mental model and paves the way for improved error messages.

CCS Concepts: • **Theory of computation** → **Type theory**; **Semantics and reasoning**; • **Software and its engineering** → **Polymorphism**.

Additional Key Words and Phrases: overloading, type inference, variational types

ACM Reference Format:
Jiří Beneš and Jonathan Immanuel Brachthäuser. 2025. The Simple Essence of Overloading: Making Ad-Hoc Polymorphism More Algebraic with Flow-Based Variational Type-Checking. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 390 (October 2025), 27 pages. https://doi.org/10.1145/3763168

1 Introduction

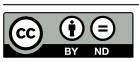
Naming things is *hard*.

> There are only two hard things in Computer Science: cache invalidation and naming things. *(Phil Karlton)*

This oft-quoted maxim reflects a fundamental challenge of programming language design. A natural solution is to reuse the same name for semantically similar operations—a practice known as *overloading*. Reusing the same name requires the language implementation to resolve the ambiguity. This is typically performed at compile time, using types to distinguish between different overloaded variants, choosing the variant that does not result in a type error.

Advantages of static overloading. Overloading offers substantial benefits to programmers. It enables consistent naming across different data types, reducing cognitive burden by eliminating the need for distinct identifiers. Static overloading is guaranteed to be a *zero-cost* abstraction: overloads are resolved at compile time, eliminating the runtime overhead of dynamic dispatch. This presents a significant advantage over dynamic alternatives.

Authors' Contact Information: Jiří Beneš, University of Tübingen, Germany, jiri.benes@uni-tuebingen.de; Jonathan Immanuel Brachthäuser, University of Tübingen, Germany, jonathan.brachthaeuser@uni-tuebingen.de.



ACM 2475-1421/2025/10-ART390
https://doi.org/10.1145/3763168

Disadvantages of static overloading. Type-directed overload resolution does not exactly have the best reputation. Over recent decades, many reasons against overloading have been brought forth, both by academics [Wadler and Blott 1989] and practitioners.

From the perspective of language design, a reused name carries multiple meanings, creating ambiguity that must be resolved not only by the programmer (trying to understand which function is called) but also by the compiler (to resolve the overload). A mismatch between the programmer's understanding of overload resolution and the actual implementation can result in surprising behavior at runtime. This is aggravated by often cryptic error messages trying to explain why resolution failed or is ambiguous. Finally, resolution instability means that seemingly innocuous program modifications can alter which overload is selected at a distance.

For language implementors, overloading introduces considerable complexity. Type inference pipelines typically backtrack through overloads, type checking each overloaded variant separately. To enable this, the type checker state must be backtrackable, which complicates the implementation's architecture. This backtracking behavior is inherently anti-modular, preventing clean phase separation and complicating reasoning about both type checker internals and outputs. Type checking of programs that use overloading extensively tends to be computationally expensive, negatively impacting compilation times.

> Swift regret (and this is a big one): type-based overloading
> When we were first developing Swift, many of the mainstream languages had type-based overloading (C++, Java, C#), and many didn't (C, Python, Objective-C). How did Swift end up with it? *(Jordan Rose, Swift Developer, 2021)*

Despite the drawbacks, overloading remains common even in newer languages like Swift, F#, and Koka. The theoretical and practical challenges, however, manifest in the folklore that type-directed overload resolution is fundamentally at odds with Hindley-Milner type inference [Milner 1978].

> First, if you're trying to do Hindley-Milner style unification-based type inference, that may not be compatible with overloaded functions. Most languages that support overloading don't have HM-style type inference and instead [do] local type inference to avoid this. *(Bob Nystrom, Dart Developer, 2024).*

After all, HM-style type inference allows us to cleanly separate constraint collection from constraint solving. How can this be reconciled with the backtracking needed to try different overloads?

1.1 Our Solution

In this paper, we present a *flow-based variational framework* that offers a novel perspective on overloading in HM-style type systems, suggesting that they are not incompatible after all. Our framework combines and extends two techniques: *variational type checking* and *algebraic subtyping*.

1.1.1 Variational Type Checking. Variational type checking [Chen et al. 2014] extends a programming language with *choices* [Erwig and Walkingshaw 2011], which we denote as $a\,\langle e_1,\ e_2\rangle$, where a is called a *dimension* with *alternatives* e_1 and e_2. While a non-deterministic execution of a variational program is possible, here we are interested in selecting one alternative for each dimension, allowing us to statically project a non-variational program. Variational type checking roughly means that a program always needs to type check, independently of the choices made for each dimension.

First key observation. Overload resolution can be cast into the framework of variational calculi. Specifically, we represent an overloaded function call as a choice between two functions:

Overloaded Source Program

```
1 + 2
```

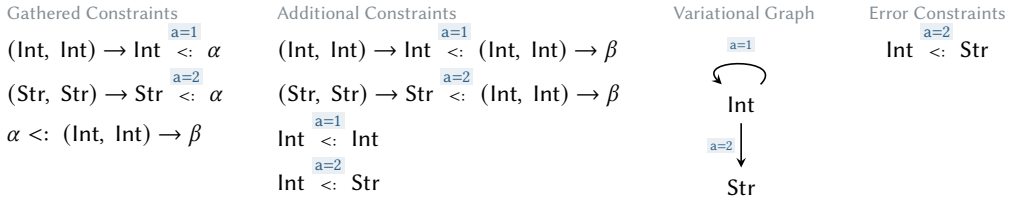
Variational Core

```
let f = a⟨addInt, addStr⟩
f(1, 2)
```

For each overloaded function call, name resolution introduces a fresh choice (and a new dimension, here a). As opposed to variational type checking, we require that there must be *exactly one* configuration in which the program type checks. If there are no configurations, then no overload can be chosen. If there are multiple configurations, the overload is ambiguous.

1.1.2 Algebraic Subtyping. Algebraic subtyping [Dolan and Mycroft 2017] is a type-inference procedure for ML-like languages with subtyping. Traditionally, HM-style type systems collect equality constraints like $\mathsf{Int} \equiv \alpha$, which are then solved by unification, resulting in a substitution mapping unification variables to a single type. Instead, algebraic subtyping collects inequality constraints of the form $\mathsf{Int} <: \alpha$, which can be read as *"type Int flows into α"*. Bi-unification solves the constraints by constructing the transitive closure of how types flow through a program, registering the lower and upper bounds for each bi-unification variable (*e.g.*, $\mathsf{Int} <: \alpha <: \top$). From the bounds, a bi-substitution is computed mapping each variable α to two types used to substitute into covariant and contravariant positions, respectively. Bhanuka et al. [2023] observed that HM-style type inference can be split into two phases: *flow analysis* and *bounds coercion*. The former phase proceeds as described above, while the latter phase coerces the bounds of each unification variable and requires them to be equal. This way, building on algebraic subtyping, our approach is compatible with subtyping which increases its applicability.

Second key observation. Constraints only need to hold for a specific configuration. We thus introduce *variational constraints* of the form $\mathsf{Int} \overset{a=1}{<:} \alpha$ saying that type Int flows into α in any world where we selected the first alternative for dimension a. Introducing α as type of f and β as type of the overall result, we collect the following constraints (first column) for the above example.



Solving the constraints results in additional constraints (second column), as usual. The first two arise from transitivity via α, the latter two from decomposition of the function type. The interesting constraints are visualized as a variational graph (third column), where the presence of edges depends on the configuration. Solving also results in *variational error constraints* (last column).

Third key observation. Error constraints are sufficient for a solver to perform overload resolution. In our example, the error constraint rules out all worlds where we select the second overload, and we thus conclude that $a = 1$ is the only valid configuration.

1.1.3 Properties. Implementing overloads in terms of our flow-based variational framework has a number of desirable properties:

- Constraint gathering is linear in the size of the program (and so is the number of constraints). In contrast to backtracking implementations of overloads, we never type check a term twice.
- Constraint solving is polynomial and can be memory efficient as it only operates on the constraints, not the program; it results in polynomial many errors.
- Overload resolution is modular: for this paper, we want to find a single valid configuration and require all others to fail. Different concrete instantiations, for example with a notion of "better overloads", are possible.

- Building on algebraic subtyping, we believe our framework is compatible with Bhanuka et al. [2023]'s approach for more detailed type error messages, which might be particularly relevant for notoriously difficult overload errors.

While type-directed name resolution is the original motivation, our framework may also be applicable to other forms of ambiguity, such as uniform function call syntax (see Section 6).

1.2 Contributions

To summarize, this paper makes the following contributions:

- a high-level, example-driven introduction to the concepts of our flow-based variational framework (Section 2). The framework gives rise to a new *semantic model* of overload resolution as the search for a single possible world without type errors. Types can simultaneously flow in multiple dimensions, but potentially only lead to errors in individual worlds.
- a formal presentation of a variational calculus $\lambda_{\propto:}$, a declarative variational type system (Section 3), and variational operational semantics.
- a formalization of overload resolution based on algebraic subtyping, cleanly separating overload resolution into constraint collection (Section 4.1), constraint solving (Section 4.2), overload resolution (Section 4.3), and program specialization (Section 4.4).
- an implementation of type-directed overload resolution as an instantiation of the framework, utilizing an off-the-shelf BDD solver (Section 4.3.1).
- a performance evaluation comparing our approach to Swift on practically motivated examples, as well as an analysis of the scaling behavior of our approach (Section 5).
- a discussion on various practical design decisions and other potential applications of our framework (Section 6).

Our framework and its semantic model enable a clear phase separation and avoid the typical problems of backtracking implementations. We believe this is not only important for implementors (who care about simplicity and performance of the implementation), but also for programmers who have to understand and reason about how overloads are resolved.

2 Flow-Based Variational Type Checking by Example

In this section, we introduce our framework and the relevant concepts by example. While the framework could potentially also be applied to other forms of ambiguity (such as overloaded syntax), in the remainder of the paper we primarily focus on type-directed overload resolution.

2.1 Flow-Based Overload Resolution

One of the most common examples of overloading is operator overloading, such as addition. As a first example, let us consider the following program (on the left) where we have a function that takes an argument x, and adds 0, a literal of type Int, to the result of adding x to x.

Overloaded Source Program

```
x ⇒ 0 + (x + x)
```

Variational Core

$\lambda(x) \Rightarrow$
let add_b = b⟨addInt, addStr⟩
let $inner : \beta$ = $add_b(x, x)$
let add_a = a⟨addInt, addStr⟩
let $result$ = $add_a(0 : \mathsf{Int}, inner)$
$result$

Like in the introduction, each mention of + is overloaded with two *alternatives*: `addInt` and `addStr`. Note that each + is overloaded independently. In our variational core, for ease of presentation, we rename the outer + to add_a, and the inner one to add_b and reify the overload into a choice,

for example the outer + to add_a = a⟨addInt, addStr⟩. Since each choice is independent, each choice has its own unique *dimension* (that is, a and b respectively).

2.1.1 The Flow of Types. Algebraic subtyping allows us to use our intuition about how values flow through the program to reason about type inference. Here, we apply the same intuition about program flow, but also use it to reason about which overload to choose. Like in the introduction, we read the subtyping constraint $\tau_1 <: \tau_2$ as τ_1 *flows into* τ_2. For simplicity, we only mention relevant flows, omit duplicate flows, and immediately decompose nested flows (such as, $\tau_1 \to \tau_2 <: \tau_3 \to \tau_4$ decomposing into $\tau_2 <: \tau_4$ and $\tau_3 <: \tau_1$ as usual). Also, while formally there are a few more bi-unification variables involved, we limit our presentation to only include β as the result of inner addition. The program gives rise to the following flows:

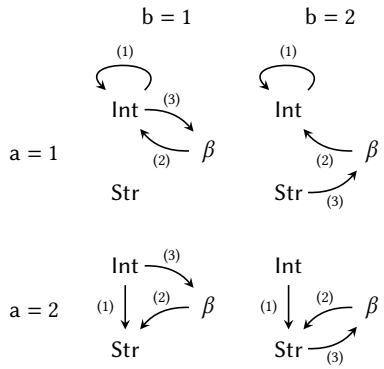
(1) Literal 0 of type Int, flows into add_a as its first argument. This gives us a flow Int <: Int if we select add_a = addInt and a flow Int <: Str if we select add_a = addStr.
(2) Additionally, selecting add_a = addInt induces a flow β <: Int as inner of type β flows to add_a as the second argument. Similarly, for the very same reason, selecting add_a = addStr induces a flow β <: Str in the program.
(3) Finally, if we select add_b = addInt, we get a flow Int <: β, since Int as the result of add_b flows into inner and therefore into β. Analogously, if we select add_b = addStr, we get a flow Str <: β.

