Constrained generation of well-typed program

Gabriel Scherer

Constrained generation of well-typed program

Preliminary report **June 10, 2024**

GABRIEL SCHERER, INRIA, France

We report on a work in progress, building a random generator of well-typed program using a constraint-based type inference engine.

1 INTRODUCTION

"Fuzzing" is a popular testing approach where we throw random inputs at a program and check that it respects some expected properties on all inputs. Fuzzing a metaprogram such as a compiler requires producing random input programs. To exercise interesting behaviors of the compiler, we want our input programs to be well-formed in a strong sense, in particular to be well-typed. Generating random well-typed program is difficult when the type-system is powerful, there is a whole scientific literature on the topic, see for example Palka, Claessen, Russo and Hughes [2011].

Most existing generators integrate a large part of the implementation of a type-checker, "inverted" to assist in top-down random term generation by efficiently discarding choices that result in ill-typed programs. Scaling this approach to a full language would result in implementing two sophisticated type-checkers, once in the compiler and once in the program generator. This is one type-checker too many!

In this report we present preliminary work on reusing the constraint-based type inference approach [Pottier and Rémy 2005] to write a single type-checker that can be used for both purposes: to check types of user-provided programs, and to efficiently guide a random program generator. This only requires a fairly simple modification to constraint generators: parametrizing them over a search monad.

Our software prototype is available at https://gitlab.com/gasche/constraints-for-random-generation.

1.1 Intuition

Instead of generating well-typed program from the start – which requires encoding complex typing rules in the generator – we propose to generate untyped programs, which is much easier to do, and use a *type inference* engine during the generation process to filter out ill-typed program fragments. Note that generating complete untyped programs first, and then filtering out the ill-typed ones would be very inefficient. We need to find a way to interleave generation and type-inference in a more fine-grained way, to reject ill-typed fragments as soon as possible and spend our time exploring the well-typed program space.

Our initial initial idea to achieve this interleaving was to manipulate terms and constraints with *holes*.

Start with just a hole `?`, then refine it by choosing a term constructor with sub-holes, for example `fun x -> ?`, which may later be refined into any term of the form `fun x -> t`. Calling the constraint generator on `fun x -> ?` returns a constraint with holes, as some parts of the constraint are still unknown. We can partially solve this constraint, evolve the solver state on the parts that are known and possibly determine that the constraints is already unsatisfiable.

Once we have solved all the known parts of the constraint, we can refine one of the holes in the source term, replace the corresponding constraint hole by a more precise constraint-with-holes, partially solve it, etc., repeating the process until we have a complete term or an error.

Filling a source-term hole is a choice point; by enumerating all possible term-formers-with-holes, we can have an enumerator of all well-typed terms, and by sampling them randomly we can

Author's address: Gabriel Scherer, INRIA, France.

generate random terms. If we can reuse the current solver state for each of these "continuations" of the generation process (note that this requires that the solver state is persistent or at least backtrackable), this means that we can share type-checking work for all terms that share a common tree prefix. We hope that this can make generation of a well-typed efficient in practice.

In fact we have found that it is *not necessary* to introduce a notion of holes in our syntax of terms and constraints to interleaving term generation and constraint-solving, this can be done in a simpler, more abstract way by injecting a constructor for an arbitrary functor – our terms and constraints are not pure syntactic trees anymore, some subtrees are described by impure computation processes. (We haven't found a very good way to explain this yet, but see the rest of this document for more details.)

1.2 Relevant previous work

Fetscher, Claessen, Pałka, Hughes and Findler [2015] start from Redex, a Racket DSL to implement simple type systems, and derive a random program generator. One way to think of their approach is that they take each rule in the user-defined type system, and turn it into a small logic program that supports enumerating (or sampling) well-typed terms. For example the function-application rule in PLT Redex syntax

```
[(tc Γ e₁ (τ₂ → τ)) (tc Γ e₂ τ₂)
 ----------------------------------
 (tc Γ (e₁ e₂) τ)]
```

gets interpreted into a logic-program clause that could be described as follows in a Prolog-like syntax:

```
tc Γ e τ :-
  exists τ₂ e₁ e₂,
  e = (e₁ e₂),
  tc Γ e₁ (τ₂ → τ),
  tc Γ e₂ τ₂.
```

The main limitation of this work in our eyes is that Redex only works with fairly simple type system, in particular we do not expect it to scale to ML-style type inference with polymorphism, which is the part that hand-crafted generators also struggle on.

