Generating Well-Typed Terms that are not "Useless"

JUSTIN FRANK, University of Maryland, USA
BENJAMIN QUIRING, University of Maryland, USA
LEONIDAS LAMPROPOULOS, University of Maryland, USA

Random generation of well-typed terms lies at the core of effective random testing of compilers for functional languages. Existing techniques have had success following a top-down type-oriented approach to generation that makes choices locally, which suffers from an inherent limitation: the type of an expression is often generated independently from the expression itself. Such generation frequently yields functions with argument types that cannot be used to produce a result in a meaningful way, leaving those arguments unused. Such "use-less" functions can hinder both performance, as the argument generation code is dead but still needs to be compiled, and effectiveness, as a lot of interesting optimizations are tested less frequently.

In this paper, we introduce a novel algorithm that is significantly more effective at generating functions that use their arguments. We formalize both the "local" and the "nonlocal" algorithms as step-relations in an extension of the simply-typed lambda calculus with type and arguments holes, showing how delaying the generation of types for subexpressions by allowing nonlocal generation steps leads to "useful" functions. We implement our algorithm demonstrating that it's much closer to real programs in terms of argument usage rate, and we replicate a case study from the literature that finds bugs in the strictness analyzer of GHC, with our approach finding bugs four times faster than the current state-of-the-art local approach.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: test generation, property-based testing, well-typed lambda terms

ACM Reference Format:
Justin Frank, Benjamin Quiring, and Leonidas Lampropoulos. 2022. Generating Well-Typed Terms that are not "Useless". *Proc. ACM Program. Lang.* 1, CONF, Article 1 (January 2022), 21 pages.

1 INTRODUCTION

Random generation of programs is a tried and trusted technique for finding bugs in optimizing compilers, as exemplified in the CSmith project (Yang et al. 2011), which uncovered hundreds of bugs in widely used C compilers. At the same time, effective and efficient random generation of programs is a highly involved process, as most compilers impose both syntactic and semantic constraints on inputs before producing meaningful output that can be examined to discover potential errors. For example, the CSmith authors went to great lengths to ensure that the generated C programs did not exhibit undefined behavior.

In a functional setting, constrained program generation takes the form of generating well-typed lambda terms, with Pałka et al. (2011) setting the bar by finding bugs in GHC's strictness analyzer, Midtgaard et al. (2017) extending the same approach to handle effects to target OCaml, and Hoang et al. (2022) further building on top of it to target a higher-order blockchain language based on System F. In their seminal work, Pałka et al. introduce an efficient local method of generating well-typed STLC terms by inverting the typing rules and viewing them as production ones. For concreteness, consider the standard typing rule for applications in STLC:

$$\textsc{TApp}\ \frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash (e_1\ e_2) : \tau}$$

Authors' addresses: Justin Frank, jpfrank@umd.edu, University of Maryland, USA; Benjamin Quiring, bquiring@umd.edu, University of Maryland, USA; Leonidas Lampropoulos, leonidas@umd.edu, University of Maryland, USA.

2022. 2475-1421/2022/1-ART1 $15.00
https://doi.org/

Using the standard typing rule interpretation, TApp states that an application $(e_1\ e_2)$ has type τ if e_1 has type $\tau' \to \tau$ and e_2 has type τ'. Instead, Pałka et al. proposed an alternative reading of this rule where the conclusion prescribes the shape of a well-typed term and its premises give rise to recursive calls for generating appropriately typed subexpressions. Viewing TApp in this way, in order to generate an expression given some type τ, we can opt to generate an application $(e_1\ e_2)$. In turn, we would need to recursively generate a function e_1 of type $\tau' \to \tau$ and an argument for it e_2 of type τ' to satisfy the rule's premises.

While this recursive approach is elegantly compositional and relatively straightforward to implement, it suffers from a significant limitation which manifests in the TApp rule: the type τ' is completely arbitrary. That means that in order to recursively generate e_1 and e_2, we must first generate τ'. Unfortunately, as Pałka et al. point out, many choices for τ' can back the generation algorithm into a corner, which it can only get out of by using backtracking, significantly impacting the performance of this approach. To tackle this problem, the authors invented a series of heuristics to improve their proposed top-down generation algorithm. For instance, a particularly effective one was to pick the type τ' so that it matches the argument type of a variable in the environment.

