# Constrained generation of well-typed program

**Work in progress** **June 23, 2025**

HUGO BARREIRO, École Polytechnique, France
GABRIEL SCHERER, INRIA, France

## 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!

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.

### 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 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.

### 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:

Authors' addresses: Hugo Barreiro, École Polytechnique, France; Gabriel Scherer, INRIA, France.

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

We are currently working on extending the prototype to handle Hindley-Damas-Milner type inference – to a toy ML language. This work is not finished (there remain a few thorny bugs to hunt) but in an advanced enough state to asses that this approach appears to scale to this type system – it can deal with this extra complexity, at acceptable levels of performance.

The mechanism of local *generalisation* that is distinctive of ML inference requires a careful handling of the scope of infereence variables. (In the simply-typed case it is always valid to lift all variables to the global scope.) This required changing our implementation of constraint-solving to use an explicit syntax of *continuations*, as in pen-and-paper presentations of constraint-solving but unlike typical implementations that use a mutable imperative datastructure to represent the constraint-solver state.

## 2 LOOK, NO HOLES!

We have found that it is *not* necessary to introduce a notion of holes in our syntax of terms and constraints to interleave term generation and constraint-solving. This can be done in a simpler, more abstract way by injecting a constructor for a search monad – our terms and constraints are not pure syntactic trees anymore, some subtrees are described by impure computation processes. In short, we call this *effectful syntax*. In OCaml code, this looks as follows:

```
module Untyped(M : MonadPlus) = struct
  type t =
  | Var : var -> t
  | ...
  | Do : t M.t -> t
end
module Constraint(M : MonadPlus) = struct
  type 'a t =
  | True : unit t
  | False : 'a t
  | Conj : 'a t * 'b t -> ('a * 'b) t
```

```
  | Map : 'a t * ('a -> 'b) -> 'b t
  | ...
  | Do : 'a t M.t -> 'a t
end
```

(Notice that our type `'a Constraint.t` is a GADT that is parametrized over the type `'a` of "elaboration witnesses" that the constraint produces when it is satisfiable. Our constraint generator takes a term of type `Untyped.t` and generates a `Typed.t Constraint.t`, which produces an explicit elaboration of the input term on success. This approach comes from Inferno.)

Our types of terms and constraints are parametrized over a search monad `M : MonadPlus`, and they both contain a constructor `Do` that injects an effectful term (or constraint) into terms (or constraints).

Our constraint generator will generate a `Do` constraint on every `Do` node of the input term:

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

Our constraint solver then takes a constraint that contains effectful syntax, and returns a result in the search monad:

```
module Solve(M : MonadPlus) = struct
  val solve : env -> 'a Constraint.t -> 'a M.t
end
```

## 3 PROGRAM GENERATION

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 = M.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 App(t, u));
    (let x = Untyped(M).Var.fresh ctx in
     let+ t = gen ctx and+ u = gen (x :: ctx) in Lam(x, t, u));
  ])
let untyped_terms : Untyped(M).t = gen []
```

We also implement a `cut_size` function that filters out the generic term to keep only terms of a certain size. On unary constructs it reduces size by one, on binary constructs it considers all possible combinations of sizes of its two subterms. A subterm that would go above the size budget is replaced by `Do M.fail`.

```
val cut_size : size:int -> Untyped(M).t -> Untyped(M).t
let sized_terms ~size = cut_size ~size untyped_terms
```

## 4 EVERYTHING TOGETHER

Our program generator uses `sized_terms` to describe all untyped terms of a given size, then the constraint generator `infer` function to turn it into an inference constraint, then the constraint solver `solve` tu turn it into a search problem for well-typed terms.

$$\texttt{size} : \texttt{int} \xrightarrow{\texttt{sized\_terms}} \texttt{Untyped(M).t} \xrightarrow{\texttt{infer}} \texttt{Typed.t Constraint(M).t} \xrightarrow{\texttt{solve}} \texttt{Typed.t M.t}$$

This is all parametric over the implementation of the search monad `M`. We provide two implementations: a monad that exhaustively enumerates all possible terms in order, and a monad that performs random sampling of the search space. Exhaustive enumeration is useful to test our generator, but unusable at large term sizes. Random sampling has to be done carefully to have reasonable performance. Our first naive implementation was exponential due to frequent backtracking, and struggled to generate terms of

size 15 or more. We explored various performance improvements, and our current generator is able to produce terms of size 100 instantly (< 100ms), and terms of size 1000 in a few seconds.

## 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.