Internship Report

Hugo Barreiro M1 MPRI
Supervised by Gabriel Scherer

April 2025 — July 2025

Contents

1 Introduction

The main objective of the internship is to extend a random generator of well-typed programs from the Simply Typed Lambda Calculus to ML-style polymorphism, leveraging a constraint-based type inference engine.

Some elements of this report are drawn from the prototype report Scherer [2024], particularly in sections not exclusively focused on the work done during the internship.

1.1 Motivation

One of the most common testing approaches is to provide random inputs to a program and verify that the outputs satisfy certain expected properties – a technique known as *fuzzing*. When fuzzing a metaprogram such as a compiler, the random inputs must themselves be valid programs. Moreover, in order to properly test the compilation phase, these input programs must be more than just well-formed – they must be well-typed. Unfortunately, generating random well-typed programs becomes increasingly difficult as the type system itself grows more complex. This topic has been relatively fairly studied in the scientific literature – see for example Pałka, Claessen, Russo, and Hughes [2011].

A typical strategy to address this challenge is to invert the compiler's type-checker to perform top-down generation of random terms, while pruning branches that would lead to ill-typed programs. Nevertheless, this kind of generator essentially reimplements a significant part of the compiler's type-checker. This duplication is not ideal when trying to scale the approach to a full programming language, since it requires maintaining two sophisticated type-checkers. Therefore, we aim to discard the generator's type-checker and rely only on the compiler's own type-checker for both typing and term generation.

In this report, we present how to use a single constraint-based type inference engine to handle both type-checking and program generation. Our approach builds upon the work introduced by Pottier and Rémy [2005], which we adapt to serve both purposes. However, one key modification is necessary: parameterizing the constraint generators over a search monad.

1.2 Intuition

We do not want to start by generating well-typed programs – doing so would require encoding complex typing rules directly into the generator. Instead, we propose to begin with the generation of untyped programs, which is significantly easier, and to leverage the *type-inference* engine during the generation process to retain only the well-typed ones. Of course, generating all untyped programs first and then type-checking them all afterward would be highly inefficient. That is why we aim to *interleave* generation and type-checking in a more refined way – cutting off ill-typed branches of the search space as early as possible, and focusing computational effort on exploring well-typed regions.

Our approach for interleaving generation and type-checking is to introduce *holes* into both terms and constraints. We can begin generation with a single hole, denoted **?**, and then refine it incrementally by selecting a term constructor that introduces sub-holes. For instance, we may refine `?` into `fun x -> ?`, and eventually arrive at `fun x -> t`, where `t` is a fully-formed term with no remaining holes. When we invoke the constraint generator on an incomplete term such as `fun x -> ?`, it produces a constraint that also contains holes – meaning that some parts of the constraint are not yet fully specified. Fortunately, the known parts of the constraint can still be partially solved, updating the solver's internal state. At this point, we can already know whether the current (partial) constraint is unsatisfiable or not.

Once the current constraint has been partially solved, we continue by refining one of the remaining holes in the term, which in turn produces a more precise constraint (possibly still with holes). We then solve the known parts of this new constraint, and repeat the process. This continues either until we have a complete, well-typed term (i.e., no remaining holes) or we detect an error.

By enumerating all possible term constructors with holes, we obtain an enumerator over the space of well-typed programs. Moreover, by sampling from this enumeration randomly, we can generate random well-typed programs. Because the solver's state is persistent – or at least backtrackable – we can reuse the solver state across different branches of the generation process. This means that shared prefixes in the term tree do not require redundant type-checking, which contributes to the efficiency of the generation procedure.

In practice, it is *not necessary* to explicitly extend the syntax of terms and constraints with holes in order to interleave generation and constraint solving. Instead, this can be handled more abstractly and elegantly by injecting a constructor for an arbitrary functor. As a result, our terms and constraints are no longer purely syntactic trees – some subtrees are described by *impure* computational processes. We will return to this point in more detail later in the report.

2 Related Work

A notable example of work based on the idea of using the same type system for both type checking and program generation is presented in Fetscher, Claessen, Pałka, Hughes, and Findler [2015]. Using Redex, a domain-specific language embedded in Racket, the authors implemented a simple type system and derived a random program generator. The key idea of their method is to transform each user-defined type system rule into a small logic program, which can then be used to enumerate and sample well-typed terms.

However, Redex is only effective with simple type systems, which significantly limits its applicability to our goals. This constraint prevents scaling to ML-style type inference with polymorphism – one of the main goals of this internship. By *scaling*, we mean that our type system should be expressive enough to support advanced features such as ML-style polymorphism, while maintaining efficient performance for both type checking and program generation. It should also be flexible and predictable enough to be appealing for use in production compilers by type-checker implementers.

In general, declarative approaches to defining type systems (including Redex) are not well-suited for expressing the kind of complex type-checkers found in real-world programming languages. There has also been work on implementing type-checkers directly in Prolog and using them for enumeration and generation, but we do not believe this approach is compelling enough for widespread adoption in compilers. In contrast to these methods, our work builds on a library that uses constraint-solving techniques – an approach known to scale well to ML-style polymorphism and already adopted in systems like the Glasgow Haskell Compiler.

3 Previous work

The origin of the whole project started with my supervisor, Gabriel Scherer, who decided to work on this problem and implemented a prototype for Simply-Typed Lambda-Calculus. Then he had the idea to propose it as an MPRI M2 project for the course *Functional Programming and Type Systems*. He also wrote a small report in order to explain how this approach works – we have already mentioned this report Scherer [2024] in the Introduction.

That is where my internship began. First, I read the report to better understand what the internship consisted of and how to carry it out. Then, I completed the project before focusing on the main objective of the internship in order to become familiar with the code. Next, I started working on the internship by building upon the previous prototype.

The initial prototype – which was extended with ML-polymorphism during this internship – succeeded in combining type inference with program generation. It consists of a simplified reimplementation of the Inferno library Pottier, Scherer, and Martinot [2022], augmented with program generation capabilities. This extension is relatively minimal and does not involve deeply invasive changes. This is why, before the beginning of the internship, we believed that extending the system to support ML-style prenex polymorphism was feasible.

However, the initial prototype's performance was below expectations – it took several seconds to generate terms of size 5. This experiment revealed that designing an efficient generator is challenging, and conversely, that it is quite easy to build one with poor performance. Yet with subtle adjustments to the hole-filling strategy, the system was later able to generate terms of size 20 within a few seconds. The generation process appears to grow exponentially with term size and produces terms with a distribution that remains poorly understood. Addressing this issue became another objective of the internship.

4 Background

This section aims to summarize constraint-based inference in an applicative style, highlighting the key ideas introduced in Pottier [2014] and later implemented in Inferno Pottier, Scherer, and Martinot [2022].

4.1 Constraint-based type inference

Most type inference approaches rely on a single traversal of the program to either infer its type (if it is typable) or raise an error. This is not the case with constraint-based type inference, which proceeds in two phases. First, the program is traversed to produce a *constraint*.

Then, this constraint is passed to a constraint *solver*, which determines whether the program is typable, and if so, provides its type; otherwise, it raises an error.

The main advantage of this method is that it neatly separates the specification of the typing rules of the language (encoded in the constraint generator) from the performance concerns of type inference (encoded in the constraint language and the solver).

To better understand what a generated constraint may look like, consider the program `fun x -> (x + 1, x)`. The constraint generator could produce the following constraint (where a is a free variable representing the type of the entire program, if any):

$$\begin{array}{ll}
\exists a_1 a_2. & \\
\quad a = a_1 \to a_2 & \text{fun x -> ...} \\
\quad \wedge\ \exists b_1 b_2. & \\
\qquad a_2 = b_1 * b_2 & \text{(..., ...)} \\
\qquad \wedge\ b_1 = \mathrm{int} \wedge a_1 = \mathrm{int} & \text{(x + 1, ...)} \\
\qquad \wedge\ b_2 = a_1 & \text{(..., x)}
\end{array}$$

where the left column is the constraint generated by the solver and the right column represents the program fragment that generated the corresponding part of the constraint.

In this example, the constraint is expressed in the following constraint language:

$$\begin{array}{ll}
C ::= & \\
\mid & \mathrm{True} \\
\mid & \mathrm{False} \\
\mid & C_1 \wedge C_2 \\
\mid & \exists a. C \\
\mid & T_1 = T_2
\end{array}$$

where T denotes ”inference types”, i.e. types that may contain inference variable.

There exist two types of constraints – the closed ones and the open ones. Closed constraints contains only bound type inference variables. Open constraints contain free type inference variables.

Closed constraints are the simpler ones to solve because it results in a solution or not, i.e. the constraint is satisfiable or not. We can consider that the constraint is evaluated to a boolean corresponding to the satisfaisability. For the open constraints, it is a little bit different. Its meaning is given by a two-place relation $\gamma \vDash C$, stating that the evaluation γ satisfies the constraint C, where a valuation is a mapping from inference variables to ground types τ (which do not contain inference variables). For the constraint C in our example above, we have $\gamma \vDash C$ if and only if $\gamma(a)$ is a valid type for this program – if it is `int -> (int * int)`.

We decided to express the constraint generation by a function $\mathrm{gen}(E, t, a)$ which constrains the inference variable a to be the type of the source term t, in an environment E mapping free term variables to inference variables. For example, the type-inference rule for a λ-abstraction may be expressed as follows:

$$\mathrm{gen}(E, \lambda x.t, a) := \left(\begin{array}{l} \exists a_x a_t. \\ \quad a = a_x \to a_t \\ \quad \wedge \, \mathrm{gen}(E[x \mapsto a_x], t, a_t) \end{array} \right)$$

We can extend the constraint language to express more advanced type-system features. In our case, for the ML-style polymorphism, we will add let-constraints to capture the infer-and-generalize of ML type inference. We will discuss it later in the report.

4.2 Constraints with elaboration

One interesting idea presented in Pottier [2014] is to extend the constraint language C with a formal `map` operation. This enhancement turns C into an applicative functor, with the following grammar:

$$\begin{array}{rl} C ::= & \ldots \\ | & \mathrm{map}(C, f) \\ | & \mathrm{witness}(a) \end{array}$$

Here, f must be a function defined in some ambient meta-language.

The main idea behind this approach is to evaluate constraints not to booleans (i.e., whether they are satisfiable or not), but directly to arbitrary *values*. For instance, the constraint $C_1 \wedge C_2$ could evaluate to the pair (V_1, V_2), where C_1 evaluates to V_1 and C_2 to V_2.

Specifically, the constraint $\mathrm{map}(C, f)$ evaluates C to a value V and then applies f to yield the final result $f(V)$. The constraint $\mathrm{witness}(a)$ returns the type τ inferred for the inference variable a in the final solution to the constraint – assuming that ground types τ can be expressed in our meta-language. For instance, if the meta-language is typed, then it makes sense to classify constraints by the type of their resulting values.

With this extension, the semantics of constraints are now expressed by a ternary judgment $\gamma \vDash C \rightsquigarrow V$, which reads: under the valuation γ, the constraint C evaluates to the value V.

Using this enriched constraint language, we can capture elaboration. From a term in the source language, we can generate constraints that not only verify its type but also evaluate to a version of the term annotated with explicit typing information. Therefore, type checker designers can simultaneously implement type-checking rules for their language and construct explicitly typed representations of well-typed programs.