However, the inherent limitation still lurks: *the generation of the type of an expression is independent from the generation of the expression itself.*

That is the root cause behind the excessive backtracking which could cause top-down generators to be potentially *inefficient*. Additionally, this limitation also causes top-down generators to be potentially *ineffective*: decoupling the generation of types and expressions significantly biases generation towards functions that *do not use their arguments*. Such "use-less" function arguments lead to wasted computation, both in terms of generation time (as the code that generates such arguments is essentially dead) and in terms of compilation time (as the test runner now needs to compile a larger program).

In this paper, we propose a new way of generating well-typed lambda terms that is guided by creating variable uses and bindings at the same time. We achieve that by deferring generation of types until they are actually needed, borrowing a page from many a playbook in the random testing literature that rely on laziness to improve test generation (Lampropoulos et al. 2017a; Reich et al. 2012; Runciman et al. 2008).

Concretely, our contributions are:

- We formalize the algorithm of Pałka et al. (2011) by extending STLC to a calculus with typed holes (called $\lambda_{\square}$), in which their top-down algorithm takes the form of a big-step relation, rendering the limitation in the generation algorithm glaringly apparent (Section 2).
- We introduce a novel algorithm for generating well-typed lambda terms, that is able to dynamically extend function types during generation. We formalize it by further extending $\lambda_{\square}$ to a calculus with arguments holes (called $\lambda_{\triangleright}$), in which our lazy generation algorithm manifests as a small-step relation, allowing for more fine-grained control of the resulting distribution (Section 3).
- We mechanized the metatheory of $\lambda_{\triangleright}$ in Coq and prove type soundness of our generation rules.
- We describe an efficient implementation of our algorithm in OCaml (Section 3.4), show that it is 4 times faster at finding bugs than Pałka et al. (2011) (Section 4.1), and demonstrate empirically that the terms generated by our algorithm have fewer unused parameters, as do real programs of similar size (Section 4.2).

We extensively discuss related work in Section 5 and conclude by drawing directions for future work in Section 6.

2 THE "LOCAL" GENERATOR

In this section, we will formalize the type-directed generation approach of Pałka et al. (2011) by introducing a simply-typed lambda calculus with typed holes. We will explicitly lay out the production rules that are obtained by inverting the typing relation in the style of Pałka et al., which serves both to establish the limitation of the original algorithm that makes generating well-typed terms that use their arguments less likely and as a platform to build upon when introducing our own variant of well-typed generation in the rest of the paper.

2.1 STLC with typed holes: $\lambda_\square$

Expressions e in the simply-typed lambda calculus with typed holes (which we will write as $\lambda_\square$) are either one of the standard expressions for an STLC with multiple arguments (function calls, λ expressions, variables, or constants of some base type τ_b) or a *typed hole* $\square_\tau$. The point of such a hole is to be filled with with an expression of its corresponding type τ, and we will use them to explicitly formulate program generation.

$$
\begin{array}{rcll}
e & = & e\ e\ \ldots & \text{Func. Applications} \\
 & | & \lambda\ (x : \tau\ \ldots)\ e & \lambda \text{ abstractions} \\
 & | & x & \text{Variables} \\
 & | & c & \text{Constants} \\
 & | & \square_\tau & \text{Typed holes} \\
\tau & = & \tau_b & \text{Base types} \\
 & | & \tau\ \ldots \rightarrow \tau & \text{Function Types}
\end{array}
$$

Associated with the expression language are the following typing derivation rules, where environments Γ are partial mappings from variables to types as usual:

$$
\textsc{TApp}\ \frac{\Gamma \vdash e_f : \tau_1\ \ldots\ \tau_n \rightarrow \tau \quad \Gamma \vdash e_1 : \tau_1\ \ldots\ \Gamma \vdash e_n : \tau_n}{\Gamma \vdash (e_f\ e_1\ \ldots\ e_n) : \tau} \qquad \textsc{TVar}\ \frac{[x \mapsto \tau] \in \Gamma}{\Gamma \vdash x : \tau}
$$

$$
\textsc{TAbs}\ \frac{\Gamma[x_1 \mapsto \tau_1\ \ldots\ x_n \mapsto \tau_n] \vdash e : \tau}{\Gamma \vdash \lambda\ (x_1 : \tau_1\ \ldots\ x_n : \tau_n)\ e : \tau_1\ \ldots\ \tau_n\ \rightarrow \tau} \qquad \textsc{TConst}\ \frac{c \text{ is a constant of type } \tau}{\Gamma \vdash c : \tau} \qquad \textsc{THole}\ \frac{}{\Gamma \vdash \square_\tau : \tau}
$$

All rules are standard with the exception of the typing rule for holes: a hole $\square_\tau$ has type τ under any environment.