In the intuitive description above, we can notice that some flows only exist under certain configurations. This is visualized in the following table on the left.

Constraints for Each World

	b = 1 (addInt)	b = 2 (addStr)
a = 1 (addInt)	(1) Int <: Int (2) β <: Int (3) Int <: β	(1) Int <: Int (2) β <: Int (3) Str <: β
a = 2 (addStr)	(1) Int <: Str (2) β <: Str (3) Int <: β	(1) Int <: Str (2) β <: Str (3) Str <: β

Flow Graph for Each World



Each row corresponds to a different overload for add_a, that is, selections for dimension a: a = 1 stands for selecting function addInt; a = 2 stands for selecting addStr. Similarly, each column corresponds to a different overload for add_b. Each cell of the table is a *world*, combining the selections for a and b. The table contains four different worlds, each being a different way of resolving the choices/overloads induced by add_a and add_b, each containing different constraints.

2.1.2 Eliminating Invalid Worlds. The goal of overload resolution is to show that all but one world lead to type errors. Naïve backtracking implementations typically proceed as follows: for each of the possible worlds, they perform type checking to find whether a single world type checks or not. Similarly, but without type checking the program over-and-over again, in our model, we seek to eliminate all *invalid* worlds, that is, worlds leading to an invalid, contradictory flow (such as Int <: Str). To this end, we again revisit the table above (to the right), but this time each world does not consist of the collected constraint, but of the graph resulting from constraint solving.

First off, we can notice that in the worlds in row a = 2, *i.e.*, the worlds where add_a = `addStr`, we have that constraint (1) is a flow from Int to Str. These two worlds are thus invalid, as Int and Str are incomparable in our system. Therefore add_a may **never** resolve to `addStr`, or in other words: we reject **every** world where a = 2 as invalid, here the bottom row. For this elimination process, we keep our reasoning open-world, assuming new alternatives could show up later. Specifically, we do not yet say that a = 1 (add_a = `addInt`) just because there are two options and a ≠ 2.

Ruling out the bottom row leaves us with two worlds. Next, let us consider the top-right world, where a = 1 *and* b = 2. In this world add_a is resolved to `addInt` and, at the same time, add_b is resolved to `addStr`. Inspecting the flow graph, we can find a *transitive flow* from Str to Int via β: Int <: β (3) followed by β <: Str (2). This means that the top-right world is, again, invalid.

Finally, we see that only the top-left world remains, where we resolve add_a to `addInt` and add_b to `addInt`. Since this world is valid—it contains no direct or transitive erroneous flows—and we have eliminated all the other worlds, we use this knowledge to finally resolve the overloads. Mentions of add_a can be substituted by `addInt` and, at the same time, add_b can be substituted by `addInt`, as that is the only consistent resolution of the overloading problem at hand.

2.1.3 Variational Constraints. In the above walkthrough, illustrating the *model* of our system by showing all possible worlds, it might appear that our approach is not much different from traditional ones. We individually visited each possible world and reasoned about it individually to conclude only the top-left world, where both additions are on integers, is valid.

However, not only do we perform type checking only once, to gather constraints, but those constraints are *variational*, admitting a compact representation. In particular, here is how all constraints of all possible worlds look like in our system:

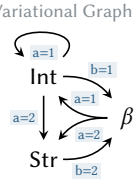
Constraints

(1) Int $\overset{a=1}{<:}$ Int, Int $\overset{a=2}{<:}$ Str

(2) β $\overset{a=1}{<:}$ Int, β $\overset{a=2}{<:}$ Str

(3) Int $\overset{b=1}{<:}$ β, Str $\overset{b=2}{<:}$ β

Variational Graph



Error Constraints

Int $\overset{a=2}{<:}$ Str by (1)

Int $\overset{b=1 \& a=2}{<:}$ Str by (3) & (2)

Str $\overset{b=2 \& a=1}{<:}$ Int by (3) & (2)

Constraint solving produces a compact representation of the bounds (middle column, here presented as a graph with variational edges) and error constraints (right column). Inspecting the variational graph, we can see that it concisely expresses the four worlds above. In our formalization, however, it is merely a means to generate error constraints obtained by constructing the transitive closure. A solver only processes the error constraints, immediately rejecting the worlds corresponding to the configurations annotated on the error constraints. In the second and third error constraint, we can notice that the error constraint is annotated with a conjunctive configuration (*e.g.*, b = 1 & a = 2).

How we arrive at this configuration may seem trivial in retrospect, but rests on the following key insight: following edges in the flow graph one after another means applying transitivity; in order for a flow to be meaningful, we collect the conjunction of configurations along the way! For example, a flow from Int via β to Str goes through edges labeled with configurations b = 1 and a = 2. Note that configurations can be *absurd*, that is, going through edges labeled a = 1 and a = 2 results in a configuration a = 1 & a = 2. However, there exists no world for which these contradicting requirements hold, and the flow is thus irrelevant.

2.2 Precise Ambiguity

The previous subsection showed an example where overload resolution found a unique valid world. The difficulty in designing and implementing overload resolution, however, surfaces when handling

cases where no valid configuration exists (the program never type checks) or where multiple configurations are satisfiable (that is, *ambiguous overloads*). In these situations implementations typically confront the user with all possible choices, producing overwhelming error messages. We now turn to an example that illustrates how our system handles ambiguity with more precision, using flow-based reasoning to eliminate inconsistent overload combinations.

Return-type overloading presents a fundamental challenge in type systems. Consider the `Show` and `Read` type classes, which enable conversion between values and their string representations and are ubiquitous in functional languages. The composition `show . read` notoriously results in ambiguity errors in systems with type classes, as the type variable connecting these operations cannot be determined uniquely. While traditional type systems struggle with this ambiguity, our variational approach offers a principled solution. Let us consider this example (on the left) translated to core with annotated type variables and choices (on the right):

Overloaded Source Program

```
str ⇒
  let value = read(str)
  let result = show(value)
  result
```

Variational Core

$\lambda(str : \alpha) \Rightarrow$
let *read* = a⟨readInt, readDouble⟩
let *value* : β = *read*(*str*)
let *show* = b⟨showInt, showDouble, showStr⟩
let *result* : γ = *show*(*value*)
result

Type checking the above example, our system generates the following simplified constraints:

(1) Flow of choice a into *value*: Int $\overset{a=1}{<:}$ β, Double $\overset{a=2}{<:}$ β

(2) Flow of *value* into choice b: β $\overset{b=1}{<:}$ Int, β $\overset{b=2}{<:}$ Double, β $\overset{b=3}{<:}$ Str

(3) Flow of *str* into argument of *read*: α $\overset{a=1}{<:}$ Str, α $\overset{a=2}{<:}$ Str

(4) Flow of choice b into *result*: Str $\overset{b=1}{<:}$ γ, Str $\overset{b=2}{<:}$ γ, Str $\overset{b=3}{<:}$ γ

For the purpose of this discussion, we focus on constraints (1) and (2), which reveal the interactions between the dimensions a and b. The following table shows the constraints of all six possible worlds (2 alternatives for a × 3 alternatives for b) at a glance:

	b = 1 (showInt)	b = 2 (showDouble)	b = 3 (showStr)
a = 1 (readInt)	(1) Int <: β (2) β <: Int	(1) Int <: β (2) β <: Double	(1) Int <: β (2) β <: Str
a = 2 (readDouble)	(1) Double <: β (2) β <: Int	(1) Double <: β (2) β <: Double	(1) Double <: β (2) β <: Str

Like before, each cell in the table represents a complete configuration—a world in our framework. For example, the top-left cell denotes the world where both a = 1 & b = 1 hold simultaneously. In the remainder, we use a shorthand multiplicative notation a_1b_1 to identify these configurations.

2.2.1 Eliminating Invalid Worlds. The key insight of our approach is the ability to identify invalid worlds through flow-based reasoning. Again, we do so by inspecting the variational graph (Figure 1, top left) that results form solving the variational constraints. It compactly represents the six possible worlds (Figure 1, right), which we can obtain by instantiating our variational graph structure. Grayed-out subgraphs are independent of the configuration and thus common amongst all worlds. Solving the constraints amounts to computing the transitive closure, which leads to the error constraints on the bottom left of Figure 1. Inspecting the error constraints, we can immediately eliminate four configurations with inconsistent constraints: a_1b_2 (*i.e.*, `readInt`, `showDouble`), a_1b_3 (*i.e.*, `readInt`, `showStr`), a_2b_1 (*i.e.*, `readDouble`, `showInt`), and finally a_2b_3 (*i.e.*,

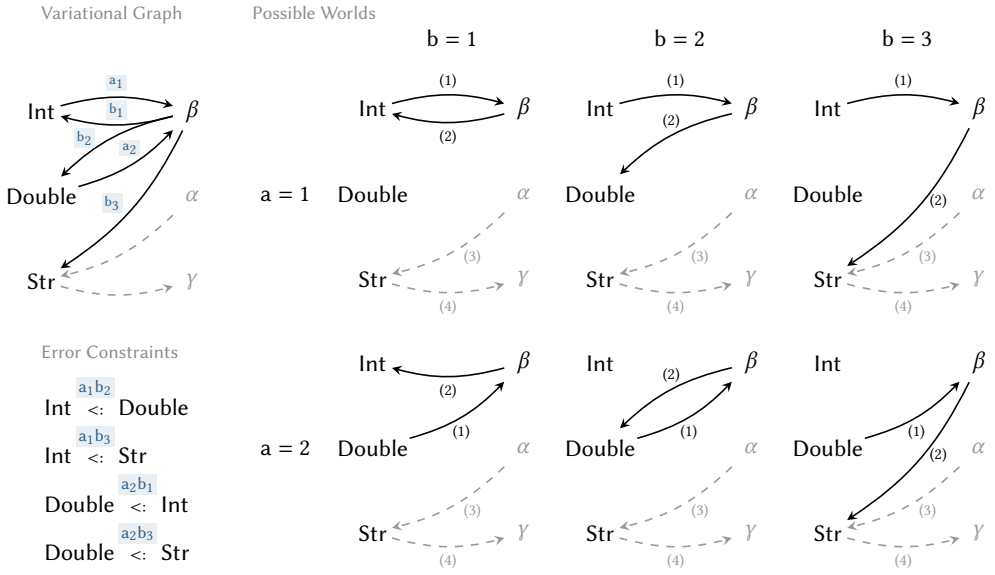


Fig. 1. Variational graph of the program show . read and its semantic expansion into six possible worlds.

readDouble, showStr). After eliminating the four worlds corresponding to these configurations, only two valid worlds remain. World a_1b_1 (where *read* = readInt and *show* = showInt) and world a_2b_2 (where *read* = readDouble and *show* = showDouble).

This demonstrates that our system can identify ambiguity with precision. While multiple valid worlds remain (two in this case), we successfully eliminate invalid combinations, reducing the search space from 6 to 2 possible worlds. Notably, our system never needs to consider configurations involving showStr in its error reporting, as they have been conclusively eliminated.

To resolve the ambiguity to a single solution, the programmer has several options:

- to resolve to a_1b_1 (*i.e.*, via $\beta \equiv$ Int):
  - *either* resolve the *read*, dimension a, to a_1 (*read* = readInt),
  - *or* resolve the *show*, dimension b, to b_1 (*show* = showInt)
- to resolve to a_2b_2 (*i.e.*, via $\beta \equiv$ Double):
  - *either* resolve the *read*, dimension a, to a_2 (*read* = readDouble),
  - *or* resolve the *show*, dimension b, to b_2 (*show* = showDouble)

Our system conceptually provides this information in its error messages, giving the programmer guidance on how to resolve the ambiguity.

2.2.2 Comparison with Existing Approaches. Our approach contrasts sharply with how other similar systems handle such ambiguities.

Opaque type inference failures. Languages like Haskell and Rust fail with messages such as "ambiguous type variable" or "type annotations needed", identifying the ambiguity but providing limited guidance on resolution.