The following is an example of how this constraint grammar can be represented in OCaml, using a GADT (Generalized Algebraic Data Type). Hence, OCaml serves as our meta-language:

```
module Constraint : sig
  ...
  type _ t =
  | True : unit t
  | False : 'a t
  | Conj : 'a t * 'b t -> ('a * 'b) t
  | Exist : variable * 'a t -> 'a t
  | Eq : inf_type * inf_type -> unit t
  | Witness : variable -> ground_type t
  | Map : 'a t * ('a -> 'b) -> 'b t
end
```

Next, we illustrate how the constraint generator might be implemented. Recall the constraint-generation rule for λ-abstractions:

$$\mathrm{gen}(E, \lambda x.t, a) := \left(\begin{array}{l} \exists a_x, a_t. \\ \qquad a = a_x \to a_t \\ \quad \wedge\, \mathrm{gen}(E[x \mapsto a_x], t, a_t) \end{array} \right)$$

Below is the OCaml implementation of type inference for λ-abstractions. This corresponds to the constraint generation rule above, enriched with elaboration information:

```
let rec has_type (env : infer_env) (t : Untyped.t) (a : variable) : Typed.t =
  match t with
  | ...
  | Untyped.Abs (x, t) ->
    let arg = Constraint.Var.fresh (Untyped.Var.name x) in
    let res = Constraint.Var.fresh "res" in
    Exist (arg, Exist (res,
      let+ () = Eq(a, (Struct(Arrow(arg, res))))
      and+ arg_ty = Witness arg
      and+ t' = has_type (Env.add x arg env) t res
      in Typed.Abs (x, arg_ty, t')
    ))
```

Here, the expression `let+ p1 = e1 and+ p2 = e2 in body` uses OCaml's applicative functor syntax (a variant of the applicative do notation). It desugars into:

```
map (fun (p1, p2) -> body) (pair e1 e2)
```

with the following applicative operations:

```
val map : ('a -> 'b) -> 'a t -> 'b t

val pair : 'a t -> 'b t -> ('a * 'b) t
```

4.3 Solving Constraints

There are two main ways to formalize inference constraint solvers. Many research papers describe them using small-step semantics, which consist of numerous independent rewriting rules that preserve the set of solutions while progressing toward a normal form. In contrast, many software implementations prefer big-step semantics, where constraints are evaluated in a single recursive traversal.

In our approach, we update the solver's state by injecting information into the constraints as solving progresses. To efficiently implement ML-style type inference in practice, the classical method relies on two well-chosen data structures:

- A Union-Find graph is used to manage unification. The nodes of the graph represent types and inference variables. When solving the equality constraint $T_1 = T_2$, we unify the two nodes corresponding to T_1 and T_2. This approach was implemented in the initial prototype.

- An essential concept for efficiently implementing the *generalization* of let-constraints is the use of *inference levels*, as described, for example, in Oleg Kiselyov's article Kiselyov. See the next section for more details about this crucial mechanism for supporting prenex polymorphism. This mechanism was not present in the initial prototype, which handled only the Simply-Typed Lambda Calculus.

The central algorithmic feature of the initial prototype is unification. This requires an API with two functions that manipulate the solver's state:

```
type state
val unify : state -> inf_type -> inf_type -> (state, error) result
val equal : state -> inf_type -> inf_type -> bool
```

The `unify` function attempts to unify two types, updating the solver's state if the unification is successful, or returning an error otherwise. The `equal` function simply checks whether two types are already equal without attempting to unify them.

We can now define our `eval` function to solve constraints. The type of the function is slightly complex because of generation issues:

```
module Constraint : sig
  ...

  type 'a residual =
  | Ret of (state -> 'a)
  | Fail of error

  val eval : state * 'a t -> state * 'a residual
end
```

The `eval` function uses the solver's state to solve the constraint, updating the state and returning a *residual constraint*, which represents any parts of the constraint that have not yet been resolved. At this stage, our constraint constructs do not allow partial solutions, so the *residual* acts more like a value-or-error result. We will discuss later how to enable the solver to partially resolve constraints to support generation. If resolution is successful, the function returns a thunk of type `state -> 'a` that produces a witness of type `'a` from the final solver state; otherwise, it returns an error.

A notable feature (and potential disadvantage) of this approach is that the `Ret` constructor carries a function of type `state -> 'a`, which is parameterized by the final solver state – the "solution" of the constraint. This implies that all constraints must be fully resolved before we can evaluate constraints like `Witness a`. In other words, we cannot determine the type of an inference variable `a` until all unification constraints have been solved. For example, consider solving `Conj(c, d)`:

```
let rec eval (st, c0) =
  match c0 with
  | ...
  | Conj (c, d) ->
    let st, nc = eval (st, c) in
    match nc with
    | Fail e -> st, Fail e
    | Ret v ->
      let st, nd = eval (st, d) in
      match nd with
      | Fail e -> st, Fail e
      | Ret w ->
        st, Ret (fun sol -> (v sol, w sol))
```

Here, `c` and `d` are first evaluated into residual constraints. If either fails, we return the corresponding error. Otherwise, if both succeed, we construct a pairing thunk `state -> ('a * 'b)` using the individual thunks `state -> 'a` and `state -> 'b`.

We can now summarize the constraint-solving process. Starting from an `Untyped.t` source term, we generate a constraint using the following function:

```
val infer : Untyped.t -> Typed.t Constraint.t
```

This function also produces a witness of type `Typed.t`, i.e., a type-annotated term. We then solve the constraint using our `eval` function and an initial solver state:

```
val eval : state * 'a Constraint.t -> state * 'a Constraint.residual
```

This yields the final state along with either an error or a `state -> Typed.t` value, from which we can obtain our typed term. Thus, we can define the type-checking process as follows:

```
let typecheck (source_term : Untyped.t) : (Typed.t, error) result =
  let constr : Typed.t Constraint.t = infer source_term in
  let (sol, nc) = eval (initial_state, constr) in
  match nc with
  | Ret thunk -> Ok (thunk sol)
  | Fail error -> Error error
```

4.4 Let-constraints

In the previous sections, we explained how to manage constraint-based type inference, particularly in the case of the Simply-Typed Lambda Calculus. Our objective is to handle ML-style polymorphism, which requires introducing an additional algorithm alongside unification: *generalization*. This section presents this new mechanism.

4.4.1 What are Let-constraints?

The fundamental difference between the Simply-Typed Lambda Calculus (STLC) and ML-style type inference is that let-bound values are polymorphic in ML. A simple example to illustrate this:

```
let id x = x in
(id false, id 0)
```

In ML (or System F), this program evaluates to the value `(false, 0)`, whereas in the Simply-Typed Lambda Calculus, it would result in a type error.

This is because, in STLC, the type of `id` is `'a -> 'a`. When type inference processes `id false`, the type `'a` is inferred as **`bool`**. Then, when it processes `id 0`, the type **`int`** conflicts with the previously inferred **`bool`** type, leading to a type error. In contrast, in ML, the type of `id` is $\forall \alpha . \alpha \to \alpha$. This means that when type inference processes `id false`, `id` is instantiated with the type `boolean`, which can be seen as creating a specialized function `id_boolean`. Similarly, when processing `id 0`, `id` is instantiated with **`int`**, as if it were resulted in `id_int`. Thus, type inference succeeds.

How does this polymorphism work in ML? All values defined with **`let`** are *generalized*, meaning that all bound variables introduced by the **`let`** are not treated as ordinary inference variables. Instead, they are added to the scheme of the **`let`**. In the example above, the value `id` has the scheme $\forall \alpha . \alpha \to \alpha$ because the inference variable α was introduced by the lambda inside the **`let`**.

A scheme consists of a root, generics, and quantifiers. The root of the scheme is a variable that serves as the entry point of the scheme. Generics include all type variables that appear in the body of the scheme – that is, the right-hand side of the scheme, excluding its quantifiers. Quantifiers are a subset of the generics; they are the type variables bound in the scheme and are introduced by a *forall*. The key distinction between generics and quantifiers in a given scheme is that quantifiers are the forall-bound type variables of the scheme, whereas generics may also include type variables introduced during constraint generation.

For example, consider the function `fun f x = (x, x)`. Its scheme is $S(w) = \forall \alpha . \alpha \to (\alpha * \alpha)$, and the corresponding constraint is $S := \Lambda w . \exists \alpha \beta . \beta = (\alpha * \alpha) \wedge w = \alpha \to \beta$. Therefore, the root of this scheme is w, the generics are $w = \alpha \to \beta$, $\beta = \alpha * \alpha$, and α, and the quantifiers consist solely of α.

This brings us to the concept of instantiation. Informally, instantiation can be seen as a kind of beta-reduction. Referring to the previous example, $f \leq w' \rightsquigarrow S[w := w']$ where w' is a type variable. Instantiation involves copying the entire initial scheme – including its root and generics – while replacing all type variables in the scheme with fresh ones. This process can be understood as the creation of a new scheme that is equivalent to the original, but with a specialized type instead of a universal one. By substituting the quantifiers with the desired type variables, we instantiate the scheme to obtain the specific type we need.

For instance, continuing the previous example, if we write `f 42`, we copy the entire scheme of `f` using fresh type variables. In particular, the variable α is replaced with `int`, resulting in the type `int` $\rightarrow$ (`int` $*$ `int`) for the funcion `f` in the application. This is achieved by instantiating its original scheme $\forall\alpha.\,\alpha \rightarrow (\alpha * \alpha)$.

Therefore, when solving constraints, we must compute the scheme for each let-definition. To do this, we identify which bound inference variables were introduced by the current let-definition. This is equivalent to bringing the *forall* quantifiers to the top of each let-definition.

Although this might seem inefficient, there is a technique for efficiently computing schemes: the *inference levels* method.

4.4.2 Inference levels method

When generalizing a **`let`**, we must identify all the inference variables that it depends on in order to include them in its scheme. This would usually require traversing the entire constraint to determine whether a variable depends on the **`let`** being generalized, which is inefficient. The *inference levels* technique provides a more efficient approach.

The idea is to assign an inference level corresponding to the nesting depth of **`let`** bindings in the program. The top-level of the program is level 0; entering a **`let`** increases the level to 1, and so on. To track the current level, we maintain a level counter in the solver's state. Initially, this counter is set to -1 (assuming the program is always wrapped in a **`let`**), and it is incremented upon entering a **`let`** and decremented when exiting.

When entering a **`let`**, after increasing the inference level, we evaluate the definition. Once evaluated, we compute the scheme associated with the definition and then decrease the inference level before evaluating the body.

To perform generalization using this method, we augment each inference variable with its own inference level and a status: *flexible* or *generic*. A generic variable has already been generalized and is part of a let scheme. A flexible variable has not yet been generalized. The generalization process is as follows:

1. **Compute the young level:** Retrieve the current solver level (young level).

2. **Discover the young generation:** Collect all variables that remain generalizable at the young level. All these variables are the *young generation*.

3. **Update the levels:** For each variable in the young generation, and for levels from 0 to the young level, update their level to the minimum of the current and young levels. Recursively apply this to each variable's children. A variable's children are the variables it depends on; e.g., if a represents $a_1 \rightarrow a_2$, then a_1 and a_2 are its children.

4. **Generalization phase:** For each variable in the young generation, if its level is equal to the young level, mark it as generic by changing their status. All these variables are

the generic variables of the scheme.

5. **Build the scheme:** Starting from the scheme root, traverse it and its children to collect all reachable generic variables. Return the scheme : the root, the reachable generics, and the quantifiers (for instantiation information).

6. **Finish the generalization:** Decrease the inference level and return the computed scheme.

4.4.3 In practice

To support generalization, we extend our constraint syntax as follows:

$$\begin{aligned} C ::= &\ \ldots \\ \mid &\ \text{witness\_scheme}(s) \\ \mid &\ \text{instance}(s, a) \\ \mid &\ \text{let}(s, a, C_1, C_2) \end{aligned}$$

Here, s is a scheme variable. These new constraints are defined as:

- `witness_scheme`(s): Similar to `witness`(a), but for scheme variables. Returns the list of quantifiers and the inferred type τ for the scheme associated with s.

- `instance`(s, a): Attempts to instantiate the scheme S associated with s using inference variable a. If successful, unifies a with a copy of S's root and updates the solver state; otherwise, returns an error.

- `let`(s, a, C_1, C_2): Infers the scheme of inference variable a based on constraint C_1, generalizes it, associates it with scheme variable s, then solves constraint C_2.