2.2 Term Generation as a Step Relation.

Random generation in the style of Pałka et al. can be encoded in $\lambda_\square$ as a big-step reduction relation, that instantiates typed holes. Each typing rule with the exception of THole gives rise to a corresponding production rule of the form $\square_\tau \Downarrow_\Gamma e$, where each production rule is annotated by an enclosing environment Γ.

$$\textsc{GVar}\ \frac{[x \mapsto \tau] \in \Gamma}{\Box_\tau \Downarrow_\Gamma x} \qquad \textsc{GConst}\ \frac{\emptyset \vdash c : \tau}{\Box_\tau \Downarrow_\Gamma c}$$

$$\textsc{GAbs}\ \frac{\Box_\tau \Downarrow_{\Gamma[x_1 \mapsto \tau_1 \ldots x_n \mapsto \tau_n]} e}{\Box_{\tau_1 \ldots \tau_n \to \tau} \Downarrow_\Gamma \lambda\ (x_1 : \tau_1 \ldots x_n : \tau_n)\ e}$$

$$\textsc{GApp}\ \frac{\Box_{\tau_1 \ldots \tau_n \to \tau} \Downarrow_\Gamma e \quad \Box_{\tau_1} \Downarrow_\Gamma e_1 \ldots \Box_{\tau_n} \Downarrow_\Gamma e_n}{\Box_\tau \Downarrow_\Gamma e\ e_1 \ldots e_n}$$

We have four basic choices: we can completely fill the hole with a correctly typed variable or constant; if we are trying to fill a hole of type $\tau_1 \ldots \tau_n \to \tau$, then we can construct a λ term that takes *fresh* variables of types $\tau_1 \ldots \tau_n$ whose body is recursively generated from a hole of type τ; or we can create an application comprised of terms generated by a hole for the function and holes for the arguments for some arbitrary argument types $\tau_1 \ldots \tau_n$.

These big-step rules are fairly straightforward to implement: we can pick a production rule, recursively satisfy its premises, and combine any results as needed. However, they are also quite coarse: in the application rule, we have to fully generate either the function or an argument before generating the rest. At this point it is natural to wonder if there is a small-step equivalent that allows for finer control. The answer, of course, is yes.

2.3 From big-step to small-step

The first step is to introduce *expression contexts*: the type of one-hole contexts C of expressions e.

$$\begin{array}{rcll} C & = & e \ldots C\ e \ldots & \text{Func. or Arg. contexts} \\ & | & \lambda\ (x : \tau \ldots)\ C & \lambda \text{ contexts} \\ & | & \Diamond & \text{Context holes} \end{array}$$

Contexts C always end in a single *context hole* $\Diamond$, which can be "filled in" by an expression, or another context. We write $C \blacklozenge e$ to mean filling in the context hole in C with the expression e. This operation is associative, so that $(C_1 \blacklozenge C_2) \blacklozenge e = C_1 \blacklozenge C_2 \blacklozenge e = C_1 \blacklozenge (C_2 \blacklozenge e)$. This notation is equivalent to the traditional "$C_1[C_2[e]]$" — we use it so that the small-step generation rules throughout the paper are easier to parse.

Given an expression e, we can decompose it into C and e' such that $e = C \blacklozenge e'$ to *focus* on the subexpression e' within e, which lives inside the context C. Note that when we decompose the term in this way the context C provides an environment of variables whose scope encompasses e' — if e is a closed term, then any free variable in e' comes from a binding provided in C. We define $\Gamma(C)$ to be this environment.