By "scaling" in this context we have two different qualities in mind:

(1) The approach should be expressive enough to capture interesting type-system features with acceptable (checking and) generation performance.
(2) The approach should be flexible and predictable enough that implementors of type-checkers for production compilers could be convinced to adopt it.

PLT Redex – and in general most declarative approaches to type-system definitions – is designed to be very convenient to express simple type systems, not to build production type-checkers for complex programming languages.

(Another brand of related work is the implementation of a type-checker directly in Prolog, and then using them for program enumeration/generation; we believe tha they could be pushed in interesting directions regarding aspect (1) above, but will still fail at (2).)

In constrast, our work builds on top of library-based constraint-solving approaches that are known to scale well to ML-style polymorphic type systems, and have already been put in production GHC.

1.3 Current status

We have a working prototype that demonstrates the interleaving of type inference and program generation, but only for a toy type system, namely the simply-typed lambda-calculus. Our prototype is a simplified re-implementation of the library Inferno [François Pottier 2022], extended with program generation capabilities. The extension with program generation is actually fairly minimal, it is not an invasive change.

We believe that the approach can be extended to support ML-style prenex polymorphism, but this requires more experimentation and validation work.

We found out with this experiment that it is difficult to obtain a generator with good performance, or rather that it is very easy to obtain a generator with bad performance. Subtle changes to the hole-filling strategy result in large qualitative differences, from generators that take seconds on very small terms sizes (5), to generators that appear to be exponential but scale to reasonably-sized terms (20 or so), to generators that can produce terms of arbitrary size but whose distribution we don't understand well.

It is too early to tell whether this issue of generation performance means that the idea will not work in practice on more complex type systems, or will remain a constant difficulty without preventing interesting applications, or can be solved with a crisp, good idea.

2 BACKGROUND: CONSTRAINT-BASED INFERENCE WITH ELABORATION

This section merely summarizes the key ideas of constraint-based inference in applicative style, as presented in Pottier [2014] and implemented in Inferno [François Pottier 2022] that paper (without any mention of ML-style polymorphism, which our prototype does not support yet). It does not contain new material. The software snippets we show are extracted for our own prototype.

2.1 Constraint-based type inference

Constraint-based type inference takes the program to be type-checked and generates a *constraint* from it, that is then passed to a constraint *solver*. Compared to previous approaches to type-inference that would proceed in one go by traversing the program, this allows to separate minute aspects of the programming language typing rules, encoded in the constraint generator, from the essential aspects of efficient type inference, encoded in the constraint language and solver.

For example, the input program `fun x -> (x + 1, x)` could generate the following constraint, with a free constraint variable a which represents the type of this program (if any):

$\exists a_1 a_2.$	fun x -> ...
$\quad a = a_1 \to a_2$	
$\quad \wedge\ \exists b_1 b_2.$	
$\quad\quad a_2 = b_1 * b_2$	(..., ...)
$\quad\quad \wedge\ b_1 = \mathsf{int} \wedge a_1 = \mathsf{int}$	(x + 1, ...)
$\quad\quad \wedge\ b_2 = a_1$	(..., x)

(The code in the column on the right represents the program fragment that generated the corresponding part of the constraint.)

This example constraint is expressed in the following constraint language:

$$
\begin{array}{rcl}
C & ::= & \\
& | & \mathsf{True} \\
& | & \mathsf{False} \\
& | & C_1 \wedge C_2 \\
& | & \exists a.\, C \\
& | & T_1 = T_2
\end{array}
$$

where T are "inference types", types that may contain inference variables.

A closed constraint C either has a solution or does not – in a sense, it may be evaluated to a boolean. For open constraints (which contain free type inference variables), the meaning of constraints is given by a two-place relation $\gamma \vDash C$, stating that the valuation 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)`.