Additionally, we extend our API with functions for manipulating the solver's state:

```
val enter : state -> state

val exit : state -> variable -> state * scheme

val instantiate : state -> scheme -> state * variable * variable list
```

Here, `enter` increments the inference level; `exit` computes the scheme using the provided variable as root and returns the updated state and scheme; `instantiate` returns a copy of the scheme's root and its quantifiers with the updated state.

The solver can now use these functions to perform the *generalization* process as described for the `witness_scheme`, `instance`, and `let` constraints.

5 Previous prototype

We showed earlier how to perform constraint-based inference in an applicative style for the purpose of type-checking programs. Now, we will explain how the previous prototype handled term generation.

5.1 Terms and constraints with subcomputations

We previously mentioned that our solver cannot halt computation during the type-checking process. As a result, the residual type of constraints is merely a success-or-failure value. In this section, we address and resolve this issue.

First, we define what we refer to as a *functor*: a *functor* is a module `F` that implements the following signature:

```
module type Functor : sig
  type 'a t
  val map : ('a -> 'b) -> 'a t -> 'b t
end
```

This refers to the concept of a functor from category theory, not to be confused with ML module system functors, which are entirely different concepts.

Let us recall the structure of our old module for untyped terms:

```
module Untyped : sig
  ...

  type t =
  | Var of Var.t
  | App of t * t
  | ...
end
```

We now need to transform this module to support generation. To achieve this, we parameterize it over an arbitrary functor `F`, derived from an ML module functor over category-theory functors, and we extend the term syntax with a new constructor `Do`, as shown below:

```
module Untyped (F : Functor) : sig
  ...

  type t =
  | Var of Var.t
  | App of t * t
  | ...
  | Do of t F.t
end
```

Similarly, we can parameterize our constraints module in the same way, adding an analogous `Do` constructor:

```
module Constraint (F : Functor) : sig
  ...
  type 'a t =
  | ...
  | Do of 'a t F.t
end
```

The intuition behind these changes is that terms and constraints now represent syntactic trees in which some subtrees correspond to computations expressed within an external computational context `'a F.t`. This means that if `F` is a non-determinism monad (for instance, if `F` is `List`), then the `Do` constructors allow us to describe untyped terms or constraints in which a subtree can be a non-deterministic enumeration of all possible subtrees, potentially including nested computations.

We can also extend constraint generation to handle the new `Do` constructor as follows:

```
let rec infer env a : Typed.t Constraint(F).t = function
| ...
| Do (m : Untyped(F).t F.t) ->
  Do (F.map (infer env) m : Typed.t Constraint(F).t F.t)
```

5.2 Constraint solving with subcomputations

As mentioned earlier, we can extend our residual type to obtain a more informative form of unresolved constraints, rather than simply a success-or-failure result. To achieve this, we introduce a `Do` constructor, which encapsulates a subcomputation within the functor `F` that returns a constraint. This can be defined as follows:

```
module Constraint (F : functor) : sig
  ...

  type 'a residual =
  | Ret of (state -> 'a)
  | Fail of err
  | Do of 'a t F.t
end
```

With this change, the solver can use the `Do` constructor in the residual for each input constraint that contains the `Do` constructor. When the solver recursively encounters a residual with a `Do` constructor, it can generate a new residual constraint by propagating new operations using the functor map. To illustrate this, consider the case of conjunction:

```
| Conj (c, d) ->
  let st, nc = eval (st, c) in
  match nc with
  | Fail e -> st, Fail e
  | Do m -> st, Do (F.map (fun c -> Conj (c, d)) m)
  | Ret v ->
    let st, nd = eval (st, d) in
    match nd with
    | Fail e -> st, Fail e
    | Do m -> st, Do (F.map (fun d -> Conj (pure v, d)) m)
    | Ret w -> st, Ret (fun sol -> (v sol, w sol))
```

5.3 Search monad

Let us introduce the concept of a *search monad*. A search monad is a module `M` with the following signature:

```
module type SearchMonad : sig
  type 'a t

  val return : 'a -> 'a t
  val bind : ('a -> 'b t) -> 'a t -> 'b t
  val sum : 'a t list -> 'a t
  val run : 'a t -> 'a Seq.t
end
```

where `'a Seq.t` is the OCaml type for a sequence of elements of type 'a, computed on demand – i.e., in a lazy manner.

This kind of monad allows us to describe, using a value of type `'a M.t`, a search problem with zero, one, or more solutions of type `'a`. For example, the prototype implements two monads: an exhaustive one and a random one. These two monads allow us, via the search-monadic value, either to enumerate all described terms or to sample a certain quantity of them.

5.4 Iterated search

We can modify the type of the solver's `eval` function so that it takes as an argument a `'a Constraint(M).t` and returns a `'a Constraint(M).t residual` which contains subcomputations of type `'a Constraint(M).t M.t`, for any search monad `M`.

This allows the generation of terms by iterating the solving process on the results of these subcomputations. It leads to a new function, `solve`, which transforms a `'a Constraint(M).t` into a `'a M.t`. This function also takes as arguments the solver's state and an integer `~size`, representing the remaining number of solving iterations we are allowed to perform:

```
  let rec solve ~size state (cstr : 'a Constraint(M).t) : 'a M.t =
  if size < 0 then M.sum []
  else
    let state, nf = eval state cstr in
    match nf with
    | Fail _ -> M.sum []
    | Do p -> M.bind (solve ~size:(size - 1) state) p
    | Ret v -> if size > 0 then M.sum [] else M.return (v state)
```

Here, returning the value `M.sum []` indicates a failure – meaning that we cannot provide a subterm, produced by solving a sub-constraint, to construct a term of size exactly `~size`.

5.5 Generic untyped and typed terms

To generate as many terms as we want, we still need to implement a ”generic” `Untyped(M).t` term – for any search monad `M` – which describes the complete set of possible untyped terms by listing, under a `Do` constructor, all possible term formers:

```
let untyped_terms : Untyped(M).t =
  let rec gen (ctx : Untyped(M).Var.t list) : Untyped(M).t =
    Do (
      let rule_var =
        let+ x = of_list ctx
        in Var x
      in

      let rule_app =
        let t = gen ctx in
        let u = gen ctx in
        M.return (App(t, u))
      in

      let rule_lam =
        let x = Untyped(M).Var.fresh ctx in
        let t = gen (x :: ctx) in
        M.return (Lam(x, t))
      in

      M.sum [ rule_var; rule_app; rule_lam ]
    )
  in
  gen []
```

Thus, if we call the function `solve ~size` on this generic untyped term, we obtain a generic typed term whose resulting solutions are all well-typed terms of size exactly `~size`:

```
let well_typed_terms ~size : Typed.t M.t =
  let cstr : Constraint(M).t = infer untyped_terms in
  solve ~size empty_state cstr
```

To complete the term generation, we simply need to instantiate a search monad `M` that suits our goals – whether it be exhaustive enumeration, random sampling, or any other strategy.

6 Internship

This section presents the new work carried out during the internship and its main outcomes. The code repositories are available on GitLab and GitHub.

6.1 Performance

6.1.1 Context

The previous prototype implemented a random search monad to sample random terms from the generic typed term described earlier. Generating a term of size 10 was instantaneous, but for a term of size 20, it became too slow – averaging about 15 seconds. This slowdown appears to be due to the exponential growth in time complexity relative to the size of the term being generated. Part of the internship focused on resolving this performance issue.

On the other hand, sampling multiple terms appears to scale sub-linearly: generating one term of size 16 took about 2 seconds, while generating 10 terms of the same size took only around 8 seconds. Preserving this desirable property is a goal when adding ML-style polymorphism.

As mentioned before, re-implementing the prototype was assigned as a project for the M2 MPRI course *Functional Programming and Type Systems*. As a result, a student named Neven Villani developed a random search monad capable of generating a term of size 200 almost instantaneously. However, we dit not understand why this monad exhibited such a significant performance improvement compared to the original prototype.

This section presents our findings on how to implement an efficient random generator, focusing particularly on random search monads and term generators.

6.1.2 Random Monads

In this section, we present different monads to improve the performance of the term generator. OCaml implementations of these monads are available in the Appendix.

Prototype-Naive Monad vs Neven Monad

Our initial goal was to understand why the Neven random-search monad was significantly more efficient than our Prototype-Naive version.

First, the naive monad simply picks random constructors when available, with a small optimization that removes the `Fail` constructor from the `Sum` – in practice, we use a `Fail` constructor instead of `Sum []`.

The Neven monad is more sophisticated. This implementation introduces the concepts of *computed* terms and *to-be-computed* terms. That is, normal-form terms – introduced, for example, by `one_of` constructors – are separated from the rest of the monadic terms that still require computation to reach normal form. This approach allows us, first of all, to avoid reusing terms that have already been tried. We simply remove them from the list of *computed* terms when they are selected. Furthermore, it enables us to discard *empty* paths, meaning monadic generator terms that no longer contain any selectable terms to be tried.

These two actions significantly reduce the number of failing paths. First, we avoid selecting a term that has already been used to fill the same `Do` node. Second, by removing *dead ends* from the search tree, we recursively reduce enough paths to save many iterations.

In summary, the naive implementation tends to propose the same choices multiple times and explore dead-end paths when attempting to fill the same `Do` node.

Full-Removal Monad

The main objective of this new monad is to confirm our intuition about why the Neven monad was so efficient. To achieve that, we aim to reimplement the idea of not following the same path through the generator twice – that is, not trying the same term twice to fill the same `Do` node – while also removing dead ends. The twist, compared to the Neven version, is to implement it more simply with code that is fully understandable.

To do this, we revisit the naive model in which each monadic function is associated with a constructor of an OCaml GADT, along with a `next` function that attempts to produce a new term from a given generator (while also updating the generator in the process):

```
type 'a t =
  | Return : 'a -> 'a t
  | Fail : 'a t
  | Delay : 'a t Lazy.t -> 'a t
  | Map : 'a t * ('a -> 'b) -> 'b t
  | Bind : 'a t * ('a -> 'b t) -> 'b t
  | Sum : 'a t list -> 'a t
  | One_of : 'a array -> 'a t

val next : 'a t -> 'a option * 'a t
```

It is relatively straightforward to ensure these properties for all constructors except `Bind`. In the naive monad, when we encounter a `Bind(ta, f)`, we first generate a value `a` from `ta`, then compute `f a`, which gives us a new generator `tb`, from which we finally generate a value `b`.

With this approach, we cannot remove the `a` from `ta`, because we only generate a single `b` and not all possible values from `tb`, which leads to a loss of information. Now, we want to save `a` as a result from `ta`, and then remove the `b` from its corresponding `tb`.

To do this, we change the type of `Bind` to:

```
| Bind : 'b t option * 'a t * ('a -> 'b t) -> 'b t
```

We can observe a slight change in logic: here, we remove `a` from `ta` and store the resulting `tb` on the side of the `Bind` as an additional argument. The plan is to draw from `ta` when `tb` is empty – which is the case initially, before any values have been drawn, for example. Once `tb` is not empty, we systematically draw from it.

There is a potential issue with this method: it can be inefficient in the presence of *deep* dead ends. If the `a` we initially drew is particularly bad, we may have to perform many attempts to find a value or exhaust the generator before trying another value from `ta`, which might perform significantly better – but we remain stuck in the same `tb`.

And… the results are disappointing! The Full-Removal monad performs worse than both the Prototype-Naive and Neven monads. To bounce back, we implemented a new monad that we believed could potentially solve this performance issue.

Local-Retries Monad

This new monad is a variation of the Full-Removal monad that attempts to find a term by drawing a new value locally instead of retrying from the beginning. So, when we draw from a sub-value of type `'a t` and it results in failure, we attempt to redraw from this value instead of propagating the failure to the top level and failing there – as was the case in the Full-Removal monad.

The advantage of these *local retries* is that they can significantly reduce the number of retries needed to generate the next new term. However, this approach does not truly solve the problem of deep dead ends: if we initially choose a poor generator, it may be worse to retry locally than restarting from the top.