Exhaustive enumeration. F# lists all possible overloads without narrowing down to valid combinations, overwhelming the programmer with options that may not be consistent.

First-error halting. Languages like Swift, Scala, Koka, or Effekt stop at the first ambiguous overload without analyzing subsequent constraints, missing opportunities to narrow down valid options.

Restricted overloading. Some languages like Java or C++ simply do not support return-type overloading, which avoids the issue entirely but loses expressiveness.

2.2.3 Summary. In this subsection, we demonstrated how variational typing with flow-based reasoning handles ambiguity with greater precision. Providing developer-friendly error messages is traditionally challenging, since type checking and overload resolution are interleaved. By separating the two phases, our approach gives the solver the full picture, enabling more precise errors.

2.3 Detailed Failure

In the previous subsection, we illustrated how our system handles ambiguity, *i.e.* when multiple valid worlds remain. Here we briefly discuss scenarios where all worlds are invalid. Consider the following source program and its desugaring into our core calculus on the left.

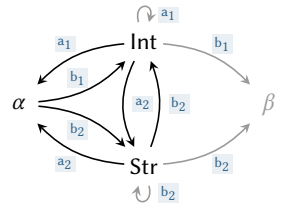
Overloaded Source Program

```
x ⇒ (x + 2) + "3"
```

Variational Core

```
λ(x) ⇒
  let add_a = a⟨addInt, addStr⟩
  let fst : α = add_a(x, 2 : Int)
  let add_b = b⟨addInt, addStr⟩
  let snd : β = add_b(fst, "3" : Str)
  snd
```

Variational Graph



Invalid Configurations

a_2	by	Int	$\overset{a_2}{<:}$	Str		
b_1	by	Str	$\overset{b_1}{<:}$	Int		
a_1b_2	by	Int	$\overset{a_1}{<:}$	α	$\overset{b_2}{<:}$	Str
a_2b_1	by	Str	$\overset{a_2}{<:}$	α	$\overset{b_1}{<:}$	Int

In this example no overload can be chosen, since the integer 2 conflicts with the string "3" through the additions. Some language implementations commit early—for example, resolving the first addition (dimension a) to `addInt`—and then report an error on the string argument, stating that it is not an integer. In our implementation, type checking yields the variational graph in the middle column. Edges irrelevant to type errors in any world are greyed out. Inspecting the graph reveals four conflicting flows, summarized on the right: the first two invalid configurations are immediate, while the latter two arise via transitivity and an interaction between two choices. Our system does not privilege one conflicting overload over another; it is up to the user to manually insert a conversion from Int to Str or vice versa if they so desire.

Here, we do not develop a theory of error messages, but note that the last configuration (*i.e.* a_2b_1) is also covered by the first (*i.e.* a_2) and it thus might be preferable to only report the first three. Building up on our system and the work of Bhanuka et al. [2023], we conjecture it should be possible to add detailed explanations as to why a particular overload **cannot** be taken. For example, the conflicting configuration a_1b_2 could be presented as:

> *"The result of overload add_a =* `addInt` *of type Int flows into the first argument of overload add_b =* `addStr`*, which expects a Str".*

2.4 Complex Choices

So far, the alternatives e_1 and e_2 of a choice $a\langle e_1, e_2\rangle$ were always mere names. As illustrated above, restricting ourselves to choose between names is sufficient to express type-directed overload resolution. Our system, however, does not impose such a restriction and allows alternatives to be arbitrary expressions. For example, we can model an overload for a default *value* of a type without resorting to naming and floating out the individual expressions that compose its alternatives.

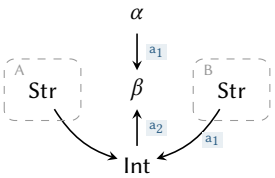
$$\textbf{let } default \;=\; a\langle 0,\; 0.0,\; \text{“”}\rangle$$

2.4.1 Failing Locally vs. Failing Universally. Allowing arbitrary expressions as alternatives opens up an interesting interaction with type checking and overload resolution. One such interaction is illustrated in the following pair of examples. Example A (on the left) includes an explicit, non-generalized binding named *invalid*, which causes a type error as Str cannot be compared with Int. Consequently, the whole compilation unit fails during type checking, regardless of the configuration. This failure shows up as an erroneous constraint Str <: Int that holds in every world.

Example A

let *invalid* : α = addInt(“”, 0)
a ⟨*invalid*, 42⟩ : β

Variational Graphs for A and B



Example B

a ⟨addInt(“”, 0) : α, 42⟩ : β

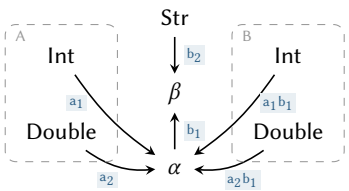
In contrast, Example B (on the right) inlines *invalid* at its sole use, so the invalid application of addInt to “” and the corresponding constraint Str <: Int occur only in configuration a_1, as the application only happens in worlds where a = 1. Consequently, type checking and overload resolution both succeed: we reject the invalid world a_1 which contains the malformed application, but since world a_2 does not pose a problem, we resolve the overload to the second alternative: 42.

2.4.2 Nesting Choices. If it is possible to use arbitrary expressions as alternatives, it is also possible to nest choices. Take the following pair of examples A and B where we attempt to decompose our default value overload a ⟨0, 0.0, “”⟩ into two separate choices.

Variational Core A

let *defaultNum* : α = a ⟨0, 0.0⟩
b ⟨*defaultNum*, “”⟩ : β

Variational Graphs for A and B



Variational Core B

b ⟨ a ⟨0, 0.0⟩ : α, “”⟩ : β

In Example A, we extract and bind the inner choice; in Example B, we inline it. The difference manifests itself in the constraints. In Example A, constraints (and hence the edges representing the flow) Int <: α and Double <: α live in worlds where we choose a_1 and a_2, respectively. In Example B, the whole choice a ⟨0, 0.0⟩ lives under configuration b = 1, so the constraints also include b_1. As seen in Subsection 2.1.3, transitivity of subtyping leads to a conjunction of the corresponding configurations; here, the conjunction is introduced by nesting.

Our system also allows nesting the *same* choice, which leads to an interesting quirk.

let *default* = a ⟨ a ⟨0, 0.0⟩, “”⟩

Applying flow-based reasoning: selecting a_1 resolves *default* to 0 : Int, while selecting a_2 resolves *default* to “” : Str. The case 0.0 : Double is thus unreachable, as also observed by Chen et al. [2014].

2.4.3 Summary. In this section, we explored flow-based variational overload resolution by example. We saw how constraints can be solved into a variational graph and error constraints. Using tabular enumeration of all possible worlds as semantic model, we observed how the compact variational graph representation enables precise reporting of ambiguous overloads and facilitates detailed error messages. Following multiple edges corresponds to transitivity of subtyping, which in turn leads to the conjunction of the associated configurations. Finally, we discussed several advanced use cases beyond the domain of overload resolution.

Variational Core Terms:

Expressions	e	::=	x
		|	true | false | 42 | 3.14 | ...
		|	$\lambda(\overline{x_i}) \Rightarrow e$
		|	$e(\overline{e_i})$
		|	**if** e_1 **then** e_2 **else** e_3
		|	$a\,\langle \overline{e_i} \rangle$
Dimensions	a	::=	a | b | ...
Abbreviations	**let** $x = e_1;\ e_2$	$\doteq$	$(\lambda(x) \Rightarrow e_2)(e_1)$

Variational Core Types:

Types	τ	::=	Int | Bool | Str | ...
		|	$\top$
		|	$\bot$
		|	$(\overline{\tau_i}) \rightarrow \tau$
Environment	Γ	::=	$x : \tau, \Gamma$
		|	$\bullet$

Fig. 2. Syntax of terms and types of the $\lambda_{\diamond<:}$ calculus.

3 Variational Core

In this section, we formally describe the variational core calculus $\lambda_{\diamond<:}$ as the essence of our framework. We formalize its syntax, operational semantics, and declarative type system, leaving type inference to the subsequent Section 4.

3.1 Syntax

Figure 2 defines the syntax of terms and types of the $\lambda_{\diamond<:}$ calculus.

Syntax of terms. Expressions e are mostly standard. Variables x are assumed to be globally unique [Barendregt 1984] and names in $\lambda_{\diamond<:}$ are thus, in a sense, not overloaded. Each variable has a single definition and overloads need to be manually desugared into explicit choices. In contrast, two choices can refer to the same dimension, as illustrated in the previous section. Expressions further include primitive literals, multi-argument lambda abstractions $\lambda(\overline{x_i}) \Rightarrow e$, applications $e(\overline{e_i})$, and conditionals **if** e_1 **then** e_2 **else** e_3. We express non-generalized let bindings **let** $x = e$; e by desugaring them, as usual. Most interestingly, expressions include choices $a\,\langle \overline{e_i} \rangle$, which consist of a dimension a together with alternatives e_i that the choice in question presents.

Syntax of types. Again, types are mostly standard for a language with subtyping and consist of primitive types like Int, top and bottom types $\top$ and $\bot$, and multi-argument function types $(\overline{\tau_i}) \rightarrow \tau$. Note that all the types in the core calculus are monotypes. In the declarative system, environments Γ are unsurprising—we will extend them in Section 4 when talking about type inference.

3.2 Declarative Typing

Figure 3 defines the declarative typing rules of the $\lambda_{\diamond<:}$ calculus. The typing judgement $\Gamma \vdash_\Delta e : \tau$ states that an expression e has type τ in the variable context Γ and a *resolution* Δ. A resolution Δ maps each dimension $a \in Dim$ to the index of the selected alternative $1 \leqslant i \leqslant ar(a)$, where $ar(a)$ is the arity of dimension a. We require Δ to be well-formed, which means it chooses exactly one alternative for each dimension, uniquely determining the variational program's semantics. For declarative typing, we assume the resolution Δ to be arbitrarily provided by an oracle.

The declarative typing rules are mostly standard. The only interesting rule is rule Choice, which expresses a selection driven by Δ. It requires that the alternative in the resolution Δ, and only this one, is well-typed. That is, if the resolution context Δ selects the j-th alternative to the dimension a and if e_j has type τ, then the whole choice also has type τ. Note that using a different resolution Δ can result in a different type, which is the entire point of overloads. For example, $\vdash_\Delta \mathrm{a}\langle 0, \text{“”} \rangle : \tau$ types as $\tau = \mathsf{Int}$ for $\Delta(\mathrm{a}) = 1$ and as $\tau = \mathsf{Str}$ for $\Delta(\mathrm{a}) = 2$.

Declarative typing $\boxed{\Gamma \vdash_\Delta e : \tau}$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_\Delta x : \tau}\ [\textsc{Var}] \qquad \frac{}{\Gamma \vdash_\Delta 42 : \mathsf{Int}}\ [\textsc{Prim}]$$

$$\frac{\Gamma \vdash_\Delta e : (\overline{\tau_i}) \to \tau \quad \overline{\Gamma \vdash_\Delta e_i : \tau_i}}{\Gamma \vdash_\Delta e(\overline{e_i}) : \tau}\ [\textsc{App}] \qquad \frac{\Gamma, \overline{x_i : \tau_i} \vdash_\Delta e : \tau}{\Gamma \vdash_\Delta \lambda(\overline{x_i}) \Rightarrow e : (\overline{\tau_i}) \to \tau}\ [\textsc{Abs}]$$

$$\frac{\Gamma \vdash_\Delta e_1 : \mathsf{Bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash_\Delta \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau}\ [\textsc{If}]$$

$$\frac{\Gamma \vdash_\Delta e_j : \tau \quad \Delta(a) = j}{\Gamma \vdash_\Delta a\langle \overline{e_i} \rangle : \tau}\ [\textsc{Choice}] \qquad \frac{\Gamma \vdash_\Delta e : \tau_1 \quad \tau_1 <: \tau_2}{\Gamma \vdash_\Delta e : \tau_2}\ [\textsc{Sub}]$$

Declarative subtyping $\boxed{\tau_1 <: \tau_2}$

$$\frac{\tau <: \tau' \quad \tau' <: \tau''}{\tau <: \tau''}\ [\textsc{S-Trans}] \qquad \frac{\overline{\tau'_i <: \tau_i} \quad \tau <: \tau'}{(\overline{\tau_i}) \to \tau <: (\overline{\tau'_i}) \to \tau'}\ [\textsc{S-Fun}]$$

$$\frac{}{\tau <: \tau}\ [\textsc{S-Refl}] \qquad \frac{}{\tau <: \top}\ [\textsc{S-Top}] \qquad \frac{}{\bot <: \tau}\ [\textsc{S-Bot}]$$

Fig. 3. Declarative typing and subtyping for the $\lambda_{\diamond<:}$ calculus.