Advanced type-system features can be expressed by adding more constraint-formers; in particular ML-style polymorphism can be expressed with let constraints that capture the infer-and-generalize approach of ML type inference.

Constraint generation. One way to express a constraint generator is as a function $\mathsf{gen}(E, t, a)$ 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:

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

2.2 Constraints with elaboration

Pottier [2014] remarks that this language of constraints C can be extended with a formal map operation – turning it into an applicative functor.

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

In this grammar, f must be a function in some ambient meta-language. The idea is that instead of corresponding to booleans (a constraint either is satisfiable or isn't), constraints now evaluates to arbitrary *values*; for example, the conjuction constraint $C_1 \wedge C_2$ now evaluate to the pair (V_1, V_2), if C_1 evaluates to V_1 and C_2 to V_2. The constraint $\mathsf{map}(C, f)$ evaluates C to a value V and returns the value $f(V)$. The constraint $\mathsf{witness}(a)$ returns the type τ inferred for the inference variable a in the final solution of the constraint – assuming that ground types τ can be expressed in our ambient meta-language. (If the meta-language is typed, it is natural to classify constraints by the type of their value.)

The meaning of constraints is now given by a three-place judgment $\gamma \vDash C \rightsquigarrow V$: under the valuation γ, the constraint C evaluates to the value V.

This extension captures elaboration: from a term in our source language, we can now generate a constraints that not only determines whether the term is typable, but evaluates to an explicitly-typed version of the source term. Authors of the type-checker can thus describe two things at once,

the type-checking rules for their language and the explicitly-typed representation of well-typed programs.

OCaml realization. This notion of constraints-with-values can be captured nicely as a GADT (guarded/generalized algebraic datatype), in OCaml syntax – so here the meta-language being used is OCaml:

```
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
```

Remember our constraint-generation rule for λ-abstractions:

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

Below is the OCaml implementation of type-inference for λ-abstractions in our software prototype. It corresponds to this constraint-generation rule, enriched with elaboration information:

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

Remark. `let+ p1 = e1 and+ p2 = e2 in body` is the OCaml syntax for applicative functors (our applicative-**do** notation), it desugars into `map (fun (p1, p2) -> body) (pair e1 e` for suitable applicative operations

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

2.3 Solving constraints

In research papers, inference constraint solvers are often formulated in small-step style, as formed of many independent rewriting rules that preserve the set of solutions and progress towards a normal form. Software implementations typically use a more big-step style, where the constraint is evaluated to a value in one recursive traversal.

The information in the constraint is injected into the state of the solver, which is mutated as the solving process evolves. There are two key data structures for the efficient implementation of ML type inference:

- Unification is implemented efficiently by a Union-Find graph, where nodes are types and inference variables. An equality constraint $T_1 = T_2$ is evaluated by unifying the two sides in the graph. (Our prototype implements this.)
- The key operation to implement prenex polymorphism is *generalization* at **`let`**-bindings, which is implemented by using *inference levels*, as described for example in Oleg Kiselyov's article How OCaml type checker works – or what polymorphism and garbage collection have in common. Inferno implements this, but our prototype does not.

The main algorithmic service provided to our solver is thus unification. It exposes the following API, where `state` is the type of the solver state:

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

The function `unify` unifies two types, updating the state in the process, or fails with an error if they are not unifiable. The function `equal` merely checks whether two types are already equal (without performing any unification).