It is important to note that this strategy must build on top of the Full-Removal monad because the Full-Removal strategy ensures that empty generators will become `Fail` – after finitely many retries. Without this, the Local-Retries strategy could result in an infinite loop within an empty term that is not (and will never be reduced to) a `Fail`.

This approach can potentially improve upon the previous monad but does not truly solve the issue of deep dead ends that undermine performance – which still lags behind the two initial monads. Fortunately, we found a solution to counter this phenomenon.

Reset Monad

The advantage of our new idea is that it is fairly modular – we can use it for both the Full-Removal and Local-Retries monads. However, there is a disadvantage: it is not particularly elegant – it feels more like a hack than a clever technique.

The idea is to maintain a counter of the number of retries in order to ”reset” the search if it exceeds a certain threshold. When we decide to reset, we restart drawing from the initial – i.e., the top-level – generator.

This approach helps avoid getting stuck for too long in deep dead ends of the search space, for example when we are retrying locally after failures. Another benefit is that it prevents being stuck with the cached `tb` value given by the `Bind(tb, ta, f)` constructor if it turns out to be a dead end itself.

On the downside, the (major) drawback is that this method relies on a constant – the limit of retries before resetting the search from the top-level generator – which is somewhat arbitrary. If this constant is too small, the generator becomes too unstable to generate very large terms. Conversely, setting the constant too high makes the reset strategy ineffective in practice, as it would take too long to reset when stuck in deep dead ends.

Our experiments suggest that 1000 retries are sufficiently effective in practice. However, this conclusion is based solely on heuristics, not on any formal proof that this value is optimal. That said, this kind of approach is not novel. There is a connection between our notion of reset and the *restart* mechanism in Conflict-Driven Clause Learning-based SAT/SMT solvers. In such solvers, when a conflict is encountered, a new clause is added to prevent repeating the same type of conflict. Moreover, they choose to restart from the beginning – with the newly learned clauses – after a certain number of conflicts. The number of conflicts before restarting can follow a specific sequence, such as: 1, 1, 2, 1, 2, 4, 1, 2, 4, 8, ...

Unfortunately, despite being our best monad among the new ones, it still performs worse than the two original ones. At this point, we hadn't found any new ideas that could enable performance similar to the Neven monad – or even to the naive one.

6.1.3 Generators

Currently, our generator works by passing the untyped generic term to the `solve` function to produce a typed term. The `solve ~size` function will explain exactly `size` times a `Do` node, and create a term that may still contain holes. And then it fails if at least one hole remains – if evaluation returns a residual `Do`. This is inefficient because most terms of a given size have at least one hole, so most runs are rejected in the end.

To mitigate this, we can implement a `cut_size` function that prunes the exploration tree – derived from our untyped generic term – to retain only the branches that lead to a term of exactly size `~size`:

```
val cut_size : size:int -> Untyped(M).term -> Untyped(M).term
```

We provide only the signature of `cut_size` here, as the full implementation is too verbose. An OCaml example is available in the Appendix.

With the `cut_size` function ensuring that only terms of exact `~size` are generated, we can now modify our `well_typed_terms` function to produce well-typed terms of the correct size. Consequently, our `solve` function can be simplified to ignore size considerations, as shown below:

```
let rec solve state (cstr : 'a Constraint(M).t) : 'a M.t =
  let state, nf = eval state cstr in
  match nf with
  | Fail _ -> M.sum []
  | Do p -> M.bind (solve state) p
  | Ret v -> M.return (v state)
```

```
let well_typed_terms ~size : Typed.t M.t =
  untyped_terms
  |> cut_size ~size
  |> infer
  |> solve empty_state
```

We can now compare the overall performance – including that of the search monads – between our previous generator and this new *cut-early* approach: the latter performs worse for both the Prototype-Naive and the Neven monads. Surprisingly, it performs quite well with the newer monads!

With the cut-early strategy, we can generate a term of size 300 instantaneously (using Local-Retries with Reset strategy), a term of size 600 in under 1 second on average (again using Local-Retries with Reset strategy), and a term of size 1000 in under 5 seconds on average (this time using Full-Removal with Reset strategy).

These results pertain to the Simply-Typed Lambda Calculus with `let` and pairs. The phrase *on average* reflects the inherent randomness of generation: sometimes a term is found unusually quickly, while at other times we are "unlucky," and it takes significantly longer.

A remark on monad efficiency depending on the generator they operate with: we observed that both the Prototype-Naive and Neven monads handle `Sum` reasonably well but perform poorly on `Bind`. In contrast, Full-Removal-based monads are designed to handle `Bind` efficiently but have more difficulty with `Sum`.

Our intuition about why Full-Removal-based monads perform better with the cut-early strategy – while others do not – is linked to the fact that the search space is more precise, in the sense that fewer branches lead to dead ends. At the same time, however, it is more complex due to the introduction of approximately twice as many `Sum` nodes. One hypothesis is that Full-Removal-based monads handle this kind of search space effectively, whereas the Neven monad's lack of performance with the cut-early strategy might result from the fact that it was likely designed to work well with the initial search space, without the cut-early approach. The exact reasons behind the performance differences with the cut-early strategy remain not fully understood.

Remark: The exhaustive search monad has become significantly more efficient with the cut-early strategy. For instance, exhaustively generating the 67,009 terms of size 9 took around 6 minutes and 30 seconds using the initial generator strategy, whereas with the cut-early approach, it took only about 8 seconds. The cut-early approach appears to provide an exponential time saving compared to the initial generator strategy.

6.2 The impact of generalization

During the internship, we extended the prototype to support ML-style polymorphism. In addition to implementing the *generalization* process, we also had to modify several components and approaches within the prototype. We present these changes in this section.

6.2.1 Solver State

Since the beginning of this report, we have been referring to the solver's *state* without clearly defining what it actually is. In the prototype, it was simply the unification state. Concretely, this corresponds to an OCaml `Map` between inference variables and their internal representation in the unification store – a data structure that allows us to unify inference variables efficiently.

As explained in the Background section about let-constraints, polymorphism requires us to now handle both *generalization* and scheme variables. Scheme variables are internal to the solver, whereas the generalization mechanism introduces its own state – similar to unification.

Therefore, the solver must now manage two external states: both generalization and unification states. We start with two empty states and update them whenever we call API functions that affect these states. Alongside these two external states, the solver must also manage its own environment in order to map scheme variables to their corresponding schemes.

6.2.2 Solver Continuations

A natural approach for evaluating the `Exist(x, s, c)` constraint is to add the inference variable to the environment, and then evaluate the constraint `c`:

```
| Exist (x, s, c) -> begin
  env := Unif.Env.add x s !env;
  match eval c with
  | Fail e -> Fail e
  | Do p -> Do (let+ c = p in Exist(x, s, c))
  | Ret v -> Ret v
end
```

This might be the implementation in the prototype, without considering generation backtracking. Since we re-enter the entire constraint each time we attempt to fully resolve it to generate a term – and we do this many times – we don't need to reintroduce the variable each time. Doing so risks overwriting existing data in the environment and making the resolution incorrect.

To address this issue, the prototype uses a workaround. It checks whether the variable is already in the environment before introducing it:

```
| Exist (x, s, c) -> begin
  if not (Unif.Env.mem x !env) then env := Unif.Env.add x s !env;
  match eval c with
  | Fail e -> Fail e
  | Do p -> Do (let+ c = p in Exist(x, s, c))
  | Ret v -> Ret v
end
```

The drawback of this method is that it does not scale well to ML-style polymorphism. We encounter similar conflicts with the generalization mechanism: entering a new generalization level, introducing new bound variables, associating scheme variables with their corresponding schemes, and finally exiting the generalization level.

Of course, we could apply a similar hack to handle all these generation-backtracking issues, but it would lack elegance – we also encountered bugs when we tried to do this, which we were not sure how to fix. This is why we decided to introduce internal continuations in the solver to ensure each part of the constraint is processed only once, using OCaml GADTs:

```
type ('a1, 'a) cont =
  | Done : ('a, 'a) cont
  | Next : ('a1, 'a2) cont_frame * ('a2, 'a) cont -> ('a1, 'a) cont

and ('a1, 'a) cont_frame =
  | KMap : ('a1 -> 'a2) -> ('a1, 'a2) cont_frame
  | KConj1 : 'a2 constr -> ('a1, 'a1 * 'a2) cont_frame
  | KConj2 : 'a1 on_sol -> ('a2, 'a1 * 'a2) cont_frame
  | KExist : var -> ('a, 'a) cont_frame
  | KLet1 :
    (scheme_var * var) list * 'a2 constr
    -> ('a1, 'a1 * 'a2) cont_frame
  | KLet2 :
    scheme_var list * 'a1 on_sol
    -> ('a2, 'a1 * 'a2) cont_frame

and 'a on_sol = (var -> ty) -> 'a
```

This may seem difficult to understand, but fortunately it is not too complex. We will explain what this type means.

First, the `cont` type helps us to recursively list the actions to be performed in the correct order. Either it's `Done`, meaning the computation is complete, or we must perform a computation step before proceeding with the remaining tasks.

The `cont_frame` type describes the specific actions to perform during computation:

- `KMap` indicates that we must apply a function `f` to the current constraint in the solver before continuing the computation.
- `KConj1` means the left-hand constraint of a conjunction has been computed, and now the right-hand one must be evaluated. `KConj2` means that, using the results of both sides of the conjunction, we can construct the resulting conjunction solution.
- `KExist` means the new variable has already been introduced and its corresponding constraint has been evaluated; we can now continue the computation. In theory, we could remove the variable from the solver's state, but we cannot do this in practice because our ”solution” system requires the full state to map witnesses for all variables to construct the term.

- `KLet1` indicates we have entered a new inference level, introduced variables, and evaluated the definition constraint. Now we must exit the current inference level and construct the schemes to associate them with their corresponding scheme variables before evaluating the body constraint. `KLet2` means the entire let-constraint has been evaluated. We can now remove the scheme variables from the environment – unlike existential variables – because scheme variables do not participate in building the "solution".

We can observe that, with this type of continuation, all solver steps are, to some extent, formally described.

Additionally, we can now improve how we handle the `Do` case. Previously, in the prototype, we had:

```
type 'a residual =
  | Ret of (state -> 'a)
  | Fail of err
  | Do of 'a constr F.t

let rec eval (st, c0) : state * 'a residual =
  match c0 with
  | ...
  | Do p -> Do p

let rec solve state cstr =
  let state, nf = eval state cstr in
  match nf with
  | Fail _ -> M.sum []
  | Do p -> M.bind (solve state) p
  | Ret v -> M.return (v state)
```

Now, we can do slightly better with:

```
type 'a residual =
  | Ret of state * (state -> 'a)
  | Fail of err
  | Do of 'a residual F.t

let rec eval st c0 k : 'a residual =
  match c0 with
  | ...
  | Do p -> Do (T.map (fun c -> eval st c k) p)

let solve cstr =
  let rec loop = function
    match cstr with
    | Fail _ -> M.sum []
    | Do m -> M.bind m loop
    | Ret (state, v) -> M.return (v state)
  in
  loop (eval empty_state cstr)
```

First, this means that the solver's state is now fully managed internally by the solver. We can observe this change by the fact that when we call the `eval` function with `empty_state`, it never returns the updated state until it has completely solved the constraint – unless it encounters a `Fail` or `Do` node. Additionally, the type of `Do` has been changed to accept a residual constraint instead of a constraint. This means that we can now handle the filling of `Do` nodes internally rather than in the `solve` function.

This change allows us to better separate responsibilities between the `eval` and `solve` functions. Now, `eval` is responsible for all aspects of constraint resolution, such as managing its own state and handling `Do` nodes. Meanwhile, the `solve` function simply handles loop control: deciding what to do when the result is a `Fail`, a `Do`, or a `Ret`.