The $\lambda_{\diamond<:}$ calculus supports a simple form of subtyping by means of the subsumption rule Sub, making declarative typing non-syntax-directed. Figure 3 defines subtyping, which, as usual, is a reflexive (S-Refl), transitive (S-Trans) binary relation with a greatest element $\top$ (S-Top) and a smallest element $\bot$ (S-Bot). Finally, the rule S-Fun states that function types are contravariant in their argument types and covariant in their result type.

3.3 Operational Semantics

We formalize the semantics of $\lambda_{\diamond<:}$ using a CK abstract machine, where machine states $M = \langle\, e \mid \mathsf{K} \,\rangle$ are expression-context pairs. Figure 4 defines the operational semantics, which is standard except for supporting of multiple-argument functions and choices. Contexts K model the machine stack as lists of frames F, which in turn are expressions with a single hole.

The stepping relation (defined in Figure 4) requires a resolution Δ to take a machine state M to M'. Most rules ignore the resolution Δ and are completely unsurprising. The only interesting rule is *(choice)*, which chooses the alternative according to the resolution Δ. Similar to the typing rules, this means that the result of a step depends on the resolution Δ.

When we *specialize* the choices (resolve the overload) using the next section's machinery, no choices remain, the *(choice)* rule never applies, and the operational semantics becomes independent of the resolution context. Figure 5 contains standard, CK machine typing, extensional frame typing, and the operational semantics. The extensional definition of frame typing follows the extensional definition of context typing by Jacobs [2016].

3.4 Metatheoretical Properties

The $\lambda_{\diamond<:}$ calculus satisfies the usual soundness properties of progress and preservation.

Context and Frames

$$\begin{array}{llcll} \text{Context} & \mathsf{K} & ::= & F ::: \mathsf{K} & \text{frame} \\ & & \mid & \bullet & \text{empty} \end{array} \qquad \begin{array}{llcll} \text{Frame} & F & ::= & (\lambda(\overline{x_i}) \Rightarrow e)(\overline{v_j},\ \Box,\ \overline{e_k}) & \text{app-argument} \\ & & \mid & \Box(\overline{e_k}) & \text{app-function} \\ & & \mid & \textbf{if } \Box \textbf{ then } e_1 \textbf{ else } e_2 & \text{if-condition} \end{array}$$

Reduction Rules $\boxed{\Delta \vdash \langle\, e \mid \mathsf{K} \,\rangle \longrightarrow \langle\, e' \mid \mathsf{K}' \,\rangle}$

$$\textit{(choice)}\quad \Delta \vdash \langle\, a\,\langle \overline{e_i}\rangle \mid \mathsf{K} \,\rangle \longrightarrow \langle\, e_j \mid \mathsf{K} \,\rangle \qquad \text{where } \Delta(a) = j$$

$$\begin{array}{lll} \textit{(pushfn)} & \Delta \vdash & \langle\, (\lambda(\overline{x_i}) \Rightarrow e')(e,\ \overline{e_k}) \mid \mathsf{K} \,\rangle \longrightarrow \langle\, e \mid (\lambda(\overline{x_i}) \Rightarrow e')(\Box,\ \overline{e_k}) ::: \mathsf{K} \,\rangle \\ \textit{(switch)} & \Delta \vdash & \langle\, v \mid (\lambda(\overline{x_i}) \Rightarrow e')(\overline{v_j},\ \Box,\ e,\ \overline{e_k}) ::: \mathsf{K} \,\rangle \longrightarrow \langle\, e \mid (\lambda(\overline{x_i}) \Rightarrow e')(\overline{v_j},\ v,\ \Box,\ \overline{e_k}) ::: \mathsf{K} \,\rangle \\ \textit{(popfn)} & \Delta \vdash & \langle\, v \mid (\lambda(\overline{x_i},\ x) \Rightarrow e)(\overline{v_i},\ \Box) ::: \mathsf{K} \,\rangle \longrightarrow \langle\, e\,[\overline{x_i \mapsto v_i},\ x \mapsto v] \mid \mathsf{K} \,\rangle \\ \textit{(pusharg)} & \Delta \vdash & \langle\, e(\overline{e_k}) \mid \mathsf{K} \,\rangle \longrightarrow \langle\, e \mid \Box(\overline{e_k}) ::: \mathsf{K} \,\rangle \\ \textit{(poparg)} & \Delta \vdash & \langle\, (\lambda(\overline{x_i}) \Rightarrow e) \mid \Box(\overline{e_k}) ::: \mathsf{K} \,\rangle \longrightarrow \langle\, (\lambda(\overline{x_i}) \Rightarrow e)(\overline{e_k}) \mid \mathsf{K} \,\rangle \\ \textit{(pushif)} & \Delta \vdash & \langle\, \textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2 \mid \mathsf{K} \,\rangle \longrightarrow \langle\, e \mid \textbf{if } \Box \textbf{ then } e_1 \textbf{ else } e_2 ::: \mathsf{K} \rangle \\ \textit{(iftrue)} & \Delta \vdash & \langle\, \text{true} \mid \textbf{if } \Box \textbf{ then } e_1 \textbf{ else } e_2 ::: \mathsf{K} \,\rangle \longrightarrow \langle\, e_1 \mid \mathsf{K} \,\rangle \\ \textit{(iffalse)} & \Delta \vdash & \langle\, \text{false} \mid \textbf{if } \Box \textbf{ then } e_1 \textbf{ else } e_2 ::: \mathsf{K} \,\rangle \longrightarrow \langle\, e_2 \mid \mathsf{K} \,\rangle \end{array}$$

Fig. 4. Operational semantics of the $\lambda_{\diamond<:}$ calculus as variational abstract machine. All rules, except rule *(choice)* are standard.

Machine Typing $\boxed{\Gamma \vdash_\Delta \langle\, e \mid \mathsf{K} \,\rangle : \tau}$

$$\frac{\Gamma \vdash_\Delta e : \tau \qquad \vdash_\Delta \mathsf{K} : \tau \Rightarrow_M \tau'}{\Gamma \vdash_\Delta \langle\, e \mid \mathsf{K} \,\rangle : \tau'}\ [\textsc{MachineType}] \qquad \frac{}{\Gamma \vdash_\Delta \bullet : \tau}\ [\textsc{MachineStackNil}]$$

$$\frac{\Gamma \vdash_\Delta F : \tau \Rightarrow_F \tau' \qquad \Gamma \vdash_\Delta \mathsf{K} : \tau' \Rightarrow_M \tau''}{\Gamma \vdash_\Delta F ::: \mathsf{K} : \tau \Rightarrow_M \tau''}\ [\textsc{MachineStackFrame}]$$

Frame Typing $\boxed{\Gamma \vdash_\Delta \mathsf{K} : \tau \Rightarrow_F \tau'}$

$$\Gamma \vdash_\Delta F : \tau \Rightarrow_F \tau' \quad := \quad (\forall e.\, \Gamma \vdash_\Delta e : \tau) \Rightarrow \vdash_\Delta F[e] : \tau'$$

Fig. 5. Typing rules for machine states and frames.

Theorem 3.1 (Progress).
If $\Gamma \vdash_\Delta M$ *then either* $M = \langle\, v \mid \bullet \,\rangle$, *or there exists* M' *such that* $\Gamma \vdash_\Delta M \longrightarrow M'$.

Proof. By case distinction on e, where $M = \langle\, e \mid \mathsf{K} \,\rangle$. The only interesting case is $a\,\langle \overline{e_i}\rangle$. Applying the inversion lemma, to remove applications of subsumption, reveals an application of rule Choice with premise $\Delta(a) = j$. If Δ is not defined at a, we would be stuck, but this cannot be the case since a typing derivation exists, and we proceed by applying rule *(choice)* with $\langle\, e_j \mid \mathsf{K} \,\rangle$. □

Theorem 3.2 (Preservation).
If $\Gamma \vdash_\Delta M$ *and* $\Delta \vdash M \longrightarrow M'$ *then* $\Gamma \vdash_\Delta M'$.

Proof. By case distinction on the step taken. The only interesting case is *(choice)*, which requires us to show that $\Gamma \vdash_\Delta \langle\, e_j \mid \mathsf{K} \,\rangle$ where $\Delta(a) = j$. Applying inversion on the typing derivation gives

Environment and Constraints

$$\begin{array}{llll} \text{Environment} & \Gamma & ::= & x : \tau, \Gamma \quad \text{binding} \\ & & \mid & a_i, \Gamma \quad \text{configuration} \\ & & \mid & \bullet \quad \text{empty} \end{array}$$

$$\begin{array}{llll} \text{Constraint} & c & ::= & \tau_1 \overset{\Gamma}{<:} \tau_2 \\ \text{Constraints} & C & ::= & c, \ldots \\ \text{Extended Types} & \tau & ::= & \ldots \mid \rho \end{array}$$

Constraint Gathering $\boxed{\Gamma \vdash e : \tau \mid C}$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \mid \emptyset}\ [\textsc{Var}] \qquad \frac{}{\Gamma \vdash 42 : \mathsf{Int} \mid \emptyset}\ [\textsc{Prim}]$$

$$\frac{\Gamma \vdash e : \tau \mid C \quad \overline{\Gamma \vdash e_i : \tau_i \mid C_i} \quad \rho \textit{ fresh}}{\Gamma \vdash e(\overline{e_i}) : \rho \mid C \cup (\bigcup_i C_i) \cup \{ \tau \overset{\Gamma}{<:} (\overline{\tau_i}) \to \rho \}}\ [\textsc{App}] \qquad \frac{\Gamma, \overline{x_i : \rho_i} \vdash e : \tau \mid C \quad \overline{\rho_i} \textit{ fresh}}{\Gamma \vdash \lambda(\overline{x_i}) \Rightarrow e : (\overline{\rho_i}) \to \tau \mid C}\ [\textsc{Abs}]$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \mid C_1 \quad \Gamma \vdash e_2 : \tau_2 \mid C_2 \quad \Gamma \vdash e_3 : \tau_3 \mid C_3 \quad \rho \textit{ fresh}}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \rho \mid C_1 \cup C_2 \cup C_3 \cup \{ \tau_1 \overset{\Gamma}{<:} \mathsf{Bool}, \tau_2 \overset{\Gamma}{<:} \rho, \tau_3 \overset{\Gamma}{<:} \rho \}}\ [\textsc{If}]$$

$$\frac{\overline{\Gamma, a_i \vdash e_i : \tau_i \mid C_i} \quad \rho \textit{ fresh}}{\Gamma \vdash a\langle \overline{e_i} \rangle : \rho \mid (\bigcup_i C_i) \cup \bigcup_i \{ \tau_i \overset{\Gamma, a_i}{<:} \rho \}}\ [\textsc{Choice}]$$

Fig. 6. Algorithmic typing of the $\lambda_{\diamond<:}$ calculus.

us the premise $\Delta(a) = j'$ and $\vdash e_{j'} : \tau$. Since M steps under the same resolution Δ it is typed in, and as Δ is well-formed, $\Delta(a)$ is required to be unique, and therefore we have that $j \equiv j'$. □

4 Inference

In the previous section, we defined the static and dynamic semantics of the $\lambda_{\diamond<:}$ calculus. Both were parametrized by a resolution Δ that we assumed to be fixed but arbitrarily provided by an oracle. This section, which constitutes the heart of the paper, formally shows how to perform inference, which proceeds in a few steps. We first gather constraints (Subsection 4.1), which are then solved (Subsection 4.2). Constraint solving, amongst others, results in error constraints $\mathcal{E}$, which are then processed by an overload solver (Subsection 4.3), which, if successful, yields a resolution Δ. Finally, we specialize (Subsection 4.4) the input program with respect to the resolution Δ.

4.1 Constraint Gathering

Figure 6 defines the algorithmic type system as syntax-directed typing rules that emit constraints.