On top of this unifier, we define an `eval` function for our constraints, whose type is slightly non-standard, it is designed to be reusable for program-generation purposes:

```
module Constraint : sig
  ...

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

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

`eval` takes a solver state and a constraint, and returns an updated state and a *residual constraint*, representing parts of the constraint to be solved later. With the constraint constructions that we have at this point, nothing prevents our solver from fully determining the result, so a `residual` constraint is morally a value-or-error: either the resolution succeeds, and it returns a function that will build a witness of type `'a` from the final solver state, or it fails with an error.

Notice that the argument of `Ret` has type `state -> 'a`, it is parametrized over the final solver state after solving the constraint, which we call the "solution" of the constraint. This access to the "final" state after all parts of the constraints have been resolved (not just the current subterm) is crucial to be able to evaluate `Witness`a constraints. The type inferred for the inference variable a may not be fully known at the point in the solving process where this sub-constraint is traversed.

It will become fully determined only later, as more unifications are done in other parts of the constraint.

A representative case of the solver definition is `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, Rel (fun sol -> (v sol, w sol))
```

We evaluate `c` and `d` to residual constraints. If either is an error, we fail with an error. Otherwise they return thunks of type `state -> 'a`, `state -> 'b`, and we build a pairing thunk `state -> ('a * 'b)`.

2.4 Putting it back together

We start from an `Untyped.t` source term, we use a constraint generator

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

to produce a constraint that produces a witness of type `Typed.t`, that is, a typed core term. We pass this constraint to our `eval` function

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

(along with some initial solver state), and we get a final state and either a `state -> Typed.t` value or an error. In the first case, passing the final state gives a `Typed.t` term, the result of running type-inference over our source program.

In code:

```
let typecheck (source_term : Untyped.t) : (Typed.t, error) result =
  let constr : Typed.t Constrain.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
```

3 ADDING PROGRAM GENERATION

3.1 Terms and constraints with subcomputations

We call *functor* a module `F` implementing the following signature:

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

(So we mean a category-theory-functor and not a ML-module-system-functor.)

Our untyped terms were previously defined in a module looking like this:

```
module Untyped = sig
  ...
  type t =
  | Var of Var.t
  | App of t * t
  | ...
end
```

To support program generation, we parametrize it over an arbitrary functor `F` (forming a ML-module-functor over category-functors), with a new `Do` constructor as follows:

```
module Untyped (F : Functor) = sig
  ...
  type t =
  | Var of Var.t
  | App of t * t
  | ...
  | Do of t F.t
end
```

We parametrize our type of constraints in this same way, with a similar-looking `Do` constructor:

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

Our intuition is that terms and constraints represent syntactic trees where some subtrees are in fact computations in some external notion of computation `'a F.t`. For example, if `F` is a non-determinism monad (`F` is `List`), these `Do` constructors let us describe untyped terms, or constraints, where one sub-tree is a non-deterministic enumeration of subtrees, possibly with nested computations.

Constraint generation can be trivially extended to support this new `Do` constructor:

```
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)
```

3.2 Constraint solving with subcomputations

We extend our type of residual constraints with a `Do` constructor, which contains a subcomputation in the functor `F` returning a constrait – a not-yet-solved constraint, not a residual constraint.

```
module Constraint (F : Functor) = sig
  ...
  type 'a residual =
  | Ret of (state -> 'a)
  | Fail of err
```

```
  | Do of 'a t F.t
end
```

Our constraint-solver will use `Do` residuals on each `Do` constructor in the input constraint. This means that all other cases will need to handle `Do` as a new possible residual. For example the case `Conj(c, d)` could become (the unchanged parts are grayed out):

```
  | 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, Rel (fun sol -> (v sol, w sol))
```

With this version, if `c` contains a `Do` subcomputation, then we do not evaluate `d` at all in the current state – this is arguably not the smallest "residual constraint" as it stops on the first undetermined term. It is possible to do better by evaluating both subterms in this case:

```
let residual_pair : 'a residual * 'b residual -> ('a * 'b) residual =
  function
  | Fail e, _ | _, Fail e -> Fail e
  | Do p, Do q -> Do (let+ c = p in Conj(c, Do q))
  | Do p, Ret w -> Do (let+ c = p in Conj (c, Ret w))
  | Ret v, Do q -> Do (let+ d = q in Conj (Ret w, d))
  | Ret v, Ret w -> Ret (fun sol -> (v sol, w sol))

let rec eval (st, c0) = match c0 with
| ...
| Conj (c, d) ->
  let (st, nc) = eval st c in
  begin match nc with
  | Fail e -> Fail e
  | _ ->
    let (st, nd) = eval st d in
    residual_pair nc nd
  end
```

3.3 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 a sequence of elements of type `'a` computed on demand. We can think of a value of type `'a m` as describing a search problem with zero, one or more solutions of type `'a`.