Moreover, in the solver, we no longer need to map over the constraint for each constructor, as we did in the `Exist` implementation above. Thanks to continuations, we can directly resume evaluation using the final continuation when we encounter a `Do` node. We can simplify the `Exist` implementation as follows:

```
| Exist (x, s, c) ->
  let env = Env.add x s env in
  eval env c (Next (KExist x, k))
```

In summary, since we now resume from the final continuation instead of restarting from the top of the constraint, we no longer need hacks to manage backtracking. Each part of the constraint is processed exactly once. Furthermore, this approach simplifies other cases as well, because we stop evaluation when encountering a `Do` and resume from that point. As a result, we no longer encounter residual-`Do` constraints when evaluating other cases.

Finally, continuations also allow us to significantly improve how we print constraints. In the prototype, we attempted to simplify constraints to understand our position in the solving process, but this approach does not scale well – especially due to let-constraints in polymorphism. Now, the entire solving process is encoded in the continuations. We can determine which parts of the constraint have been solved, which part we are currently solving, and which parts are yet to be solved. We will explain this in more detail in the Debuggability section.

6.3 Debuggability

6.3.1 General debugging

We paid particular attention to printing in order to facilitate debugging, for two main reasons: the initial prototype evolved into a project for an M2 MPRI course, so Gabriel added extensive debugging support to make students' lives easier. Furthermore, during the internship, we encountered numerous bugs – mostly due to subtleties in generalization – so we had to dedicate significant time to debugging.

That is why, we implemented several printers – for example, for untyped and typed terms, or for constraints, which we will explain in more detail in the next section. Typed term printing includes type annotations that help us better understand whether our generated terms are well-formed – that is, whether they are correct, i.e., whether the typing yields the term we intended to obtain at the end of the process.

Different debug printings for the various environments were implemented. In the unification phase, we can print the environment to display all the registered variables – which participate in the unification – along with their equivalence class representations. In the generalization phase, we can also print its environment, in addition to the unification one, to observe each variable's level as well as the pools of generalization levels. Finally, in the solver, we can print the current scheme variables and their computed schemes, in addition to the unification and generalization environments.

In addition, we added some `assert` statements in the code. They are mainly located in the *Generalization* part, and to a lesser extent, in the *Unification* part as well. These assertions help us verify certain specifications of the program. For example, they can check preconditions or postconditions to ensure specific properties of the inputs or outputs. They can also represent loop invariants within functions. These assertions were essential for fixing certain bugs, as they revealed not only that a particular specification was incorrect, but also precisely where the issue occurred.

6.3.2 Constraints Printing

We can use the solver's internal continuations to produce a clear and accurate printing of constraints. We can now use the solver's state and the current continuations to print the constraint while reflecting the progress of the resolution. Continuations provide us with exactly what has already been resolved, what we are currently resolving, and what remains to be resolved. In this way, previously resolved constraints are simplified. We can print the current constraint with the complete solver's state – with which we are attempting to resolve the constraint – and with the different environments: unification, generalization, and solver environments. Then, we can include the parts – that remain to be resolved – corresponding to the initial constraint.

We present an example of constraint resolution printing for a simple program such as:

```
lambda a.
  let id = lambda x. x in
  id a
```

At the very beginning of the solving process, we obtain an initial constraint. We can observe that the context is empty and the entire constraint must be resolved:

```
-- hole {}
  (
    let ?final_scheme : ?final_term =
      ∃?a ?wt (?warr = ?a -> ?wt).
        ?final_term = ?warr
        ∧ (let ?scheme_id : ?id =
          ∃?x ?wt/2 (?warr/1 = ?x -> ?wt/2).
            ?id = ?warr/1 ∧ ?wt/2 = ?x ∧ decode ?x
        in
          (∃?wu (?wt/1 = ?wu -> ?wt). ?scheme_id ≤ ?wt/1 ∧ ?wu = ?a)
          ∧ decode_scheme ?scheme_id)
        ∧ decode ?a
    in decode_scheme ?final_scheme
  )
```

The following is an example taken during the solving process, precisely at the moment when the scheme of the `id` function has been successfully computed. We then proceed to solve the body of the **`let`**, namely `id a`, which requires an instantiation represented by `?scheme_id ≤ ?wt/1`:

```
<- let ?final_scheme : ?final_term =
    ∃?a.
      ∃?wt.
        ∃?warr.
          ⊤
          ∧ let ?scheme_id : ?id =
            hole
            {
              Schemes :
                scheme_id: ∀wt/2. id [id wt/2]
              Env :
                warr |--> final_term
                x |--> wt/2
                warr/1 |--> id
                a(0)
                final_term(0) = a -> wt
                wt(0)
                id(G) = wt/2 -> wt/2
                wt/2(G)
              Pool :
                0 |-->
                   warr |--> final_term
                   wt(0)
                   a(0)
                   final_term(0) = a -> wt
            }
            (⊤)
          in
            (∃?wu (?wt/1 = ?wu -> ?wt). ?scheme_id ≤ ?wt/1 ∧ ?wu = ?a)
            ∧ decode_scheme ?scheme_id
          ∧ decode ?a
  in decode_scheme ?final_scheme
```

Finally, we present the printout at the very end of the solving process. The constraint has been successfully resolved, and we obtain the final context used to generate the "solution":

```
<- let ?final_scheme = ⊤
  in
  hole
  {
    Env :
      a |--> wt
      fresh_id |--> wt/1
      fresh_wt |--> wt
      warr |--> final_term
      wu |--> wt
      x |--> wt/2
      warr/1 |--> id
      final_term(G) = wt -> wt
      id(G) = wt/2 -> wt/2
      wt(G)
      wt/1(G) = wt -> wt
      wt/2(G)
  }
  (⊤)
```

6.3.3 Choice paths

We encounter a bug reproducibility issue as the number of well-typed terms increases. For example, with size 11, there are 3,628,944 well-typed terms – composed of variables, applications, lambdas, lets, pairs, and let-pairs – using ML-style polymorphism. The problem arises when we want to test whether any assertions fail: generating all well-typed terms of size 11 exhaustively can take too much time – up to 5 minutes in this case – before encountering a term that triggers the assertion, making it difficult to debug efficiently.

This is why we came up with a mechanism to resume the search from a specific point. The idea is to record the *path* followed in the search monad tree so that it can be displayed once a term is successfully generated. This amounts to encoding a sequence of constructor choices: first selecting a specific constructor, then others in a particular order to progressively build the term.

With this concept of paths, we implemented a second `run` function that allows us to start not from the beginning of the search tree, but from a predefined path. This way, we can begin an exhaustive enumeration of well-typed terms and print the corresponding path in the search tree. If the enumeration crashes, we can retrieve the path of the last successfully generated term and resume the search from that point, saving a significant amount of time.

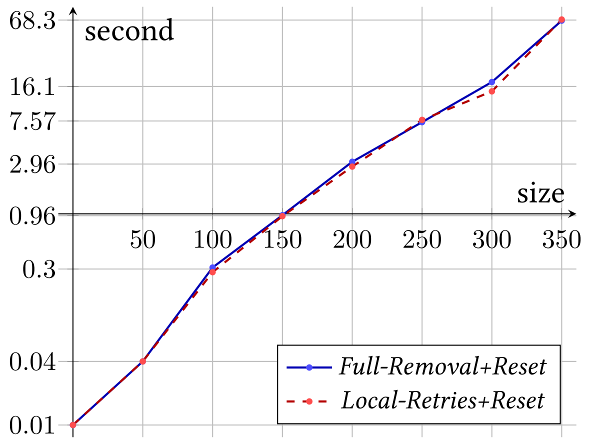
An implementation of this sequential search monad is provided in the Appendix. We did not implement this method for the random search monad due to the inherent randomness, which makes the task more difficult.

6.4 Results

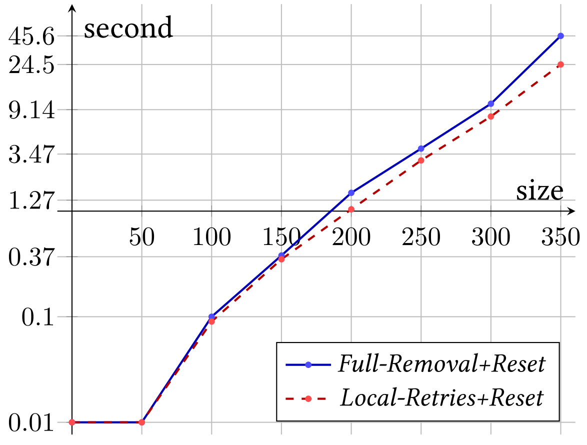
In this section, we present performance results for the final version of the software, which includes ML-style polymorphism. We carried out the experiments on an *Intel(R) Core(TM) i7-10510U CPU* with its frequency fixed at *2GHz*. Evaluating the performance of our random term generator is challenging due to its stochastic nature: most of the time, generating a term takes roughly the same amount of time, but occasionally it takes significantly longer. In addition, we assume that the generation time follows an exponential distribution. Consequently, we consider the geometric mean to be a more accurate indicator of the expected generation time than the arithmetic mean.

6.4.1 Generation of a random well-typed term

The performance results for generating a single typed term of a given size with the *Full-Removal+Reset* and *Local-Retries+Reset* random-search monads are shown below:



Arithmetic mean



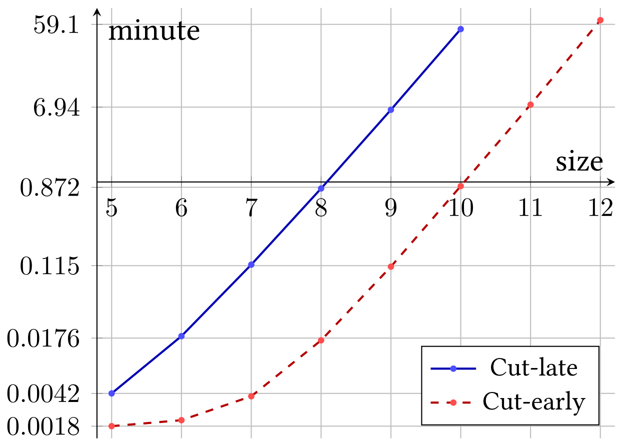
Geometric mean

These graphs show that the *Local-Retries+Reset* random-search monad is slightly faster than *Full-Removal+Reset*. However, *Full-Removal+Reset* exhibits lower variance. For example, at $size = 350$ it achieves a lower arithmetic mean while its geometric mean remains higher, indicating fewer pathological cases. This behaviour likely results from situations in which locally retrying is less efficient than restarting from scratch.

In summary, generating a term of size 100 is virtually instantaneous, and a term of size 200 can be produced in around one second on average (geometric mean). Moreover, once several terms have been generated, subsequent generations become faster because the generator reuses elements computed during earlier runs. Our prototype appears to scale sufficiently for real-world use, especially considering that it can be further optimized – for instance, by using more advanced data structures.

6.4.2 Exhaustive enumeration of well-typed terms

This benchmark compares the *cut-late* and *cut-early* strategies for the exhaustive enumeration of all well-typed terms of a given size:



Computation time of exhaustive enumeration

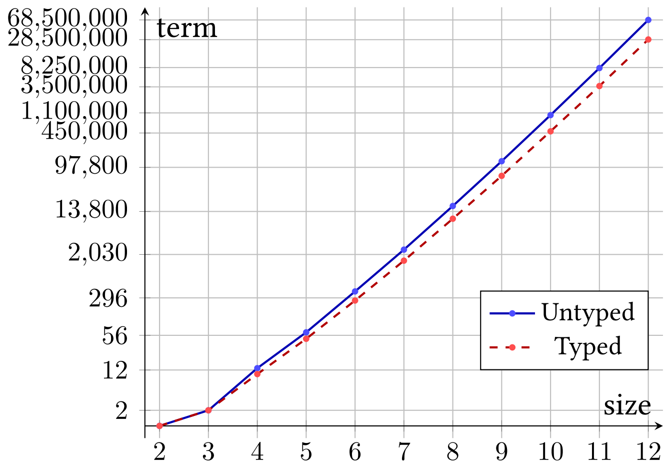
For sizes below 5, the computation time is instantaneous. We do not have the computation time for *cut-late* at sizes 11 and 12, but we know that for $size = 11$, the computation time exceeds 3 hours. Similarly, we lack the computation time for *cut-early* at size 13, although we know it exceeds 2 hours. Given the difference in computation time between *cut-late* and *cut-early*, it appears that *cut-early* offers a *significant* time saving compared to *cut-late*.