Armed with the notion of contexts, we can describe generation as a sequence of small-step reductions that focus on a specific typed hole. Once again, each typing rule of $\lambda_\Box$ gives rise to a production rule:

$$\textsc{GenVar}\quad C \blacklozenge \square_\tau \Longrightarrow C \blacklozenge x \quad \text{where } [x \mapsto \tau] \in \Gamma(C) \qquad \textsc{GenConst}\quad C \blacklozenge \square_\tau \Longrightarrow C \blacklozenge c \quad \text{where constant } c \text{ has type } \tau$$

$$\textsc{GenAbs}\quad C \blacklozenge \square_{\tau_1 \ldots \tau_n \to \tau} \Longrightarrow C \blacklozenge (\lambda\ (x_1 : \tau_1 \ldots x_n : \tau_n)\ \square_\tau) \quad \text{where } x_1 \ldots x_n \text{ are } \textit{fresh}$$

$$\textsc{GenApp}\quad C \blacklozenge \square_\tau \Longrightarrow C \blacklozenge (\square_{\tau_1 \ldots \tau_n \to \tau}\ \square_{\tau_1} \ldots \square_{\tau_n}) \quad \text{for any types } \tau_1 \ldots \tau_n$$

It can be easily verified that these rules are type-preserving as they share the same one-to-one correspondence with the typing rules that exists with the big-step generation rules. The notable difference is that the production rule corresponding to application GenApp now produces $n + 1$ typed holes, each of which we can choose to instantiate in any order. That allows us to *interleave* the generation of arguments and function expressions, which will prove crucial later on.

With this relation at hand, we can now formally define generation of a well-typed term e for a given type τ. We will say that an expression e is the result of such generation if e contains no holes, i.e. e cannot be stepped further, and can also be produced by the transitive closure of the step relation starting from a hole of type τ:

$$(\square_\tau \Longrightarrow^* e)$$

As well-typedness is preserved by the production rules and $\square_\tau$ is well-typed in an empty context, we can also conclude that e also has type τ in an empty context.

This small-step view of generation provides an alternative, but equally straightforward implementation: after choosing a transition we can fill in any new holes by recursive descent — we never have to look upwards into the context. Adding a decreasing *size* parameter to an implementation ensures termination: when the size becomes zero, remaining typed holes can be filled in with "trivial" terms such as `nil`, `1`, etc.. A hole with a function type can step to a λ with a body that is "trivial" in the same way, which structurally decreases the size of the type, even though it does not decrease the number of holes.

2.4 Extended Example

For concreteness, let's step through an example generation of a list of integers using all production rules. This example is shown in Figure 1.

(1) We begin with a typed hole of type `List Int` and an empty context. At this point, multiple rules apply: we could either fill the hole with a constant (such as the empty list) using GenConst or we could expand the hole with an application using GenApp. A generation strategy at this point would make a randomized choice according to some distribution.
(2) For the sake of the example, let's go with the more interesting choice of GenApp. Immediately, however, we're faced with yet another choice: how many arguments should application use and over which types? To keep things going, let's generate a single-argument application over lists of integers.
(3) We now have two holes — we need to focus on one to continue generation. Let's focus on the left.
(4) To expand the left hole that needs a function, we could choose to apply the GenAbs rule, creating a new hole for the body.

$$
\begin{array}{rcll}
\Diamond & \blacklozenge & \square_{\texttt{List Int}} & (1) \\
 & \Longrightarrow & \textsc{GenApp} & \\
\Diamond & \blacklozenge & \square_{\texttt{List Int}\to\texttt{List Int}}\square_{\texttt{List Int}} & (2) \\
 & = & (\textit{focus on left hole}) & \\
\Diamond\ \square_{\texttt{List Int}} & \blacklozenge & \square_{\texttt{List Int}\to\texttt{List Int}} & (3) \\
 & \Longrightarrow & \textsc{GenAbs} & \\
\Diamond\ \square_{\texttt{List Int}} & \blacklozenge & \lambda\ (x : \texttt{List Int})\ \square_{\texttt{List Int}} & (4) \\
 & = & (\textit{focus on inner hole}) & \\
(\lambda\ (x : \texttt{List Int})\ \Diamond)\ \square_{\texttt{List Int}} & \blacklozenge & \square_{\texttt{List Int}} & (5) \\
 & \Longrightarrow & \textsc{GenVar} & \\
(\lambda\ (x : \texttt{List Int})\ \Diamond)\ \square_{\texttt{List Int}} & \blacklozenge & x & (6) \\
 & = & (\textit{refocus on remaining hole}) & \\
(\lambda\ (x : \texttt{List Int})\ x)\ \Diamond & \blacklozenge & \square_{\texttt{List Int}} & (7) \\
 & \Longrightarrow & \textsc{GenConst} & \\
(\lambda\ (x : \texttt{List Int})\ x)\ \Diamond & \blacklozenge & \texttt{nil} & (8) \\
 & = & (\textit{remove focus}) & \\
\Diamond & \blacklozenge & (\lambda\ (x : \texttt{List Int})\ x)\ \texttt{nil} & (9)
\end{array}
$$