3.4 Iterated search

For any search monad `M`, our solver `eval` takes a `'a Constraint(M).t` and returns a `'a Constraint(M).t` which contains sub-computations of type `'a Constraint(M).t M.t`. We can "tie the knot" by iterating the solving process on the result of those sub-computations. This gives a function `solve` turning a `'a Constraint(M).t` into a `'a M.t`, parametrized by the state of the solver and a parameter `size:int` counting the total number of repeated expansions we allow.

```
let rec solve ~size state cstr =
  if size = 0 then M.sum [] (* fail *)
  let state, nf = eval state cstr in
  match nf with
  | Fail -> M.sum []
  | Ret v -> M.return (v state)
  | Do p -> M.bind (solve ~size:(size - 1) state) p
```

In fact our implementation is designed to return terms of size *exactly* `size`, rather than of size at most `size`, so our `Ret v` case is: `if size = 1 then M.return (v state) else M.sum []`.

3.5 Generic untyped and typed terms

For any search monad `M`, we can define a "generic" `Untyped(M).t` term that describes all possible untyped terms, by listing all possible term-formers under a `Do` constructor.

```
let rec gen (ctx : Untyped(M).Var.t list) : Untyped(M).t =
  Do (M.sum [
    (let+ x = of_list ctx in Var x);
    (let x = Untyped(M).Var.fresh ctx in
     let+ t = gen (x :: ctx) in
     Lam(x, t));
    (let+ t = gen ctx
     and+ u = gen ctx in
     M.return (App(t, u)));
  ])

let untyped_terms : Untyped(M).t = gen []
```

Calling the function `solve ~size` on this generic untyped term gives a generic typed term, whose solutions are all well-typed terms of size at most `size`.

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

At this point we are done! If we instantiate `M` with a random-sampling monad, we will get a random sampler of well-typed terms. A different search monad would give exhaustive enumeration, etc.

3.6 Summary

Our approach to interleave program generation and constraint-solving is to add a constructor in our type of source programs and constraints that allows search sub-computations.

This is much lighter than using holes / metavariables, as we do not need any specific mechanism to link "term holes" with "constraint holes" and figure out how to expand/refine constraint holes when the term holes are refined. The connection between `Do` computations in terms and `Do` computations in constraints is just a `F.map` call.

We initially expected to have to abstract terms and constraints over search monads `M` or at least applicative functors, but for now it seems that just abstracting terms and constraints over any functor `F` is enough. The only part of the work that needs to know about the search monad structure is the `solve` function that is defined outside/separately.

There would be another way to write the `eval` function on constraints that would benefit from `F` being an applicative functor: in the `Conj(c, d)` case, if both `c` and `d` evaluate to a `Do` subcomputation, we could take the applicative product of the two computations to return a single `Do` result – instead of our approach which still keeps two `Do` node, one nested inside the other. (Technically this relies on the fact that we consider *strong* functors.) We believe that this would not be better, but in fact worse: our `solve` function relies on the fact that it can count the number of `Do` node that it fills as a reasonable measure for the amount of work performed, and this approach of pairing `Do` nodes together in a single residual `Do` node breaks this assumption, it leads to a combinatorial explosion in searched term sizes that destroys sampling performance.

4 SOFTWARE PROTOTYPE

Our software prototype is available at https://gitlab.com/gasche/constraints-for-random-generation.

It implements constraint-based generation of well-typed terms in a simply-typed lambda-calculus with functions and pairs. (Fun fact: we originally wrote this implementation for the purpose of turning it into a programming project for a master-level class: we removed the essential parts mentioned in this report and asked students to reimplement them.)

We instantiate the generator with two different monads, one that enumerates all solutions and one that randomly samples them. Generation speed appears to be exponential, it is only practical to generate terms up to size 15-20. It may be possible to improve our generator, or the implementation of the search monads, to improve generation performance.