6.4.3 Measuring the benefits of our approach

Our approach is based on the idea that interleaving generation and type-checking is significantly more efficient than generating all well-syntactically-scoped terms first, and then type-checking them. We try to evaluate this assumption by estimating the proportion of well-typed terms among well-syntactically-scoped terms.

Small sizes: exhaustive enumeration

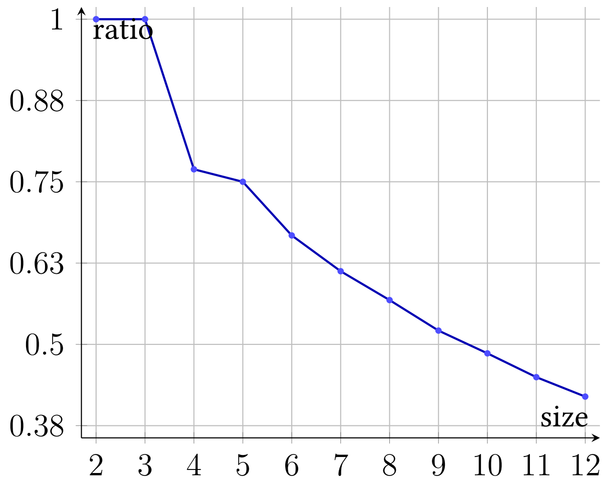
We begin by counting the number of well-syntactically-scoped and well-typed terms for a given size, including pairs and let-pairs. For small sizes this can be computed exactly.



Number of untyped and typed terms

Size	Untyped	Typed
0	0	0
1	0	0
2	1	1
3	2	2
4	13	10
5	64	48
6	394	263
7	2 524	1 546
8	17 560	9 975
9	128 560	67 009
10	997 587	485 011
11	8 071 358	3 628 944
12	68 474 601	28 739 233

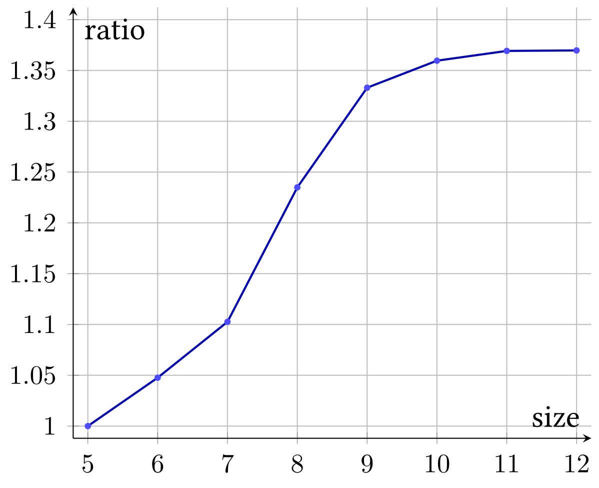
As expected, the number of well-syntactically-scoped terms increases exponentially. The number of well-typed terms also grows exponentially, but at a slower rate than the well-syntactically-scoped terms. This supports our claim that our approach will be more efficient. We can further examine the ratio between well-typed and well-syntactically-scoped terms to confirm that it tends toward 0. Such a trend would indicate that our approach will be significantly more effective:



Ratio between well-typed and well-syntactically-scoped terms

Size	Ratio
2	1.0000
3	1.0000
4	0.7692
5	0.7500
6	0.6675
7	0.6125
8	0.5681
9	0.5212
10	0.4862
11	0.4496
12	0.4197

As expected, the ratio tends towards 0 as the size increases, indicating that our approach becomes increasingly more efficient than the naive one. To verify this assumption, we compare the execution time of the naive approach with our interleaved strategy. The following graphs show the execution time for the exhaustive enumeration of well-typed terms using the naive approach, along with the ratio between it and the interleaved approach:

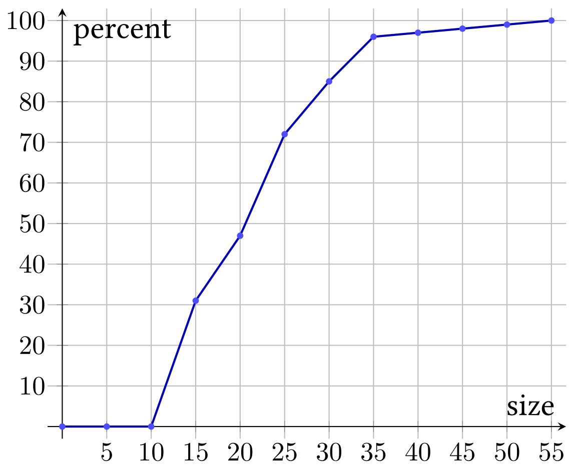


Ratio between naive and interleaving

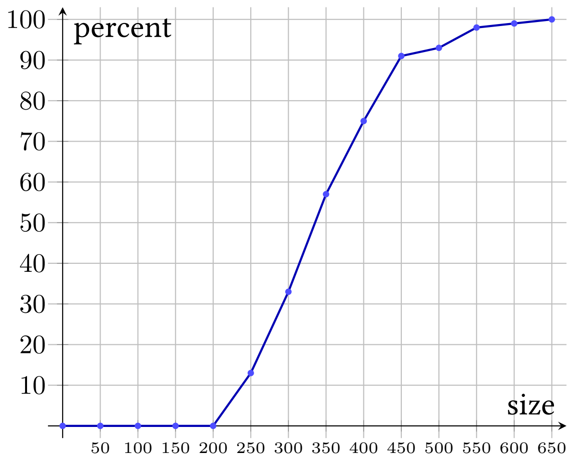
As we can observe, our approach becomes progressively more efficient as the size of the terms increases, compared to the naive method. In this case, the performance gains are not as substantial as expected because the ratio between well-typed and well-syntactically-scoped terms remains around 40% at size 12. However, as this ratio decreases with larger sizes, the performance gains will increase proportionally.

Larger sizes: random generation

Comparing the generation of random well-typed terms using the naive and interleaving approaches is challenging. Starting from $size = 15$, the naive method becomes highly unstable: generation is either instantaneous or takes an excessively long and unpredictable amount of time – for example, it can sometimes take well over 15 minutes when $size = 15$. One idea, to measure the difference between the naive and interleaving method, involves computing the proportion of timeouts for both methods. We allow a maximum of 10 seconds to generate a single well-typed term of a given size before stopping:



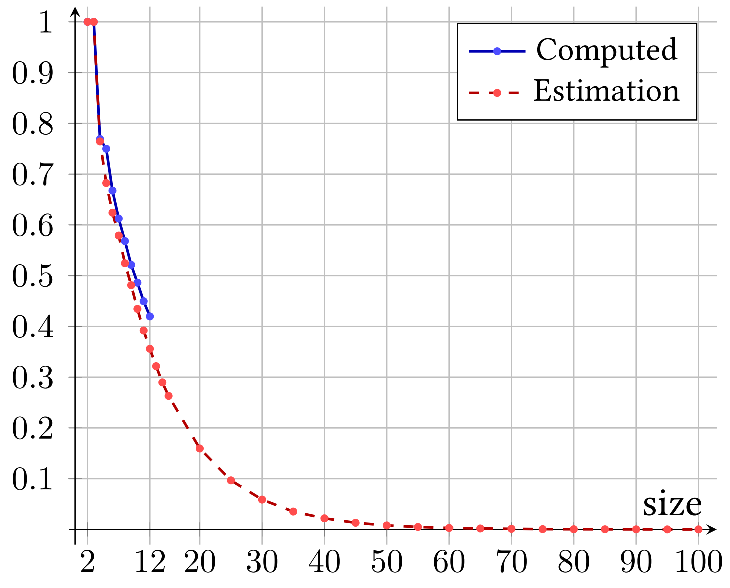
Proportion of timeouts with Naive



Proportion of timeouts with Interleaving

For the interleaving approach, the proportion of timeouts remains at 0 up to $size = 200$. As expected, from $size = 15$ onward, our method of random generation becomes significantly more efficient, with time savings that appear to grow exponentially. This is due to the fact that, from this size, the proportion of timeouts for the naive approach increases drastically. For the interleaving method, timeouts only begin to occur at $size = 250$. Moreover, the proportion of timeouts increases significantly more slowly than with the naive approach: it is only from $size = 650$ that timeouts always occur. This shows that our method scales better for larger sizes than the naive one.

If the distribution of our generator was uniform, we could estimate the proportion of well-typed terms among the well-syntactically-scoped terms for larger size by sampling a large number of untyped terms and compute how many of them are well-typed. The distribution of our generator is not uniform but it is relatively close. In this way, our estimations are close to the real proportions – we can compare the results for small sizes, for $size = 12$, the real proportion is 0.4197 and the estimation is 0.3559. In addition, we use the same generator – and thus the same distribution – for the naive and the interleaving approaches, so this estimation correctly predicts the performance difference between them:



Ratio between well-typed and well-syntactically-scoped terms

Size	Estimated ratio
5	0.6824
10	0.4345
15	0.2631
20	0.1595
25	0.0967
30	0.0588
35	0.0352
40	0.0219
45	0.0131
50	0.0079
55	0.0049
60	0.0029
65	0.0019
70	0.0012
75	0.0006
80	0.0004
85	0.0003
90	0.0002
95	0.0001
100	0.0001

From the graph, it is clear that the ratio between well-typed and well-syntactically-scoped terms tends towards 0. This illustrates why our interleaving method scales better than the naive approach when generating random well-typed terms of larger sizes: although the proportion of well-typed terms is nearly 0 for $size = 100$, the interleaving allows us to generate terms instantaneously, unlike the naive approach.

In summary, for $size \geq 15$, our interleaving approach becomes increasingly more efficient than the naive method for generating random well-typed terms, with time savings that appear to be exponential.

7 Conclusion

7.1 Summary

We studied constraint-based type inference for ML-style polymorphism. Then, we presented a strategy for generating random well-typed terms by interleaving type-checking and generation. This approach aims to be more efficient than the naive method, which consists in generating all well-syntactically-scoped terms before performing type-checking on each of them.

At the beginning of the internship, we focused on performance issues. First, we explored random search monads, and later, we examined how to describe the search space more precisely .

We also made significant modifications to the prototype to support ML-style polymorphism. This required a complete reconsideration of our approach to the solver, particularly in how we manage the state during type-checking and generation.

Additionally, we undertook a debugging effort, primarily focused on the printing of constraints. Finally, we implemented a mechanism that allows restarting from a specific point in the search space.

Our prototype scales well enough for real-world applications; we achieved promising results. we are able to generate random well-typed terms of size 100 almost instantaneously, and terms of size 200 in approximately one second. Furthermore, we can exhaustively list all well-typed terms of size less than 12 in under 70 minutes.

7.2 Limitations

The first limitation lies in our not fully satisfactory understanding of the performance issues related to random search monads. We believe that a significant part of the problem is understood, but it is still not fully clear.

Initially, we expected better results from exhaustive enumeration. In fact, for the sizes we tested – from 1 to 12 – we were roughly 50% more efficient than the naive approach. We know that we will become more efficient as the size of the generated terms increases, but we had hoped to achieve greater time savings.

7.3 Future Work

It would be useful to fully understand all the issues related to performance and complexity: why certain random search monads are more efficient than others, and similar questions apply to the design of search spaces. We believe that a significant part of the problem is understood, but it is still not fully clear. It could also be interesting to study the exponential nature of the problem, and especially whether interleaving can allow us to outperform the exponential growth in the size of the desired well-typed term.