Fig. 1. Example derivation of a generation

(5) We then further focus on the body hole, transforming the context appropriately.
(6) We are once again faced with a typed hole of type `List Int`, as in the beginning, but this time the context contains a binding that is in scope: `x`. That means we can use GenVar.
(7) After completely filling the functional part of the application, we refocus on the remaining hole: the argument.
(8) For the third occurrence of a `List Int` hole we can pick to fill it with GenConst.
(9) Generation is now finished as no holes remain.

2.5 Limitations

This example renders the inherent limitation mentioned in the introduction glaringly obvious: to preserve the straightforward recursive nature of the algorithm, the types $\tau_1 \ldots \tau_n$ need to be generated before focusing on any of the newly generated holes. Unfortunately, a poor choice of types during this production rule can significantly impact the performance and effectiveness of the generation. For example, what if at step (2) we had picked a different type, such as `Int` or `Float`? In the former case, generating a body that uses the abstracted variable meaningfully would require some luck (e.g. choosing to generate an application of a cons cell whose head can be the variable). In the latter case, using the argument is near impossible. Pałka et al. (2011) attempt to address this problem by trying to make as educated a choice as possible during such productions.

The way they proposed to make a more educated choice was to introduce an additional production rule for application, that combined the GenApp and GenVar rules to make the use of existing functions (for example, primitives) more likely:

$$
\begin{array}{l}
\textsc{GenFunVar} \\
\quad C \blacklozenge \square_\tau \Longrightarrow C \blacklozenge (f\ \square_{\tau'}) \\
\text{where } [f \mapsto \tau' \to \tau] \in \Gamma(C)
\end{array}
$$

However, this rule still doesn't ensure that f will use it's arguments and therefore doesn't completely solve the problem; it only makes it more likely f itself will be used.

While in theory the effect of this rule could be achieved with just the basic four, observing this behavior in practice would be quite unlikely, as the generation would have to go through a very particular set of guesses to form the right type. In contrast, we propose an alternative generation strategy that *postpones making such a choice* until a use for the function argument is encountered.

3 THE "NONLOCAL" GENERATOR

The essence of the problem is that the local algorithm needs to guess a type that will eventually appear on a typed hole. We introduce a way for the algorithm to delay this choice. At a high-level, when generating a function f we can leave its parameters *unspecified*, and introduce a rule that adds a parameter to f of some type. Composing this rule with one that fills a hole with a variable, f is able to fill some hole $\square_\tau$ with a variable x and add $x : \tau$ as a parameter. This rule introduces a non-locality in the generation process that the local approach does not exhibit and can only be expressed when adopting a small-step view of generation.

Such a method does not come without its complications. For example, the function f may impose additional nonlocal constraints within the generated fragment, such as calls to f or passing f as a higher-order argument. As a result, any changes to the type of f need to be propagated to the type of these constrained locations.

3.1 Extending $\lambda_\square$

To capture these locations, we introduce the notion of *arguments holes* denoted as "$\triangleright$", which are labeled with globally scoped identifiers e.g. $\triangleright^\alpha$. Informally we use these arguments holes to link applications with all the lambdas that could be called from that site and vice-versa. To do this we extend the syntax of $\lambda_\square$ to add variants of each function type, application, and lambda parameter list that is annotated with an arguments hole. Finally, we modify the typing rules for applications and lambdas to require that the function type is annotated with the same arguments hole. This allows us to define an operation of "extending" an arguments hole $\triangleright^\alpha$ by extending every lambda, application, and function type annotated with the same $\triangleright^\alpha$ in the appropriate way. The supplemental material contains a Coq formalization of the following type system and a proof of soundness of arguments hole extension.