4.1.1 Constraints. Unlike Hindley-Milner type systems that employ type equality constraints $\tau_1 \equiv \tau_2$, our type system, based on algebraic subtyping [Dolan and Mycroft 2017], relies on type inequality constraints like $\tau_1 <: \tau_2$, expressing that τ_1 is a subtype of τ_2. Our system augments these constraints with the *modal* configurations they are expected to be valid in. Consider

$$\mathsf{Int} \overset{a_1 b_2}{<:} \alpha$$

which specifies that Int is a lower bound of α in every world where the configuration $a_1 b_2$ holds. By contrast, an undecorated type inequality $\tau_1 <: \tau_2$ represents a *universal constraint*, a constraint valid across all possible worlds and configurations.

Similarly, we extend typing environments to be *modal*. Beyond traditional bindings of term variables to types such as $x : \mathsf{Int}$, they include singleton configurations such as a_1 or b_2 that track the current lexical configuration under which the term is typed. As usual for Fitch-style presentations [Borghuis 1994], we only admit exchange up to configurations. The function Γ restricts the context Γ to just the constituent configurations. For example, we compute:

$$\begin{aligned} x : \mathsf{Int},\ y : \mathsf{Str},\ a_1,\ z : \mathsf{Int},\ b_2 &= a_1 b_2 \\ a_2,\ b_1,\ c_1 &= a_1 b_2 c_1 \end{aligned}$$

4.1.2 Gathering. Figure 6 defines the algorithmic typing relation $\Gamma \vdash e : \tau \mid C$, where, as usual, Γ and e are to be read as inputs while τ and C are to be read as outputs. The syntax of types is extended to also include bi-unification variables ρ. Let us first describe rule Choice, which captures the core mechanism of our system. When encountering a choice $a\,\langle \overline{e_i} \rangle$, we first type check each alternative e_i under the additional assumption of selecting the respective configuration a_i. This will result in additional constraints C_i and a type τ_i for each alternative. We then create a fresh bi-unification variable ρ, which represents the result of the choice. Lastly, we make sure that the type of each alternative τ_i flows into this result by adding a constraint $\tau_i <: \rho$. Since this flow is conditional on the selected alternative, we annotate it with the accumulated configuration $\Gamma,\ a_i$.

The remaining inference rules are mostly standard and can be found in similar form in the literature on algebraic subtyping [Binder et al. 2022]. The distinguishing characteristic is that constraints are inherently variational and explicitly track the current lexical configuration via Γ. Annotating lexical configuration is only relevant for complex choices (Section 2.4). This way, type checking one alternative leads to constraints that only need to hold in this particular configuration. Remembering the configuration in the typing context enables nested configurations.

4.2 Constraint Solving

After gathering constraints, we proceed to solving them. Figure 7 describes a solver for inequality constraints adapted from Binder et al. [2022]. The state of our solver is a four-tuple $\langle\, C \mid \mathcal{S} \mid \mathcal{B} \mid \mathcal{E} \,\rangle$ consisting of C, the to-be-resolved variational constraints, $\mathcal{S}$, a cache of already resolved (that is, *seen*) variational constraints, $\mathcal{B}$, variational upper and lower bounds for type variables, such as

$$\mathcal{B}(\alpha) = \ldots,\ \tau_1^{\Gamma_1},\ \tau_2^{\Gamma_2} <: \alpha <: \tau_3^{\Gamma_3},\ \tau_4^{\Gamma_4},\ \ldots$$

and $\mathcal{E}$, the variational constraints that produced an error. Each step of the solver takes a state and produces a new one. The initial state of a solver is $\langle\, C \mid \emptyset \mid \emptyset \mid \emptyset \,\rangle$, where C are the constraints gathered in the previous phase. The solver proceeds as follows:

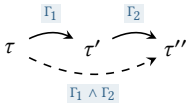
- Step CacheHit skips a constraint that is already in the cache.
- Step UpperBound (preferred over LowerBound if both apply) handles constraints $c = \alpha \overset{\Gamma}{<:} \tau$ where a type τ is an upper bound of a type variable α. We add τ together with its configuration Γ to α's upper bounds in $\mathcal{B}(\alpha)$. To ensure consistency with existing lower bounds, for every lower bound of α, τ' in configuration Γ' (from a previously resolved $\tau' <: \alpha$ in some configuration Γ'), we create a new variational constraint $\tau' <: \tau$ in the combined configuration $\Gamma' \wedge \Gamma$ since transitive flow requires both configurations at the same time.
- Step LowerBound is completely symmetrical to the previous rule: we remember the new lower bound and generate new constraints from the upper bounds to ensure consistency.

- Steps SubOK and SubKO use Sub to decompose a non-atomic constraint into smaller ones, such as splitting an inequality between functions into inequality between its results and inequalities between its arguments. If a decomposition is not possible (*e.g.*, when the arities of the function types differ), the Sub procedure returns **fail**, the constraint is classified as erroneous and added to the collection of error constraints $\mathcal{E}$.

Note that the solver does **not** stop early upon encountering an error, we just note down the error constraint and continue solving. This is crucial to our approach, as we need the failing variational constraints in order to perform the actual overload resolution by knowing which configuration failed, similarly to what Bhanuka et al. [2023] do for provenance in order to recover precise errors.

The definition of step UpperBound (and likewise LowerBound) is inspired by the following rule on the left that expresses the transitivity of variational subtyping:

$$\frac{\tau \overset{\Gamma_1}{<:} \tau' \qquad \tau' \overset{\Gamma_2}{<:} \tau''}{\tau \overset{\Gamma_1 \wedge \Gamma_2}{<:} \tau''}\ [\text{SAlg-Trans}]$$



If τ flows into τ' in configuration Γ_1, and τ' flows into τ'' in configuration Γ_2, then τ flows into τ'' in the combined configuration $\Gamma_1 \wedge \Gamma_2$. Semantically, τ then *only* flows into τ'' in the worlds where *both* configurations Γ_1 *and* Γ_2 hold. This formalizes the graphical intuition applied in Section 2 and sketched on the right, where we transitively follow the edges of the variational flow graph, computing the conjunction of the involved configurations.

4.2.1 Intermezzo: Optional Bounds Unification. The solver described in the previous section recognizes and reports type errors when a type flows into another incompatible one. For example, adding Str^{b_2} as upper bound to β in

$$\mathsf{Int}^{a_1} <: \beta <:$$

induces a flow from Int to Str in configuration $a_1 b_2$, yielding an error constraint. As is usual for algebraic subtyping, however, when faced with $\mathsf{Str} <: \beta$ in configuration b_2, the solver simply adds it as additional lower bound, producing two incomparable lower bounds:

$$\mathsf{Int}^{a_1},\ \mathsf{Str}^{b_2} <: \beta <:$$

In a system with union types, as is the case for algebraic subtyping, this means that we can replace β in negative positions with a union type $\mathsf{Str} \vee \mathsf{Int}$. However, if the reader wishes to apply our framework to a system like Hindley-Milner which does not allow for such types, we can simply unify the lower bounds pairwise (and symmetrically for the upper bounds). Unification may reveal new error constraints, which are, again, relevant for later overload resolution. In the example above, unifying the two lower bounds generates two new erroneous flows $\mathsf{Str} \overset{a_1 b_2}{<:} \mathsf{Int}$ and $\mathsf{Int} \overset{a_1 b_2}{<:} \mathsf{Str}$, in the combined configuration. This process can also yield absurd error constraints; for instance $\mathsf{Bool} \overset{a_1 a_2}{<:} \mathsf{Double}$ can be freely discarded, since no world makes both a_1 *and* a_2 true simultaneously.

This step is entirely optional. An implementor may choose to switch from a type-inequality approach to a type-equality approach in their implementation, but this decision entails a critical trade-off that directly impacts overload resolution capabilities as detailed in Section 6.2.

Constraint Solving

Constraint Solver Step: $\langle\, C \mid \mathcal{S} \mid \mathcal{B} \mid \mathcal{E} \,\rangle \rightsquigarrow \langle\, C' \mid \mathcal{S}' \mid \mathcal{B}' \mid \mathcal{E}' \,\rangle$

$$\frac{c \in \mathcal{S}}{\langle\, c,\ C \mid \mathcal{S} \mid \mathcal{B} \mid \mathcal{E} \,\rangle \rightsquigarrow \langle\, C \mid \mathcal{S} \mid \mathcal{B} \mid \mathcal{E} \,\rangle}\ [\textsc{CacheHit}]$$

$$\frac{c \notin \mathcal{S} \quad c = \alpha \overset{\Gamma}{<:} \tau \quad \mathcal{B}(\alpha) = lbs <: \alpha <: ubs \quad C' = \left\{ \tau' \overset{\Gamma' \wedge \Gamma}{<:} \tau \;\middle|\; \tau'^{\Gamma'} \in lbs \right\}}{\langle\, c,\ C \mid \mathcal{S} \mid \mathcal{B} \mid \mathcal{E} \,\rangle \rightsquigarrow \langle\, C \cup C' \mid c,\ \mathcal{S} \mid \mathcal{B}\left[\alpha \mapsto lbs <: \alpha <: ubs,\ \tau^{\Gamma}\right] \mid \mathcal{E} \,\rangle}\ [\textsc{UpperBound}]$$

$$\frac{c \notin \mathcal{S} \quad c = \tau \overset{\Gamma}{<:} \beta \quad \mathcal{B}(\beta) = lbs <: \beta <: ubs \quad C' = \left\{ \tau \overset{\Gamma \wedge \Gamma'}{<:} \tau' \;\middle|\; \tau'^{\Gamma'} \in ubs \right\}}{\langle\, c, C \mid \mathcal{S} \mid \mathcal{B} \mid \mathcal{E} \,\rangle \rightsquigarrow \langle\, C \cup C' \mid c,\ \mathcal{S} \mid \mathcal{B}\left[\beta \mapsto lbs,\ \tau^{\Gamma} <: \beta <: ubs\right] \mid \mathcal{E} \,\rangle}\ [\textsc{LowerBound}]$$

$$\frac{c \notin \mathcal{S} \quad c = \tau_1 \overset{\Gamma}{<:} \tau_2 \quad \tau_1,\ \tau_2 \notin TyVar \quad \mathrm{Sub}(c) = C'}{\langle\, c,\ C \mid \mathcal{S} \mid \mathcal{B} \mid \mathcal{E} \,\rangle \rightsquigarrow \langle\, C \cup C' \mid c,\ \mathcal{S} \mid \mathcal{B} \mid \mathcal{E} \,\rangle}\ [\textsc{SubOK}]$$

$$\frac{c \notin \mathcal{S} \quad c = \tau_1 \overset{\Gamma}{<:} \tau_2 \quad \tau_1,\ \tau_2 \notin TyVar \quad \mathrm{Sub}(c) = \textbf{fail}}{\langle\, c,\ C \mid \mathcal{S} \mid \mathcal{B} \mid \mathcal{E} \,\rangle \rightsquigarrow \langle\, C \mid c,\ \mathcal{S} \mid \mathcal{B} \mid c,\ \mathcal{E} \,\rangle}\ [\textsc{SubKO}]$$

Non-Atomic Constraint Decomposition: $\mathrm{Sub}(c) = C$ or **fail**

$$\mathrm{Sub}((\tau_1, \ldots, \tau_n) \to \tau \overset{\Gamma}{<:} (\tau'_1, \ldots, \tau'_n) \to \tau') = \bigcup_i \{\, \tau'_i \overset{\Gamma}{<:} \tau_i \,\} \cup \{\, \tau \overset{\Gamma}{<:} \tau' \,\}$$

$$\mathrm{Sub}(\tau \overset{\Gamma}{<:} \top) = \emptyset \qquad \mathrm{Sub}(\tau \overset{\Gamma}{<:} \tau) = \emptyset$$
$$\mathrm{Sub}(\bot \overset{\Gamma}{<:} \tau) = \emptyset \qquad \mathrm{Sub}(-) = \textbf{fail}$$

Fig. 7. Rules for solving and decomposing constraints.

4.3 Overload Resolution

Now that we have gathered all the failing variational constraints, the *errors* $\mathcal{E}$, we can finally use them to resolve overloads, ideally producing a well-formed resolution Δ. Semantically, the gist is that we look at all possible worlds (*i.e.*, all complete configurations), and then for each error eliminate all of the worlds where this configuration holds, since they are invalid. Afterwards, we are left with only the worlds which are valid. There are three possibilities based on their cardinality: zero, one, or many.