One idea is to extend the work to handle more type features, in order to generate well-typed terms in a more expressive type system. To conclude this subject, it would be beneficial to implement a type checker for System F to ensure that our witnesses are well-formed. For example, there is an issue with let-tuples: we have to find a way to add type abstractions for each component of a tuple.

Another area of work is improving performance. Although the overall algorithm is efficient in theory, it can be significantly optimized to become much more efficient in practice. One proposal is to use Store-backed mutable state to perform rank updates as *O(1)* reference writes. Another idea is to delay the occurs check until generalization time, rather than performing it after each unification.

7.4 Acknowledgements

I would like to begin by thanking the entire lab team for their warm welcome, hospitality, and kindness. I am especially grateful to Gabriel Scherer for his contagious enthusiasm, excellent guidance, and invaluable support throughout this interesting and enriching internship.

References

Burke Fetscher, Koen Claessen, Michał Pałka, John Hughes, and Robert Bruce Findler. Making random judgments: Automatically generating well-typed terms from the definition of a type-system. In Jan Vitek, editor, *Programming Languages and Systems*, pages 383–405, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 978-3-662-46669-8.

Oleg Kiselyov. How ocaml type checker works – or what polymorphism and garbage collection have in common. `https://okmij.org/ftp/ML/generalization.html`.

Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, page 91–97, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450305921. doi: 10.1145/1982595.1982615. URL `https://doi.org/10.1145/1982595.1982615`.

François Pottier. Hindley-Milner Elaboration in Applicative Style. In *ICFP 2014: 19th ACM SIGPLAN International Conference on Functional Programming*, Goteborg, Sweden, September 2014. ACM. doi: 10.1145/2628136.2628145. URL `https://inria.hal.science/hal-01081233`.

Francois Pottier and Didier Rémy. *The Essence of ML Type Inference*, pages 389–489. 01 2005.

François Pottier, Gabriel Scherer, and Olivier Martinot. Inferno. `https://gitlab.inria.fr/fpottier/inferno`, 2022.

Gabriel Scherer. Constrained generation of well-typed program. Technical report, Inria - Paris 7, June 2024. URL `https://inria.hal.science/hal-04607309`.

Appendix

The OCaml Prototype-Naive Monad Implementation

This section presents the implementation of the random search monad used in the prototype:

```
type 'a t =
  | Return : 'a -> 'a t
  | Fail : 'a t
  | Delay : 'a t Lazy.t -> 'a t
  | Map : 'a t * ('a -> 'b) -> 'b t
  | Bind : 'a t * ('a -> 'b t) -> 'b t
  | Sum : 'a t list -> 'a t
  | One_of : 'a array -> 'a t

let is_empty = function
  | Fail -> true
  | _ -> false

let return v = Return v

let fail = Fail

let delay f = Delay (Lazy.from_fun f)

let map f t = if is_empty t then Fail else Map (t, f)

let bind t f = if is_empty t then Fail else Bind (t, f)

let sum ts =
  match List.filter (fun t -> not (is_empty t)) ts with
  | [] -> Fail
  | ts -> Sum ts

let one_of arr = One_of arr

let rec next : type a. a t -> a option * a t =
 fun t ->
  match t with
  | Return x -> (Some x, t)
  | Fail -> (None, t)
  | Delay f -> next (Lazy.force f)
  | Map (t, f) ->
    let o, t = next t in
    (Option.map f o, map f t)
  | One_of arr ->
    if arr = [||] then (None, Fail)
    else (Some arr.(Random.int (Array.length arr)), t)
  | Sum ts ->
    if ts = [] then (None, Fail)
    else begin
      let arr = Array.of_list ts in
      let i = Random.int (Array.length arr) in
      let o, t = next arr.(i) in
      arr.(i) <- t;
      (o, sum (Array.to_list arr))
    end
  | Bind (tt, f) -> begin
    let o, tt = next tt in
    match o with
    | None -> (None, Bind (tt, f))
    | Some t -> (fst (next (f t)), Bind (tt, f))
  end

let rec run (gen : 'a t) : 'a Seq.t =
 fun () ->
  let o, gen = next gen in
  match o with
  | None -> run gen ()
  | Some v -> Seq.Cons (v, run gen)
```

The OCaml Neven Monad Implementation

This section provides the implementation of the Neven random search monad, which is significantly more efficient than the prototype version:

```
type 'a t = {
  finished : 'a list;
  later : (unit -> 'a t) list;
}

let of_finished (a : 'a list) : 'a t = { finished = a; later = [] }

let of_later (a : (unit -> 'a t) list) : 'a t = { finished = []; later = a }

let empty : 'a t = of_finished []

let rec map (f : 'a -> 'b) (s : 'a t) : 'b t =
  { finished = List.map f s.finished;
    later = List.map (fun gen () -> map f (gen ())) s.later }

let return (x : 'a) : 'a t = of_finished [ x ]

let delay (f : unit -> 'a t) : 'a t = of_later [ f ]

let concat (s : 'a t) (s' : 'a t) : 'a t =
  { finished = s.finished @ s'.finished; later = s.later @ s'.later }

let rec bind (sa : 'a t) (f : 'a -> 'b t) : 'b t =
  sa.finished
  |> List.map f
  |> List.fold_left concat (bind_later f sa)

and bind_later (f : 'a -> 'b t) (s : 'a t) =
  of_later (List.map (fun s () -> bind (s ()) f) s.later)

let sum (li : 'a t list) : 'a t = List.fold_left concat empty li

let fail : 'a t = empty

let one_of (vs : 'a array) : 'a t =
  vs
  |> Array.to_list
  |> of_finished

type 'a chosen =
  | Picked of 'a
  | Retry
  | Empty

let rec take_nth (l : 'a list) (n : int) : 'a * 'a list =
  match (l, n) with
  | h :: t, i when i <= 0 -> (h, t)
  | h :: t, i ->
    let out, rem = take_nth t (i - 1) in
    (out, h :: rem)
  | [], _ -> raise Not_found

let run (s : 'a t) : 'a Seq.t =
  let rec try_pick (sampler : 'a t) : 'a chosen * 'a t =
    let pick_finished () =
      let idx = Random.int (List.length sampler.finished) in
      let x, rest = take_nth sampler.finished idx in
      (Picked x, { sampler with finished = rest })
    in

    let pick_later () =
      let idx = Random.int (List.length sampler.later) in
      let gen, trimmed = take_nth sampler.later idx in
      let res, gen = try_pick (gen ()) in
      match res with
      | Picked x ->
        (Picked x, { sampler with later = (fun () -> gen) :: trimmed })
      | Retry -> (Retry, { sampler with later = (fun () -> gen) :: trimmed })
      | Empty -> (Retry, { sampler with later = trimmed })
    in

    if sampler.finished = [] && sampler.later = [] then (Empty, sampler)
    else if sampler.later = [] then pick_finished ()
    else if sampler.finished = [] then pick_later ()
    else if Random.bool () then pick_finished ()
    else pick_later ()
  in

  let rec pick (sampler : 'a t) : 'a * 'a t =
    let res, trimmed = try_pick sampler in
    match res with
    | Picked t -> (t, trimmed)
    | Retry -> pick trimmed
    | Empty -> failwith "This generator is empty; no such term exists"
  in

  let rec seq (sampler : 'a t) () =
    let res, trimmed = pick sampler in
    Seq.Cons (res, seq trimmed)
  in

  seq s
```

The OCaml Full-Removal Monad Implementation

This section presents the implementation of the monad using the Full-Removal strategy. Its goal is to eliminate empty generators and avoid proposing the same term multiple times when filling a `Do` node:

```
type 'a t =
  | Return : 'a -> 'a t
  | Fail : 'a t
  | Delay : 'a t Lazy.t -> 'a t
  | Map : 'a t * ('a -> 'b) -> 'b t
  | Bind : 'b t option * 'a t * ('a -> 'b t) -> 'b t
  | Sum : 'a t list -> 'a t
  | One_of : 'a list -> 'a t

let return v = Return v

let fail = Fail

let delay f = Delay (Lazy.from_fun f)

let map f t = if t = Fail then Fail else Map (t, f)
let sum ts = Sum (List.filter (( <> ) fail) ts)

let sum_cons t ts = if t = Fail then Sum ts else Sum (t :: ts)

let bind ta f = if ta = Fail then Fail else Bind (None, ta, f)

let bind_cons tb ta f = if tb = Fail then bind ta f else Bind (Some tb, ta, f)

let one_of arr = One_of (Array.to_list arr)

let one_of_list = function
  | [] -> Fail
  | li -> One_of li

let list_pop_rand = function
  | [] -> None
  | li ->
    let len = List.length li in
    let i = Random.int len in
    Some (List.nth li i, List.filteri (fun j _ -> j <> i) li)

let rec next : type a. a t -> a option * a t = function
  | Return x -> (Some x, Fail)
  | Fail -> (None, Fail)
  | Delay f -> next (Lazy.force f)
  | Map (t, f) ->
    let o, t = next t in
    (Option.map f o, map f t)
  | One_of li -> begin
    match list_pop_rand li with
    | None -> (None, Fail)
    | Some (a, li) -> (Some a, one_of_list li)
  end
  | Sum ts -> begin
    match list_pop_rand ts with
    | None -> (None, Fail)
    | Some (t, ts) ->
      let o, t = next t in
      (o, sum_cons t ts)
  end
  | Bind (Some tb, ta, f) ->
    let ob, tb = next tb in
    (ob, bind_cons tb ta f)
  | Bind (None, ta, f) -> begin
    let oa, ta = next ta in
    match oa with
    | None -> (None, bind ta f)
    | Some a -> next (bind_cons (f a) ta f)
  end

let rec run (gen : 'a t) : 'a Seq.t =
 fun () ->
  if gen = Fail then Seq.Nil
  else
    let o, gen = next gen in
    match o with
    | None -> run gen ()
    | Some v -> Seq.Cons (v, run gen)
```

The OCaml Local-Retries Monad Implementation

This section details the implementation of the monad using the Local-Retries strategy. In addition to the Full-Removal strategy, it aims to retry finding terms locally rather than re-drawing from the top level:

```
type 'a t =
  | Return : 'a -> 'a t
  | Fail : 'a t
  | Delay : 'a t Lazy.t -> 'a t
  | Map : 'a t * ('a -> 'b) -> 'b t
  | Bind : 'b t option * 'a t * ('a -> 'b t) -> 'b t
  | Sum : 'a t list -> 'a t
  | One_of : 'a list -> 'a t

let return v = Return v

let fail = Fail

let delay f = Delay (Lazy.from_fun f)

let map f t = if t = Fail then Fail else Map (t, f)

let sum ts = Sum (List.filter (( <> ) fail) ts)

let sum_cons t ts = if t = Fail then Sum ts else Sum (t :: ts)

let bind ta f = if ta = Fail then Fail else Bind (None, ta, f)

let bind_cons tb ta f = if tb = Fail then bind ta f else Bind (Some tb, ta, f)

let shuffle arr =
  for i = Array.length arr - 1 downto 1 do
    let j = Random.int (i + 1) in
    let tmp = arr.(j) in
    arr.(j) <- arr.(i);
    arr.(i) <- tmp
  done

let one_of arr =
  shuffle arr;
  One_of (Array.to_list arr)

let one_of_list = function
  | [] -> Fail
  | li -> One_of li

let list_pop_rand = function
  | [] -> None
  | li ->
    let len = List.length li in
    let i = Random.int len in
    Some (List.nth li i, List.filteri (fun j _ -> j <> i) li)

let rec next : type a. a t -> a option * a t =
 fun t ->
  let retry o rest =
    if o <> None || rest = Fail then (o, rest)
    else next rest
  in

  match t with
  | Return x -> (Some x, Fail)
  | Fail -> (None, Fail)
  | Delay f -> next (Lazy.force f)
  | Map (t, f) ->
    let o, t = next t in
    retry (Option.map f o) (map f t)
  | One_of li -> begin
    match list_pop_rand li with
    | None -> (None, Fail)
    | Some (a, li) -> (Some a, one_of_list li)
  end
  | Sum ts -> begin
    match list_pop_rand ts with
    | None -> (None, Fail)
    | Some (t, ts) ->
      let o, t = next t in
      retry o (sum_cons t ts)
  end
  | Bind (Some tb, ta, f) ->
    let ob, tb = next tb in
```

```
      retry ob (bind_cons tb ta f)
    | Bind (None, ta, f) -> begin
      let oa, ta = next ta in
      match oa with
      | None -> retry None (bind ta f)
      | Some a -> next (bind_cons (f a) ta f)
    end

let rec run (gen : 'a t) : 'a Seq.t =
 fun () ->
   if gen = Fail then Seq.Nil
   else
     let o, gen = next gen in
     match o with
     | None -> run gen ()
     | Some v -> Seq.Cons (v, run gen)
```