Formally, we first extend types to contain arguments holes, so that types are either a base type, a function type, or a function type whose list of argument types may be extended:

$$\begin{array}{rcll} \tau & = & \ldots & \\ & \mid & \tau \ \ldots \triangleright^\alpha \to \tau & \text{Extendable Function Types} \end{array}$$

We also update the definitions of terms and contexts:

$$\begin{array}{rcll} e & = & \ldots & \\ & \mid & e \ e \ \ldots \triangleright^\alpha & \text{Extendable Func. Applications} \\ & \mid & \lambda \ (x : \tau \ \ldots \triangleright^\alpha) \ e & \text{Extendable } \lambda \text{ abstractions} \end{array}$$

Just like function types, both function applications and lambda abstractions are now extendable. Similarly, the contexts corresponding to function application and lambda abstraction are also now extendable.

$$\begin{array}{rcll} C & = & \ldots & \\ & \mid & e \ \ldots \ C \ e \ \ldots \triangleright^\alpha & \text{Extendable Func. or Arg. contexts} \\ & \mid & \lambda \ (x : \tau \ \ldots \triangleright^\alpha) \ C & \text{Extendable } \lambda \text{ contexts} \end{array}$$

We call this language $\lambda_\triangleright$. Note that we use the same symbol $\triangleright$ in three distinct settings: type argument lists, function application argument lists, and lambda parameter lists. When any of these

are updated for some $\triangleright^\alpha$, all locations using the same $\triangleright^\alpha$ must be updated. We can prove this using the following updated typing relation, which simply checks that the arguments holes "line up" correctly.

$$\textsc{TApp}\triangleright\quad \frac{\Gamma \vdash e_f : \tau_1 \ldots \tau_n \triangleright^\alpha \to \tau \quad \Gamma \vdash e_1 : \tau_1 \ldots \Gamma \vdash e_n : \tau_n}{\Gamma \vdash (e_f\ e_1 \ldots e_n \triangleright^\alpha) : \tau}$$

$$\textsc{TAbs}\triangleright\quad \frac{\Gamma[x_1 \mapsto \tau_1 \ldots x_n \mapsto \tau_n] \vdash e : \tau}{\Gamma \vdash \lambda\ (x_1 : \tau_1 \ldots x_n : \tau_n \triangleright^\alpha)\ e : \tau_1 \ldots \tau_n \triangleright^\alpha \to \tau}$$

A function application $e_f\ e_1 \ldots e_n$ is extendable by an arguments hole $\triangleright^\alpha$ if the type of the function application e_f is also extendable by *the same* arguments hole. This restriction ensures that all lambdas that could be applied here have the same arguments hole, guaranteeing that extending $\triangleright^\alpha$ will not introduce any arity errors. On the flip side, an extendable lambda abstraction has an extendable function type with the same arguments hole, if we can type its body e in an environment extended by all the concrete bindings with appropriate types.

We note that $\lambda_\triangleright$ is truly an extension of $\lambda_\square$: we can convert any $\lambda_\triangleright$ term to a $\lambda_\square$ one by simply erasing all of the arguments holes in it — the resulting term is still well-typed. This means that we can guarantee that generators for $\lambda_\triangleright$ terminate: once the global program term gets large enough, we can erase all arguments holes and fill remaining typed holes in with "trivial" terms, as with generators for $\lambda_\square$.

3.2 Nonlocal Generation

In the following section, we'll explore the production rules involving arguments holes via extending the step relation ($\Longrightarrow$). Each of these rules maintain well-typedness in $\lambda_\triangleright$, which we have also formalized in the Coq model, proving that they maintain the well-typedness of the program.

To begin, the rule that introduces a new $\triangleright$ is the extendable function application.

$$\textsc{GenApp}\triangleright\quad C \blacklozenge \square_\tau \Longrightarrow C \blacklozenge (\square_{\triangleright^\alpha \to \tau}\ \triangleright^\alpha) \quad \text{where } \triangleright^\alpha \text{ is } \textit{fresh}$$

Just like the previous application rule GenApp, the new rule replaces a typed hole of type τ with an application of a typed hole whose result type is τ. However, unlike the previous application rule which placed no constraints on the type(s) of the argument, GenApp$\triangleright$ delays the choice of the type(s), using a fresh arguments hole $\triangleright^\alpha$ in its place. Eventually, this arguments hole will be extended, and the argument type(s) will be "filled in". At the same time, the argument to the application is an arguments hole of the same name, ensuring that the function arguments and the function type remain in sync. This rule satisfies TApp$\triangleright$ trivially.