If there are zero worlds left, then there is no way to consistently resolve the overload in any way whatsoever, and we fail, reporting the problem to the user. If there is a single world left, we can use it as a resolution Δ, noting that it is well-formed as it chooses a single alternative for each dimension, allowing us to resolve each choice. Finally, if there are two or more worlds left, then the overloads are ambiguous. In this paper, we decide to not entertain the notion of a "better overload" (see Section 6.4), which means we again fail as we cannot provide a unique solution.

4.3.1 Resolving Efficiently. Since in our approach, the actual act of resolving is separate from type checking, we can reuse existing tools to support the process of overload resolution. As one possible implementation strategy, we have identified using Binary Decision Diagrams (BDD) as popularized by Akers [1978] to encode the resolution problem. BDDs appear to be well-suited for

our problem of overload resolution. Firstly, binary operations (such as conjunction and disjunction) on BDDs run in polynomial time [Bryant 1986] admitting a straightforward encoding (see below). Secondly and crucially, BDDs can count the number of satisfying assignments in polynomial time [Bryant 1986], which allows us to efficiently identify whether there exists a *unique* solution. We can encode overload resolution as a BDD representing the conjunction of all valid selections with all inconsistency constraints:

$$\mathit{selections} \ \wedge \neg \mathit{inconsistency}_1 \ \wedge \neg \mathit{inconsistency}_2 \ \wedge \ldots$$

Encoding Possible Selections. As a first step, we need to encode all possible selections. For this, we syntactically identify all choices present in a program together with their dimensions and numbers of alternatives. To express them as a BDD, one could use a *one-hot encoding*, where each alternative such as a_1 becomes a separate variable, with additional constraints ensuring that exactly one alternative is selected per dimension. However, this would result in $O(k)$ variables and a formula of size $O(k^2)$ per dimension, where k is the number of its alternatives. We can do better by using a *binary encoding* [Frisch and Peugniez 2001]. Each dimension (*e.g.*, a) is encoded using $\lceil log_2\ k \rceil$ boolean variables representing the bits of the selected index. For example, if $k = 4$, instead of variables for each a_1, a_2, a_3, and a_4, we have a variable for each *bit* of the selection. The selection a_3 (third index) is then represented by $10_2 \approx \mathrm{a}_{bit_1} \wedge \neg \mathrm{a}_{bit_0}$. When k is not a power of two, we add a formula of size $O(k\ log\ k)$ ensuring that the encoded index is less than k.

Encoding Inconsistencies from Variational Constraints. The errors $\mathcal{E}$ gathered during constraint solving are variational constraints specifying invalid flows. Each such constraint identifies configurations that are invalid. These become the inconsistencies we must rule out.

Error Constraint	Inconsistent World	Binary Encoding	Normalized Disjunctive Clause
Int $\overset{a_3b_2}{<:}$ Str	$\neg(\mathrm{a}_3 \wedge \mathrm{b}_2)$	$\neg((\mathrm{a}_{bit_1} \wedge \neg \mathrm{a}_{bit_0}) \wedge \mathrm{b}_{bit_0})$	$\neg \mathrm{a}_{bit_1} \vee \mathrm{a}_{bit_0} \vee \neg \mathrm{b}_{bit_0}$

For each failing variational constraint such as the above constraint on the left, we extract the invalid configuration it contains, here $\mathrm{a}_3\mathrm{b}_2$. We then encode this configuration using our binary representation (assuming $k = 4$ for a and $k = 2$ for b) and negate it to form an inconsistency clause that rules out the worlds that would satisfy the invalid configuration in question. Finally, we convert each clause into disjunctive normal form on the right, so that all such inconsistencies together give us a formula in CNF.

Resolution via BDD Solving. Taking the conjunction of valid selections and all inconsistencies as a BDD, we obtain the overall BDD representing the valid worlds for our overloaded program. Using off-the-shelf BDD solvers, we can efficiently count satisfying assignments (corresponding to valid worlds) to determine if there are zero, one, or more solutions.

If there is exactly one world left, we can efficiently get a well-formed resolution Δ by examining the unique satisfying assignment of the BDD. If there are multiple valid worlds, we can enumerate each resolution corresponding to a complete world in polynomial time per solution [Bryant 1986]. However, there might be exponentially many complete worlds, so enumeration requires time proportional to the number of solutions.

Again, we want to highlight that using an external solver is very straightforward when there is a clean, modular phase separation between the type checker and the overload resolution itself. We also believe that there is plenty of room for optimization such as better encodings of "exactly-one" constraints [Björk 2011]. In this subsection, we are merely demonstrating the use of off-the-shelf tools in a straightforward fashion enabled by the newly gained modularity of overload resolution.

4.4 Specializing Resolutions

Now that we have a well-formed resolution Δ, we can use it to replace each choice $a\,\langle\overline{e_i}\rangle$ with the chosen alternative. We call this process *specialization*. There is only one interesting case, all of the other cases are homomorphic:

$$[\![\ a\,\langle\overline{e_i}\rangle\ ]\!]_\Delta = e_j \qquad \textit{where } \Delta(a) = j$$

The whole goal of the paper is to, at compile time, replace each overload with a specific alternative distinguished by its type. Specialization is a way to achieve this: once we have a well-formed resolution Δ, we can specialize every expression $[\![\ e\ ]\!]_\Delta$ to erase all choices.

Specialization preserves types and semantics. Preservation of types additionally states that the resulting program is fully specialized and thus independent of any resolution Δ'. We use $M \to^{0,1} M'$ to denote that either $M = M'$ or $M \to M'$.

Theorem 4.1 (Type preservation).
If $\Gamma \vdash_\Delta e : \tau$, *then* $\forall \Delta', \Gamma \vdash_{\Delta'} [\![\ e\ ]\!]_\Delta : \tau$.

Proof. By case distinction on e. The only interesting case is $a\,\langle\overline{e_i}\rangle$. Given $\Delta(a) = j$, the original type must have been τ as per the type derivation via rule Choice. After specialization, the type still needs to be the same, as we have inlined the choice, reusing its typing derivation.

Since specialization removes all choices, the resulting type derivation never uses resolution Δ, therefore we can use any resolution Δ' whatsoever for the new typing derivation. □

Theorem 4.2 (Semantic preservation).
If $\Delta \vdash M \to M'$, *then* $\Delta \vdash [\![\ M\ ]\!]_\Delta \to^{0,1} [\![\ M'\ ]\!]_\Delta$.

Proof. By case distinction on e, where $M = \langle\ e \mid \mathsf{K}\ \rangle$. The only interesting case is $a\,\langle\overline{e_i}\rangle$. Given $\Delta(a) = j$, the original step must have been *(choice)*, so a step to $\langle\ e_j \mid \mathsf{K}\ \rangle$. When we specialize, we need not perform this step. All other steps will still be performed, hence $[\![\ M\ ]\!]_\Delta \to^{0,1} [\![\ M'\ ]\!]_\Delta$. □

5 Evaluation

We evaluate practicality through two complementary studies: performance comparison against Swift on real-world overloading challenges and scalability analysis on parameterized synthetic benchmarks. Our prototype (implemented in Rust) parses programs, performs type inference, then resolves overloads using BDD encoding via the BDD library of the tool AEON [Beneš et al. 2020]. All benchmarks were ran using `hyperfine` [Peter 2024] on an Apple M1 Pro with 32 GiB of RAM, running macOS 15.4. Each benchmark was executed with three warmup runs, then ran at least ten times to obtain the arithmetic mean execution time.

5.1 Performance Comparison with Swift

To get a first impression on how our system performs relative to the state-of-the-art, we compare it against Swift 5.8's `-typecheck` mode, a production-ready language that uses Hindley-Milner-based inference with a constraint-based type system and overloading.

5.1.1 Benchmark Description. Our benchmark suite consists of examples sourced from blog posts related to overloading performance (**3sat**, **uri**, **addneg**) as well as a number of challenging but realistic benchmarks (**recur**, **dist**, **cps**). When porting the examples to Swift, we tried to eliminate unrelated influences like implicit casts: an `Int` type is our own empty struct, `addInt` is our own function that ignores arguments and returns the `Int` struct, etc.

(**3sat**) Lippert [2007]'s 3-SAT encoding using overloaded boolean operations with singleton `T`/`F` types, demonstrating NP-hardness of overload resolution—variant *hard* uses a near-unsatisfiable formula requiring extensive search.

Table 1. Runtime of the performance benchmarks in milliseconds, lower is better.

	3sat		**uri**		**addneg**		**recur**	**dist**	**cps**		
	orig	hard	orig	big	orig	big			10	20	100
$\lambda_{\diamond<:}$	2.7	3.8	2.4	10.1	5.8	495.7	2.4	2.6	3.0	4.6	66.3
Swift	126.0	130.2	131.8	530.6	150.3	2783.1	123.8	124.9	286.5	TO	TO

(**uri**) Hooper [2024]'s Swift performance case where (accidentally) mixed string/integer concatenation creates exponential overload combinations that are not satisfied by any world—variant *big* extends the concatenation chain.

(**addneg**) Gallagher [2016]'s "expression too complex" example with nested arithmetic using binary addition, unary negation, and overloaded integer literals across multiple numeric types—variant *big* increases expression depth and overload count.

(**recur**) Recursive `showInt` function defining overloads of itself during its own resolution.

(**dist**) Euclidean distance on 3D vectors requiring consistent numeric types across coordinate access, arithmetic, and `sqrt`.

(**cps**) Higher-order composition of endomorphisms in continuation-passing style, where nested `compose(compose(..., f), f)` must resolve `f` consistently throughout—parametrized by call stack depth N.

5.1.2 Results. Table 1 shows the runtime of the performance benchmarks in milliseconds. The mark TO denotes that Swift times out in the larger (**cps**) benchmark after around 9 minutes, reporting that *"the compiler is unable to type-check this expression in reasonable time; try breaking up the expression into distinct sub-expressions"*. Furthermore for $N = 100$ of (**cps**), our implementation spends longer constraint solving (48 ms) than on overload resolution (27 ms).

A notable difference between our system and Swift is that Swift cannot infer the types of function parameters in benchmarks (**recur**) and (**dist**), thus requiring an explicit annotation, yet still being slower than our prototype. In the (**dist**) benchmark, our system also has an explicit annotation to constrain it to exactly one result due to lateral flows (see Section 6.2).

The Swift compiler seems to have a large constant overhead, likely due to its production-ready nature. Otherwise, it scales similarly to our prototype with the exception of the (**cps**) benchmark where it exponentially explores incompatible $f : \mathsf{Str} \to \mathsf{Int}$ paths despite only $f : \mathsf{Int} \to \mathsf{Int}$ being valid, thus leading to the aforementioned timeout. We believe these results demonstrate that the BDD-based approach of our implementation is practical as it handles all benchmark scenarios, including ones that cause Swift to either timeout or require manual annotations.

5.2 Scalability Analysis

To gain further understanding of the scalability of our system, we benchmark our implementation with a parameterized family of programs that systematically stresses overload resolution.

5.2.1 Synthetic Benchmark Design. Each program instantiates M overloaded additions with homogeneous signatures $(T,\ T) \to T$, then applies the overloaded symbol in N nested calls (Figure 8a), where each add_i represents a fresh overload choice among M alternatives. We also evaluate based on cardinality of valid worlds (zero, one, many). The many-solutions variant admits M distinct typings (one per overload). The one-solution variant constrains the context (*e.g.*, adding a literal: ... + 0 : Int at the end of the chain). The zero-solutions variant introduces a type conflict (*e.g.*, adding together two literals of disjoint types: ... + 0 : Int + "*hello*" : Str).