The OCaml Full-Removal_with_Reset Monad Implementation

This section provides the implementation of the monad employing the Full-Removal with Reset strategy. In addition to the Full-Removal strategy, it reverts to the top level when encountering persistent dead ends:

```
let limit = 1000

type 'a t =
    | Return : 'a -> 'a t
    | Fail : 'a t
    | Delay : 'a t Lazy.t -> 'a t
    | Map : 'a t * ('a -> 'b) -> 'b t
    | Bind : 'b t option * 'a t * ('a -> 'b t) -> 'b t
    | Sum : 'a t list -> 'a t
    | One_of : 'a list -> 'a t

let return v = Return v

let fail = Fail

let delay f = Delay (Lazy.from_fun f)

let map f t = if t = Fail then Fail else Map (t, f)

let sum ts = Sum (List.filter (( <> ) fail) ts)

let sum_cons t ts = if t = Fail then Sum ts else Sum (t :: ts)

let bind ta f = if ta = Fail then Fail else Bind (None, ta, f)
let bind_cons tb ta f = if tb = Fail then bind ta f else Bind (Some tb, ta, f)

let one_of arr = One_of (Array.to_list arr)

let one_of_list = function
  | [] -> Fail
  | li -> One_of li

let list_pop_rand = function
  | [] -> None
  | li ->
    let len = List.length li in
    let i = Random.int len in
    Some (List.nth li i, List.filteri (fun j _ -> j <> i) li)

let rec next : type a. a t -> a option * a t = function
  | Return x -> (Some x, Fail)
  | Fail -> (None, Fail)
  | Delay f -> next (Lazy.force f)
  | Map (t, f) ->
    let o, t = next t in
    (Option.map f o, map f t)
  | One_of li -> begin
    match list_pop_rand li with
    | None -> (None, Fail)
    | Some (a, li) -> (Some a, one_of_list li)
  end
  | Sum ts -> begin
    match list_pop_rand ts with
    | None -> (None, Fail)
    | Some (t, ts) ->
      let o, t = next t in
      (o, sum_cons t ts)
  end
  | Bind (Some tb, ta, f) ->
    let ob, tb = next tb in
    (ob, bind_cons tb ta f)
  | Bind (None, ta, f) -> begin
    let oa, ta = next ta in
    match oa with
    | None -> (None, bind ta f)
    | Some a -> next (bind_cons (f a) ta f)
  end

let tries = ref 1

let rec run orig_gen =
  let start = !tries in

  let rec run_ (gen : 'a t) : 'a Seq.t =
   fun () ->
    incr tries;
    if gen = Fail then Seq.Nil
    else
      match next gen with
      | None, gen ->
        if !tries - start > limit then run orig_gen () else run_ gen ()
      | Some v, gen -> Seq.Cons (v, run gen)
  in
  run_ orig_gen
```

The OCaml Local-Retries_with_Reset Monad Implementation

This section describes the implementation of the monad based on the Local-Retries with Reset strategy. In addition to the Full-Removal and Local-Retries strategies, it restarts from the top level when encountering deep dead ends:

```
let limit = 1000

type 'a t =
  | Return : 'a -> 'a t
  | Fail : 'a t
  | Delay : 'a t Lazy.t -> 'a t
  | Map : 'a t * ('a -> 'b) -> 'b t
  | Bind : 'b t option * 'a t * ('a -> 'b t) -> 'b t
  | Sum : 'a t list -> 'a t
  | One_of : 'a list -> 'a t

let return v = Return v

let fail = Fail

let delay f = Delay (Lazy.from_fun f)

let map f t = if t = Fail then Fail else Map (t, f)

let sum ts = Sum (List.filter (( <> ) fail) ts)

let sum_cons t ts = if t = Fail then Sum ts else Sum (t :: ts)

let bind ta f = if ta = Fail then Fail else Bind (None, ta, f)

let bind_cons tb ta f = if tb = Fail then bind ta f else Bind (Some tb, ta, f)

let shuffle arr =
  for i = Array.length arr - 1 downto 1 do
    let j = Random.int (i + 1) in
    let tmp = arr.(j) in
    arr.(j) <- arr.(i);
    arr.(i) <- tmp
  done

let one_of arr =
  shuffle arr;
  One_of (Array.to_list arr)

let one_of_list = function
  | [] -> Fail
  | li -> One_of li

let list_pop_rand = function
  | [] -> None
  | li ->
    let len = List.length li in
    let i = Random.int len in
    Some (List.nth li i, List.filteri (fun j _ -> j <> i) li)

let tries = ref 1

let start = ref !tries

exception Reset

let rec next : type a. a t -> a option * a t =
 fun t ->
  let retry o rest =
    if o <> None || rest = Fail then (o, rest)
    else if !tries - !start > limit then raise Reset
    else begin
      incr tries;
      next rest
    end
  in

  match t with
  | Return x -> (Some x, Fail)
  | Fail -> (None, Fail)
  | Delay f -> next (Lazy.force f)
  | Map (t, f) ->
    let o, t = next t in
    retry (Option.map f o) (map f t)
  | One_of li -> begin
```

```
    match list_pop_rand li with
    | None -> (None, Fail)
    | Some (a, li) -> (Some a, one_of_list li)
  end
  | Sum ts -> begin
    match list_pop_rand ts with
    | None -> (None, Fail)
    | Some (t, ts) ->
      let o, t = next t in
      retry o (sum_cons t ts)
  end
  | Bind (Some tb, ta, f) ->
    let ob, tb = next tb in
    retry ob (bind_cons tb ta f)
  | Bind (None, ta, f) -> begin
    let oa, ta = next ta in
    match oa with
    | None -> retry None (bind ta f)
    | Some a -> next (bind_cons (f a) ta f)
  end

let rec run orig_gen =
  start := !tries;

  let rec run_ (gen : 'a t) : 'a Seq.t =
   fun () ->
    incr tries;
    if gen = Fail then Seq.Nil
    else
      match next gen with
      | exception Reset -> run orig_gen ()
      | None, gen -> run_ gen ()
      | Some v, gen -> Seq.Cons (v, run gen)
  in
  run_ orig_gen
```

An OCaml `cut_size` Implementation

This implementation of the `cut_size` function supports ML-style polymorphism, type annotations, and tuples (which in this case are simply pairs). It also respects debugging constraints such as source term location:

```
let rec cut_size ~size (term : Untyped(M).term) : Untyped(M).term =
  let un ~size t f : Untyped(M).term =
    let size = size - 1 in
    if size < 1 then Do M.fail else f (cut_size ~size t)
  in
  let bin ~size ta tb f : Untyped(M).term =
    let size = size - 1 in
    if size < 2 then Do M.fail
    else
      Do (M.sum (
        List.init (size - 1) (fun idx ->
          (* { i, j | i > 0, j > 0, i+j = size } *)
          let i = idx + 1 in
          let j = size - i in
          let ta' = cut_size ~size:i ta in
          let tb' = cut_size ~size:j tb in
          M.return (f ta' tb')
        )
      ))
  in

  if size <= 0 then Do M.fail
  else
    match term with
    | Var _ -> if size = 1 then term else Do M.fail
    | App (t, u) -> bin ~size t u (fun t' u' -> App (t', u'))
    | Abs (x, t) -> un ~size t (fun t' -> Abs (x, t'))
    | Let (x, t, u) -> bin ~size t u (fun t' u' -> Let (x, t', u'))
    | Tuple ts -> begin
      match ts with
      | [ t; u ] -> bin ~size t u (fun t' u' -> Tuple [ t'; u' ])
      | _ -> assert false
    end
    | LetTuple (xs, t, u) ->
      bin ~size t u (fun t' u' -> LetTuple (xs, t', u'))
    | Annot (t, ty) -> un ~size t (fun t' -> Annot (t', ty))
    | Do m -> Do (M.map (cut_size ~size) m)
    | Loc (loc, t) -> un ~size t (fun t' -> Loc (loc, t'))
```

Choice Path Implementation

The initial implementation of the sequential search monad:

```
type 'a t = 'a Seq.t

let map = Seq.map

let delay f = fun () -> f () ()

let return = Seq.return

let bind s f = Seq.concat_map f s
let fail = Seq.empty

let one_of = Array.to_seq

let sum seqs = seqs |> List.to_seq |> Seq.concat

let run = Fun.id
```

We can observe that the initial implementation of the sequential search monad was fairly straightforward. We will now add the code required to generate the traversal path and to resume the search from a given path:

```
module ChoicePath = struct
  type t =
    | Nil
    | Return
    | Fail
    | Sum of int * t
    | Bind of t * t
end

type path = ChoicePath.t
open ChoicePath

let invalid_path f path =
  Printf.ksprintf failwith "MSeq.%s: invalid path '%s'" f
    (path |> ChoicePathPrinter.print |> Utils.string_of_doc)


type 'a t = path -> (path * 'a) Seq.t

let map f s = fun p -> s p |> Seq.map (fun (p', v) -> (p', f v))

let delay ds = fun p -> ds () p

let return v : 'a t =
 fun p ->
  match p with
  | Nil | Return -> Seq.return (Return, v)
  | Fail | Sum _ | Bind _ -> invalid_path "return" p


let bind s f =
 fun p ->
  let pa, first_pb =
    match p with
    | Nil -> (Nil, Nil)
    | Bind (pa, pb) -> (pa, pb)
```

```
      | Fail | Return | Sum _ -> invalid_path "bind" p
    in
    s pa
    |> Seq.mapi (fun i (pa', sa) -> (pa', (if i = 0 then first_pb else Nil), sa))
    |> Seq.concat_map (fun (pa', pb, va) ->
        f va pb |> Seq.map (fun (pb', vb) -> (Bind (pa', pb'), vb)))


let fail =
  fun p ->
    match p with
    | Nil | Fail -> Seq.empty
    | Return | Sum _ | Bind _ -> invalid_path "fail" p


let one_of arr =
  fun p ->
    let start =
      match p with
      | Nil -> 0
      | Sum (i, Return) ->
        if i >= Array.length arr then invalid_path "sum" p;
        i
      | Return | Fail | Sum _ | Bind _ -> invalid_path "one_of" p
    in
    Seq.ints start
    |> Seq.take_while (fun i -> i < Array.length arr)
    |> Seq.map (fun i -> (Sum (i, Return), arr.(i)))


let sum ss =
  fun p ->
    let start, start_p =
      match p with
      | Nil -> (0, Nil)
      | Sum (i, i_p) ->
        if i >= List.length ss then invalid_path "sum" p;
        (i, i_p)
      | Return | Fail | Bind _ -> invalid_path "sum" p
    in
    ss
    |> List.to_seq
    |> Seq.mapi (fun i s -> (i, s))
    |> Seq.concat_map (fun (i, s) ->
        Seq.map (fun (p, v) -> (Sum (i, p), v))
        (if i < start then Seq.empty else if i = start then s start_p else s Nil))
```

```
let run s = s Nil |> Seq.map snd

let run' s = fun p -> s p
```