The dual to this rule is the generation of extendable lambda abstractions:

$$\textsc{GenAbs}\triangleright\quad C \blacklozenge \square_{\tau_1 \ldots \tau_n \triangleright^\alpha \to \tau} \Longrightarrow C \blacklozenge \lambda\ (x_1 : \tau_1 \ldots x_n : \tau_n \triangleright^\alpha)\ \square_\tau \quad \text{where } x_1 \ldots x_n \text{ are } \textit{fresh}$$

This rule is a natural extension of the GenAbs rule to $\lambda_\triangleright$, adding multiple new bindings for the arguments already specified, and using the same arguments hole to ensure that any arguments that are added in the future will also be added as parameters to this lambda abstraction. Note that GenApp$\triangleright$ followed immediately by GenAbs$\triangleright$ will result in no new variables being introduced for the moment.

Similarly to Pałka et al. (2011) we introduce a rule for creating an application of an existing function:

$$\begin{array}{l}\textsc{GenFunVar}\triangleright\\ C \blacklozenge \square_\tau \Longrightarrow C \blacklozenge f\ \square_{\tau_1}\ \ldots\ \square_{\tau_n}\ \triangleright^\alpha\\ \text{where } [f \mapsto \tau_1\ \ldots\ \tau_n \triangleright^\alpha \to \tau] \in \Gamma(C)\end{array}$$

Unlike GenFunVar, this rule is **not** a combination of previous rules. Without this rule there would be no way to use variables of an extendable type since GenApp▷ requires that the arguments hole is *fresh*, and TApp▷ requires the same arguments hole for extendable applications to be well-typed.

Finally, we get to the crucial rule: generating a variable that does not appear concretely in the context, by extending an enclosing abstraction's parameter list:

$$\begin{array}{l}\textsc{GenParam}\triangleright\\ \begin{array}{lcllcll}C_1 & \blacklozenge & (\lambda\ (x_1 : \tau_1\ \ldots\ x_n : \tau_n\ \triangleright^\alpha)\ \Diamond) & \blacklozenge & C_2 & \blacklozenge & \square_\tau \quad \Longrightarrow\\ C_1\overline{\triangleright^\alpha + \tau} & \blacklozenge & (\lambda\ (x_1 : \tau_1\ \ldots\ x_n : \tau_n\ y : \tau\ \triangleright^\alpha)\ \Diamond) & \blacklozenge & C_2\overline{\triangleright^\alpha + \tau} & \blacklozenge & y,\end{array}\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{where } y \text{ is } \textit{fresh} \text{ and } \triangleright^\alpha \notin \tau\end{array}$$

This rule is our first example of a nonlocal generation rule. First, we decompose the context of the hole $\square_\tau$ that we want to fill in three parts: an outer context C_1, a lambda context that contains the parameter list (annotated with $\triangleright^\alpha$) to be extended, and an inner context C_2. The hole $\square_\tau$ is then filled with a fresh variable y; this variable is then used to extend the bindings of the lambda abstraction from $x_1 : \tau_1\ \ldots\ x_n : \tau_n\ \triangleright^\alpha$ to $x_1 : \tau_1\ \ldots\ x_n : \tau_n\ y : \tau\ \triangleright^\alpha$. However, since the arguments hole $\triangleright^\alpha$ is being extended, all locations in the global program term that use the same arguments hole must also be updated. We require that none of τ contains $\triangleright^\alpha$—otherwise, this update would never terminate as we would be creating a cyclic type. We write $\tau'\overline{\triangleright^\alpha + \tau}$, $e\overline{\triangleright^\alpha + \tau}$, and $C\overline{\triangleright^\alpha + \tau}$ for applying the extension $\overline{\triangleright^\alpha + \tau}$ of $\triangleright^\alpha$ by τ to types, expressions, and contexts, respectively. We will walk through each of these in detail below.