Source Program

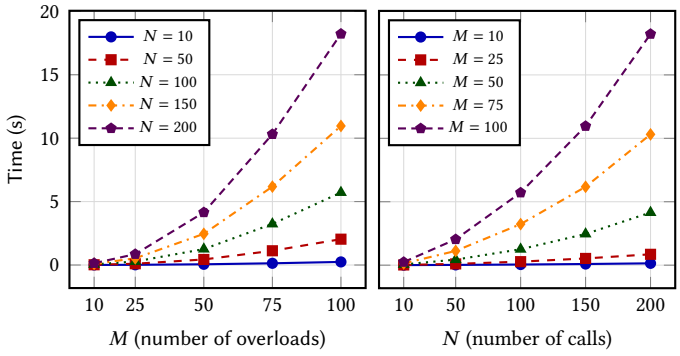
$x \Rightarrow x + x + x + \ldots$

Variational Core

let add_1 = $a\langle \texttt{addInt}, \ldots, \texttt{addT}_M \rangle$
let add_2 = $b\langle \texttt{addInt}, \ldots, \texttt{addT}_M \rangle$
...
let add_N = $z\langle \texttt{addInt}, \ldots, \texttt{addT}_M \rangle$

$\lambda(x) \Rightarrow add_1(x,\ add_2(x,\ \ldots))$

(a) Schematic of the parametric benchmark programs (overload count N, repeated additions M).



(b) Runtime vs. M for fixed N.

(c) Runtime vs. N for fixed M.

Fig. 8. Overall benchmark of runtime scaling alongside the schematic of the program structure. Since there is no notable difference between the variants with zero, one, and many solutions, the plots show only the runtime of the zero-solutions variant.

5.2.2 Results. The results in Figure 8 suggest that our implementation scales quadratically in overload count M and repeated additions N, with time spent on overloading growing faster in M than in N. Note that for $M = N = 10$, the runtime is around 5 ms. We only show the zero-solutions variant; the running times of the other variants differ less than 1% from the zero-solutions variants' running times.

6 Discussion

Having formally presented the $\lambda_{\bowtie:}$ calculus and our process of overload resolution, in this section we now discuss different aspects and possible design decisions relevant for language designers thinking about adopting our approach.

6.1 Overloading of Different Kinds

In this paper, we show how we can model various kinds of overloading: ordinary function argument-type and operator overloading (Section 2.1), return-type overloading (Section 2.2), as well as value and arbitrary expression overloading (Section 2.4). Our system also supports other kinds of overloading such as *arity overloading*, since the constraint solver produces an error constraint when resolving an arity mismatch.

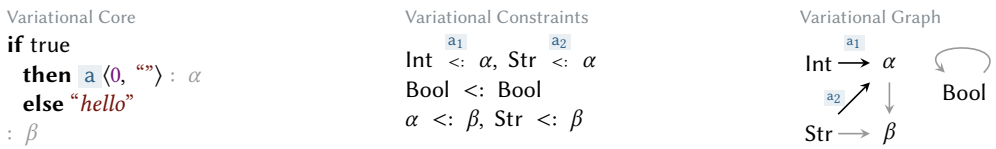
Because our system supports type-directed overloading of arbitrary expressions, it can also model features such as uniform function call syntax (UFCS) as supported by D, Koka [Leijen 2014] and Effekt [Brachthäuser et al. 2020]. When a parser encounters *person.name*, it can desugar it into a new choice with a fresh dimension, where the different alternatives reflect its supported type-distinguished meanings, for example, field access, method call, or a plain function call:

$$a\langle \texttt{FieldAccess}(person,\ name), \texttt{MethodCall}(person,\ name), \texttt{FunctionCall}(name,\ person) \rangle$$

This way the same mechanism can be used to overload syntax and names under one common framework, potentially simplifying the implementation. It also trivially enables the interaction between overloaded syntax and names (whether desired or not), while still separating the phases.

6.2 Overloading over Lateral Flows

In Subsection 4.2.1, we already hinted at the fact that our system, *a priori*, only detects errors if one type flows into another incompatible type. In contrast, two conflicting lower (or upper) bounds will not lead to an error constraint and thus also not influence overload resolution. Concretely, let us consider the following program and its constraints:



Notice that if we take the system as-is, both worlds a_1 and a_2 remain valid, no matter what. Computing the transitive lower bounds of β gives us:

$$\mathsf{Int}^{a_1},\ \mathsf{Str}^{a_2},\ \mathsf{Str} <: \beta$$

We can observe that both Int^{a_1} and Str (which holds in every world) flow into β. They do not flow into each other, but next to each other, similar to a "level-1 error" of Bhanuka et al. [2023], a *lateral* flow. Although β is always at least Str, knowing this is not yet enough to resolve the overload!

In a system with subtyping, this is the best we can do, as it could be possible that the type of β is a true set-theoretic type like $\mathsf{Int} \vee \mathsf{Str}$. However, in a Hindley-Milner-style approach, we can unify bounds [Bhanuka et al. 2023] to, in a sense, make the variational graph undirected and thus obtain an error constraint $\mathsf{Int} \overset{a_1}{<:} \mathsf{Str}$, resolving overloads in this example to a_2.

6.3 Let Generalization as a Boundary

Overloading, as described in this paper, allows some "spooky action at a distance". Consider an example that uses the same dimension a at opposite ends of a program. If we specialize one choice, we necessarily also resolve the other, no matter its distance.

$$a\,\langle e_1,\ e_2\rangle \qquad \ldots \qquad a\,\langle e_3,\ e_4\rangle$$

Curtailing a choice's scope benefits both the solver that has to resolve it, and the programmer who has to reason about it. A very natural way of limiting scope is `let` generalization. In our experiments, we have found it useful to delimit each choice by the top-level declaration it is declared in. In addition to the benefits above, it also presents a natural way of implementing `let` generalization in our system by fitting into the standard type-inference-with-constraints paradigm: Starting with a top-level definition, we gather its constraints, solve them, *resolve the overloads* based on the error constraints, and move on to the next top-level definition.

This way, overloading never reaches beyond a generalized type, and all overloading errors are confined to a single top-level definition, enabling greater error recovery. Moreover, this strategy is straightforward to implement and preserves the theoretical properties of the system. Using let generalization as a boundary differs from type classes [Wadler and Blott 1989], which reify the constraints as type-level predicates [Jones 1992] when generalizing. By contrast, overloads here are guaranteed to be statically resolved.

6.4 A Better Overload?

So far, we required that a unique resolution exists and therefore rejected the idea of resolving ambiguity by preferring one possible world over another. Nonetheless, most languages with overloading do implement language-specific tiebreaker rules. In languages with subtyping, one may prefer a more general or a more specific overload; this notion of a "better" overload, however,

introduces coherence and stability concerns: adding a new, more general alternative can result in it being preferred by the solver over the existing more specific alternative (or vice versa).

Haskell takes a different approach—*defaulting* [Marlow 2010]. To enable overloading numeric literals, GHC wraps every mention of an integer literal 123 into a call to *fromInteger* of type Num $\alpha \Rightarrow$ Integer $\rightarrow \alpha$ using a Num type class. In order to allow easy interactive use, Haskell adopts ad-hoc defaulting to Integer and Double for the Num type class to resolve ambiguities.

In this paper, we have reduced the actual overload resolution based on erroneous variational constraints into a separate modular component as opposed to it being scattered throughout the whole type checking process. If a language author wishes to concern themselves with the notion of a better overload, they either have to define additional rules in their overload resolution solver to describe their preferences, or first run our overload resolution solver and post-process the results by eliminating worlds their semantics deem invalid.

6.5 Towards Nicer Overloading Errors

In Section 2.2, we showed that our approach can produce conceptually nicer error messages for overloading ambiguities than existing systems in languages like Haskell, Rust, F#, Swift, Koka, or Effekt by considering the whole program flow across worlds at once and by rejecting impossible alternatives outright. In Section 2.3, we sketched one such error, explaining why an alternative was not selected using a flow-based error message [Bhanuka et al. 2023]. Since every overloading error corresponds to an invalid program flow, we could also present the subgraph induced by the invalid flow in the variational graph to the user. Likewise, on ambiguity, we could only show the subgraph induced by the valid variational constraints, asking the user to resolve the ambiguity manually. Finally, we believe our world-based model offers a useful starting point for explaining overloading to programmers, thus serving as a mental model.

7 Related Work

We now compare our solution to alternative approaches for overloads and variational type checking.

7.1 Type Classes

Type classes by Wadler and Blott [1989], later extended by Jones [1992] and Odersky et al. [1995] are a popular feature to allow overloading in a principled, parametric way. To model overloaded addition using type classes, one first declares a generic type class like class Add α { $add : (\alpha, \alpha) \rightarrow \alpha$ }, open to extension, then instances like instance Add Int { add = addInt }. Consequently a function doubling its argument, $\lambda(x) \Rightarrow x + x$, has type Add $\alpha \Rightarrow \alpha \rightarrow \alpha$, denoting that it works for any type α implementing the Add type class.

Unlike the overloads in this paper, type classes are inherently open—new instances can be defined anywhere. Our choices are finite and closed, restricted to the alternatives contained within them. Type classes translate to function records passed as arguments [Wadler and Blott 1989], potentially at runtime, while our overloads are specialized at compile time and thus never appear at runtime. Finally, type classes require all instances to be generalizable to a single type, whereas our choices can contain expression with types difficult to generalize such as a $\langle 0, \lambda(x) \Rightarrow x,$ addInt$\rangle$.

7.2 Overloading in Swift

Swift is a programming language with a Hindley-Milner-style type system and support for overloading. The type checker [Swift Language Team 2024] introduces fresh type variables for each overload and then represents overloaded declarations by disjunction constraints, such as:

$$\alpha := (\text{Int}, \text{Int}) \rightarrow \text{Int} \quad \text{or} \quad \alpha := (\text{Double}, \text{Double}) \rightarrow \text{Double}$$

In contrast, we represent overloads using variational choices and model type flow via inequalities rather than equality constraints. They state that the solution space of the constraint solver is exponential in the worst case.

7.3 Variational Type Checking

Kenner et al. [2010] present *type chef*, a variational type checker for the C programming language. Their goal is to type check the possible variants induced by conditional-compilation directives (*e.g.*, `#ifdef`). Erwig and Walkingshaw [2011] introduce the choice calculus as a rigorous and principled way to model software product lines. Chen et al. [2014] describe variational type inference for the choice calculus as a means to type check variational programs. Our system is greatly inspired by the prior work on variational calculi and in particular by the work of Erwig and Walkingshaw [2011], sharing their notion of choices and dimensions. However, they support language constructs for creating new dimensions [Erwig and Walkingshaw 2011] and local selection [Erwig et al. 2013].

In contrast, our $\lambda_{\diamond:}$ calculus focuses on the use case of overload resolution and thus only features choices and leaves the creation of dimensions external to the program. It is however conceivable to model user-provided overload resolution as local selection. Crucially, the goal of variational type checking is, in our parlance, to check if *all* worlds are valid, independently of any configuration-specific decisions, whereas we aim to identify exactly one valid world.

7.4 Overloads as Intersections, Merges, or Disjoint Unions

Castagna [2024] describes a commonly used way to encode overloading by using intersection types. While we model overloaded negation as a term $\texttt{a}\langle\texttt{negInt}, \texttt{negBool}\rangle$, they express this as a type $(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool})$. However, algebraic subtyping [Dolan and Mycroft 2017] and its implementations [Parreaux 2020] use a distributive lattice simplifying the intersection type to $(\mathsf{Int} \vee \mathsf{Bool}) \to (\mathsf{Int} \wedge \mathsf{Bool})$. This breaks type preservation as $\texttt{a}\langle\texttt{negInt}, \texttt{negBool}\rangle(42)$ reduces to $\mathsf{Int} \wedge \mathsf{Bool}$. In contrast, our solution works both in systems with type equality constraints and in systems with type inequality constraints, and is applicable even in the context of algebraic subtyping and its distributive lattice.

To remedy the flaws of the intersection encoding, Rioux et al. [2023] present a formal treatment of a principled merge operator $e_1 \;||\; e_2$ which combines the two expressions e_1 and e_2 into a single expression that ought to behave like both of the expressions in every context they are in. More specifically, it allows both merging records with disjoint labels, and functions with disjoint input types, where the latter can be used to model function overloading.

Rehman et al. [2022] describe a calculus with disjoint union types for type-based switches with disjoint cases, modelling nullable types and certain kinds of overloading. They exclude function type overloads due to subtyping: no two functions are disjoint as common subtypes like $\top \to \bot$ always exist. In contrast, our system supports overloading function arguments and emphasizes the *call-site* rather than the *declaration-site*: our system has no concept of an overloaded declaration, all conflicts are modelled solely at the call-site.