For example, here is how to request a random term of size 12, with a fixed seed 42 for reproductibility (... with exactly the same version as we used, commit 7e27f4239f):

```
$ dune exec -- minigen --size 12 --seed 42
lambda
(y/6 : γ/41e).
  lambda
  (y/20 : {α/41e * β/41e}).
    let
    ((z/602 : α/41e), (u/602 : β/41e))
    =
    y/20
    in
      (
        let (v/602 : α/41e) = z/602 in y/6,
        let (w/602 : α/41e) = z/602 in w/602
```

```
    )
```

`v/602` is a term variable, which is explicitly bound: it starts with a human-readable name, following by a unique stamp in hexadecimal. `γ/41e` is the name of a type variable, which is implicitly universally quantified at the begining of each term. It is apparent from this example that the generated terms are hard to read, and that our pretty-printer could be improved.

With the particular type theory that we currently implement, there are only 10 different terms of size 5, as can be checked with the `--exhaustive` option to enumerate the first N well-typed terms. Even if we give N=100, the enumeration is exhausted after 10 terms.

```
$ dune exec -- minigen --exhaustive --size 5 --count 100
lambda (v/4e : δ/e). lambda (u/65 : γ/e). lambda (z/6a : β/e). v/4e

lambda (v/4e : α/f). lambda (u/65 : δ/e). lambda (z/6a : β/e). u/65

lambda (v/4e : α/f). lambda (u/65 : γ/e). lambda (z/6a : δ/e). z/6a

lambda (v/4e : α/b). let (z/7e : α/b) = v/4e in v/4e

lambda (v/4e : α/b). let (z/7e : α/b) = v/4e in z/7e

lambda (v/4e : β/12). (v/4e, v/4e)

lambda
(v/4e : α/b). let ((y/b5 : δ/14), (z/b5 : γ/14)) = v/4e in v/4e

lambda
(v/4e : {α/b * γ/14}).
  let ((y/b5 : α/b), (z/b5 : γ/14)) = v/4e in y/b5

lambda
(v/4e : {α/b * γ/14}).
  let ((y/b5 : δ/14), (z/b5 : α/b)) = v/4e in z/b5

let (u/cb : g) = lambda (v/db : α/1a). v/db in u/cb
```

4.1 Performance

With the naive random-generation monad that we implemented, the performance of random generation is exponential in the `size` parameter. (There is a lot of noise in the result as different sampling choices can lead to very different generation times.)