7.5 Overloads at Runtime

Castagna et al. [1992] present a calculus $\lambda\&$ modelling overloading of functions using a term $M \;\&\; N$ as a restricted form of the merge operator. Its introduction form adds a new overload N to an existing overloaded function M, and its elimination form is a special application of overloaded functions to their arguments. As opposed to our system, they use subtyping in order to select the best possible overload. Their system uses a type-guided β reduction at runtime without support for nested overloading, using overloading with the notion of a better overload to implement inheritance. In contrast, our system is designed to erase all overloading at compile time.

7.6 Dot Overloading

Languages such as Haskell [Gundry 2017; Mitchell and Fletcher 2020] and Flix [Tan and Madsen 2025] overload dot notation via type classes, where $\lambda(x) \Rightarrow x.age$ has a generalized type like HasField "*age*" $\alpha \Rightarrow \alpha \rightarrow$ Int or $\text{Dot}_{\text{age}}\ \alpha \Rightarrow \alpha \rightarrow$ Int respectively. This allows the compiler to desugar all dot notation into a type class method. Although elegant, we believe this abstraction to be leaky as it is exceedingly simple to write an accidentally over-generalized program. In this paper, we present an overloading system that can also be used to overload the dot by desugaring dot-access into an overload (as outlined in Section 6.1), sacrificing some expressivity as the user cannot easily specify their own dot overload without additional language features.

8 Conclusion and Future Work

In this paper, we have presented an approach that combines algebraic subtyping with variational type checking, which we believe distills the *essence of overloading*. Our approach supports a clean separation into constraint collection, constraint solving, and overload resolution. Importantly, we treat solving the error constraints as a modular component, which localizes performance improvements and potentially supports different strategies to resolve overloads. Building on algebraic subtyping, our framework provides an intuitive notion of flow and is compatible with related work on improved error messages [Bhanuka et al. 2023]. Building on variational type checking, reasoning about possible worlds and the flows in these worlds provides an interesting mental model.

To conclude, we identify some avenues of future work that build up on the results of this paper.

Solving Independent Components Independently. In programs with multiple independent overloaded sub-expressions such as `print(1 + 2)` and `print(3 + 4)`, the error constraints naturally partition into disjoint variable sets. Their dimensions never co-occur in configurations of error constraints, indicating failure in one group provides no information about the other. By preprocessing to identify these independent components, one can decompose the overall BDD into several smaller ones, reducing resolution complexity.

Back to Conditional Compilation. In this paper, we focus on applying the techniques and concepts of variational type checking onto the domain of overloading, the biggest difference being that variational type checking verifies that all worlds are valid, whereas we are verifying that exactly one single world is valid. We believe that our system might be adaptable to the original domain of variational type checking, conditional compilation, by extending our system with an "always-choice".

Modal Type Theory. In this paper, we model the semantics of our approach with intuitive "worlds", hinting at possible world semantics pioneered by Carnap [1946]. As future work, we would like to further explore the connection to modal logics, type systems, and their semantics. We also note that our reasoning implicitly uses a graded diamond modality, focusing on exactly one world as opposed to at least one world of the standard diamond modality. This also relates to conditional compilation, which seems to use a box modality, focusing on all worlds.

Residualising Choices into Types. Following up from Section 6.3, in this paper, we currently do not generalize choices. However, it might be feasible to follow Chen et al. [2014] and residualize the choices into variational types when generalizing, reifying the choice $a\langle \texttt{negInt}, \texttt{negBool}\rangle$ into a type $a\langle \text{Int} \rightarrow \text{Int},\ \text{Bool} \rightarrow \text{Bool}\rangle$, generalizing over the dimension a, thus ending up with a modal type.

9 Data-Availability Statement

We submit an artifact containing the example programs and the benchmarked programs, the benchmarks themselves, our prototype implementation of the calculus $\lambda_{\diamond<:}$ and its type inference pipeline, and finally a web playground where a reader can input a term in Variational Core, and get the resulting type, constraints, and bounds, in addition to the result of overload resolution [Beneš and Brachthäuser 2025].

References

S. B. Akers. 1978. Binary Decision Diagrams. *IEEE Trans. Comput.* 27, 6 (June 1978), 509–516. doi:10.1109/TC.1978.1675141

Henk P. Barendregt. 1984. *The Lambda Calculus: Its Syntax and Semantics. Revised Edition.* North-Holland, Amsterdam, The Netherlands.

Nikola Beneš, Luboš Brim, Jakub Kadlecaj, Samuel Pastva, and David Šafránek. 2020. AEON: Attractor Bifurcation Analysis of Parametrised Boolean Networks. In *Computer Aided Verification (Lecture Notes in Computer Science, Vol. 12224)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer, 569 – 581. doi:10.1007/978-3-030-53288-8_28

Jiří Beneš and Jonathan Immanuel Brachthäuser. 2025. *Artifact of the paper 'The Simple Essence of Overloading'.* doi:10.5281/zenodo.16928381

Ishan Bhanuka, Lionel Parreaux, David Binder, and Jonathan Immanuel Brachthäuser. 2023. Getting into the Flow: Towards Better Type Error Messages for Constraint-Based Type Inference. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 237 (oct 2023), 29 pages. doi:10.1145/3622812

David Binder, Ingo Skupin, David Läwen, and Klaus Ostermann. 2022. Structural refinement types. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Type-Driven Development* (Ljubljana, Slovenia) *(TyDe 2022)*. Association for Computing Machinery, New York, NY, USA, 15–27. doi:10.1145/3546196.3550163

Magnus Björk. 2011. Successful SAT Encoding Techniques. *Journal on Satisfiability, Boolean Modelling and Computation* 7, 4 (2011), 189–201. arXiv:https://journals.sagepub.com/doi/pdf/10.3233/SAT190085 doi:10.3233/SAT190085

Valentijn Anton Johan Borghuis. 1994. *Coming to terms with modal logic: On the interpretation of modalities in typed λ-calculus.* Dissertation. Technische Universiteit Eindhoven, Eindhoven.

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 126 (Nov. 2020). doi:10.1145/3428194

Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.* 35, 8 (Aug. 1986), 677–691. doi:10.1109/TC.1986.1676819

Rudolf Carnap. 1946. Modalities and Quantification. *Journal of Symbolic Logic* 11, 2 (1946), 33–64. doi:10.2307/2268610

Giuseppe Castagna. 2024. *Programming with Union, Intersection, and Negation Types.* Springer International Publishing, Cham, 309–378. doi:10.1007/978-3-031-34518-0_12

Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. 1992. A calculus for overloaded functions with subtyping. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming* (San Francisco, California, USA) *(LFP '92)*. Association for Computing Machinery, New York, NY, USA, 182–192. doi:10.1145/141471.141537

Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2014. Extending Type Inference to Variational Programs. *ACM Trans. Program. Lang. Syst.* 36, 1, Article 1 (mar 2014), 54 pages. doi:10.1145/2518190

Stephen Dolan and Alan Mycroft. 2017. Polymorphism, Subtyping, and Type Inference in MLsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 60–72. doi:10.1145/3009837.3009882

Martin Erwig, Klaus Ostermann, Tillmann Rendel, and Eric Walkingshaw. 2013. Adding Configuration to the Choice Calculus. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems.* ACM, New York, NY, USA, 13. doi:10.1145/2430502.2430520

Martin Erwig and Eric Walkingshaw. 2011. The Choice Calculus: A Representation for Software Variation. *ACM Trans. Softw. Eng. Methodol.* 21, 1, Article 6 (dec 2011), 27 pages. doi:10.1145/2063239.2063245

Alan M. Frisch and Timothy J. Peugniez. 2001. Solving non-Boolean satisfiability problems with stochastic local search. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence - Volume 1* (Seattle, WA, USA) *(IJCAI'01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 282–288. doi:10.1007/978-1-4020-5571-3_8

Matt Gallagher. 2016. Exponential time complexity in the Swift type checker. https://www.cocoawithlove.com/blog/2016/07/12/type-checker-issues.html.

Adam Gundry. 2017. Overloaded Record Fields. https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0023-overloaded-record-fields.rst. Haskell Proposal 23. Implemented in GHC 8.2. Accepted on 2017-02-04.

Daniel Hooper. 2024. Why Swift's Type Checker Is So Slow. https://danielchasehooper.com/posts/why-swift-is-slow/.

Jules Jacobs. 2016. Types Mailing List: Congruence rules vs frames. https://lists.seas.upenn.edu/pipermail/types-list/2021/002383.html.

Mark P. Jones. 1992. A theory of qualified types. In *Proceedings of the European Symposium on Programming* (Rennes, France) *(Lecture Notes in Computer Science, Vol. 582)*. Springer-Verlag, 287–306. doi:10.1016/0167-6423(94)00005-0

Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. 2010. TypeChef: toward type checking #ifdef variability in C. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development* (Eindhoven, The Netherlands) *(FOSD '10)*. Association for Computing Machinery, New York, NY, USA, 25–32. doi:10.1145/1868688.1868693

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types, In Proceedings of the Workshop on Mathematically Structured Functional Programming. *Electronic Proceedings in Theoretical Computer Science*. doi:10.4204/eptcs.153.8

Eric Lippert. 2007. Lambda Expressions vs. Anonymous Methods, Part Five. https://learn.microsoft.com/en-us/archive/blogs/ericlippert/lambda-expressions-vs-anonymous-methods-part-five.

Simon Marlow. 2010. Haskell 2010 Language Report. https://www.haskell.org/onlinereport/haskell2010/.

Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17 (1978), 248–375. doi:10.1016/0022-0000(78)90014-4

Neil Mitchell and Shayne Fletcher. 2020. Record Dot Syntax. https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0282-record-dot-syntax.rst. Haskell Proposal 282. Implemented in GHC 9.2. Accepted on 2020-05-03.

Bob Nystrom. 2024. Comment on "How does Scala have function overloading?". Reddit comment in *r/ProgrammingLanguages*. https://www.reddit.com/r/ProgrammingLanguages/comments/1be7wdl/comment/kusavxy/ Accessed 2025-03-26.

Martin Odersky, Philip Wadler, and Martin Wehr. 1995. A second look at overloading. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture* (La Jolla, California, USA) *(FPCA '95)*. Association for Computing Machinery, New York, NY, USA, 135–146. doi:10.1145/224164.224195

Lionel Parreaux. 2020. The Simple Essence of Algebraic Subtyping: Principal Type Inference with Subtyping Made Easy (Functional Pearl). *Proc. ACM Program. Lang.* 4, ICFP, Article 124 (Aug. 2020), 28 pages. doi:10.1145/3409006

David Peter. 2024. *hyperfine. A command-line benchmarking tool.* https://github.com/sharkdp/hyperfine [Last access: 29-07-2025].

Baber Rehman, Xuejing Huang, Ningning Xie, and Bruno C. d. S. Oliveira. 2022. Union Types with Disjoint Switches. In *36th European Conference on Object-Oriented Programming (ECOOP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 25:1–25:31. doi:10.4230/LIPIcs.ECOOP.2022.25

Nick Rioux, Xuejing Huang, Bruno C. d. S. Oliveira, and Steve Zdancewic. 2023. A Bowtie for a Beast: Overloading, Eta Expansion, and Extensible Data Types in F. *Proc. ACM Program. Lang.* 7, POPL, Article 18 (Jan. 2023), 29 pages. doi:10.1145/3571211

Jordan Rose. 2021. Swift Regret: Type-based Overloading. https://belkadan.com/blog/2021/08/Swift-Regret-Type-based-Overloading/. Part of the *Swift Regrets* series.

Swift Language Team. 2024. Type Checker Design and Implementation. https://github.com/swiftlang/swift/blob/main/docs/TypeChecker.md.

Joseph Tan and Magnus Madsen. 2025. Overloading the Dot. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction* (Las Vegas, NV, USA) *(CC '25)*. Association for Computing Machinery, New York, NY, USA, 60–69. doi:10.1145/3708493.3712684

Philip Wadler and Stephen Blott. 1989. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *Proceedings of the Symposium on Principles of Programming Languages* (Austin, Texas, USA). ACM, New York, NY, USA, 60–76. doi:10.1145/75277.75283

Received 2025-03-25; accepted 2025-08-12