The following benchmark suggests that, for the seed `42`, term generation is instantaneous up to size 10, and unreasonably slow at size 20 (15s to generate a term):

```
$ dune exec -- \
  hyperfine -P size 10 20 "minigen --size {size} --seed 42 > /dev/null"
Benchmark 1: minigen --size 10 --seed 42 > /dev/null
  Time (mean ± σ):      54.7 ms ±   1.5 ms    [User: 48.9 ms, System: 5.3 ms]
  Range (min … max):    51.3 ms …  59.8 ms    53 runs

Benchmark 2: minigen --size 11 --seed 42 > /dev/null
  Time (mean ± σ):      17.6 ms ±   1.3 ms    [User: 13.7 ms, System: 3.6 ms]
  Range (min … max):    15.5 ms …  22.0 ms    143 runs

Benchmark 3: minigen --size 12 --seed 42 > /dev/null
  Time (mean ± σ):      45.0 ms ±   1.7 ms    [User: 39.6 ms, System: 4.9 ms]
  Range (min … max):    41.4 ms …  49.2 ms    65 runs
```

```
Benchmark 4: minigen --size 13 --seed 42 > /dev/null
  Time (mean ± σ):     661.5 ms ±  27.6 ms    [User: 629.6 ms, System: 26.9 ms]
  Range (min … max):   634.0 ms … 710.0 ms    10 runs

Benchmark 5: minigen --size 14 --seed 42 > /dev/null
  Time (mean ± σ):     619.7 ms ±  11.4 ms    [User: 588.4 ms, System: 26.7 ms]
  Range (min … max):   602.9 ms … 638.5 ms    10 runs

Benchmark 6: minigen --size 15 --seed 42 > /dev/null
  Time (mean ± σ):      56.4 ms ±   2.8 ms    [User: 50.6 ms, System: 5.2 ms]
  Range (min … max):    50.7 ms …  68.3 ms    54 runs

Benchmark 7: minigen --size 16 --seed 42 > /dev/null
  Time (mean ± σ):      2.154 s ±  0.034 s    [User: 2.068 s, System: 0.072 s]
  Range (min … max):    2.128 s …  2.217 s    10 runs

Benchmark 8: minigen --size 17 --seed 42 > /dev/null
  Time (mean ± σ):     805.2 ms ±   8.5 ms    [User: 768.4 ms, System: 32.0 ms]
  Range (min … max):   795.4 ms … 821.7 ms    10 runs

Benchmark 9: minigen --size 18 --seed 42 > /dev/null
  Time (mean ± σ):      5.124 s ±  0.163 s    [User: 4.920 s, System: 0.170 s]
  Range (min … max):    4.914 s …  5.349 s    10 runs

Benchmark 10: minigen --size 19 --seed 42 > /dev/null
  Time (mean ± σ):      9.540 s ±  0.172 s    [User: 9.181 s, System: 0.296 s]
  Range (min … max):    9.371 s …  9.872 s    10 runs

Benchmark 11: minigen --size 20 --seed 42 > /dev/null
  Time (mean ± σ):     15.961 s ±  0.304 s    [User: 15.407 s, System: 0.450 s]
  Range (min … max):   15.547 s … 16.451 s    10 runs

Summary
  minigen --size 11 --seed 42 > /dev/null ran
    2.56 ± 0.22 times faster than minigen --size 12 --seed 42 > /dev/null
    3.12 ± 0.25 times faster than minigen --size 10 --seed 42 > /dev/null
    3.21 ± 0.29 times faster than minigen --size 15 --seed 42 > /dev/null
   35.31 ± 2.73 times faster than minigen --size 14 --seed 42 > /dev/null
   37.69 ± 3.24 times faster than minigen --size 13 --seed 42 > /dev/null
   45.88 ± 3.49 times faster than minigen --size 17 --seed 42 > /dev/null
  122.72 ± 9.43 times faster than minigen --size 16 --seed 42 > /dev/null
  291.93 ± 23.85 times faster than minigen --size 18 --seed 42 > /dev/null
  543.52 ± 42.05 times faster than minigen --size 19 --seed 42 > /dev/null
  909.40 ± 70.59 times faster than minigen --size 20 --seed 42 > /dev/null
```

The timings reported above correspond to sampling just one random term. The compute cost is not linear in the number of terms generated, it appears sub-linear; for example for size 16, sampling one term takes 2.1s, but sampling 10 terms only takes 8.0s.

The performance of our current software prototype is disappointing for two reasons:

- The use-case of metaprogram testing requires fast generation performance, to find a bug we need to generate many random terms per second. The program generator should be at least as fast as the metaprogram being tested, hopefully even faster to not be a bottleneck.
- We can only expect that generation time will become slower as we add more complex type-system features such as ML-style prenex polymorphism.

On the other hand, we have not worked on generation performance at all yet, so it may be that a clever idea in the random monad implementation or term generator is enough to massively improve performance. (We don't know if we should expect it to remain exponential.)

REFERENCES

Burke Fetscher, Koen Claessen, Michał Pałka, John Hughes, and Robert Bruce Findler. 2015. Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System. In *ESOP*, Jan Vitek (Ed.).

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

Michal H. Palka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an optimising compiler by generating random lambda terms. In *International Conference/Workshop on Automation of Software Test*.

François Pottier. 2014. Hindley-Milner elaboration in applicative style. In *ICFP*. https://doi.org/10.1145/2628136.2628145

François Pottier and Didier Rémy. 2005. The Essence of ML Type Inference. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). MIT Press, Chapter 10, 389–489. http://cambium.inria.fr/~fpottier/publis/emlti-final.pdf A draft extended version is also available.