For concreteness, before formalizing this extension, let's step through an example of applying the GenParam▷ rule. Assume that generation has reached the following expression — a lambda abstraction that binds f of extendable type $\triangleright^\alpha \to \texttt{Int}$ and calls it, applied to a lambda abstraction whose parameter list contains the same arguments hole, and whose body is a typed hole:

$$(\lambda\ (f : \triangleright^\alpha \to \texttt{Int})\ (f\ \triangleright^\alpha))\ (\lambda\ (\ \triangleright^\alpha)\ \square_{\texttt{Int}})$$

This expression can be decomposed as follows, using the application that binds and applies f as the outer context, and an empty inner context:

$$C_1 \blacklozenge (\lambda\ (\ \triangleright^\alpha)\ \Diamond) \blacklozenge C_2 \blacklozenge \square_{\texttt{Int}} \quad \text{where} \quad \begin{array}{l}C_1 = ((\lambda\ (f : \triangleright^\alpha \to \texttt{Int})\ (f\ \triangleright^\alpha))\ \Diamond)\\ C_2 = \Diamond\end{array}$$

We can then apply GenParam▷ to fill the hole with a fresh variable x, that we bind in the lambda immediately above it.

$$\begin{array}{lcllcll}C_1 & \blacklozenge & (\lambda\ (\ \triangleright^\alpha)\ \Diamond) & \blacklozenge & C_2 & \blacklozenge & \square_{\texttt{Int}} \quad \Longrightarrow [\textsc{GenParam}\triangleright]\\ C_1\overline{\triangleright^\alpha + \texttt{Int}} & \blacklozenge & (\lambda\ (x : \texttt{Int}\ \triangleright^\alpha)\ \Diamond) & \blacklozenge & C_2\overline{\triangleright^\alpha + \texttt{Int}} & \blacklozenge & x\end{array}$$

At this point, to preserve well-typedness we need to also expand the arguments hole $\triangleright^\alpha$ in C_1 (and, if such a hole existed, also in C_2), using the extension metafunction on C_1. There are two occurrences of $\triangleright^\alpha$ in C_1 that need to be extended; the type of f and the application of f. For the former case we extend the type to include the added parameter type, and for the latter we place a hole of type τ in the application. Performing those extensions gives us:

$$C_1\overline{\triangleright^\alpha + \texttt{Int}} = ((\lambda\ (f : \texttt{Int}\ \triangleright^\alpha \to \texttt{Int})\ (f\ \square_{\texttt{Int}}\ \triangleright^\alpha))\ \Diamond)$$

Since there are no occurrences of $\triangleright^\alpha$ in C_2, we see that $C_2\overline{\triangleright^\alpha + \texttt{Int}} = C_2 = \Diamond$. Plugging the context holes gives us the resulting expression from the application of GenParam▷:

$$\begin{array}{l} (\lambda\ (f : \triangleright^\alpha \to \texttt{Int})\ (f\ \triangleright^\alpha))\ (\lambda\ (\ \triangleright^\alpha)\ \square_{\texttt{Int}}) \\ (\lambda\ (f : \texttt{Int}\ \triangleright^\alpha \to \texttt{Int})\ (f\ \square_{\texttt{Int}}\ \triangleright^\alpha))\ (\lambda\ (x : \texttt{Int}\ \triangleright^\alpha)\ x) \end{array} \quad \Longrightarrow [\textsc{GenParam▷}]$$

Extension metafunctions, formally. The extension metafunctions need to be defined over types, expressions, and contexts. For types, the only interesting behavior occurs when an arguments hole appears in both a type and in its extension. In this case, the type of the extension is added to the arguments position of the type, just like we did for τ above. [1]

Types

$$\begin{array}{lcll} (\tau_1 \ldots \triangleright^\alpha \to \tau_2)\overline{\triangleright^\alpha + \tau} & = & \tau_1\overline{\triangleright^\alpha + \tau} \ldots \boldsymbol{\tau}\ \triangleright^\alpha \to \tau_2\overline{\triangleright^\alpha + \tau} & \\ (\tau_1 \ldots \triangleright^\beta \to \tau_2)\overline{\triangleright^\alpha + \tau} & = & \tau_1\overline{\triangleright^\alpha + \tau} \ldots \triangleright^\beta \to \tau_2\overline{\triangleright^\alpha + \tau} & \text{where } \triangleright^\alpha \neq \triangleright^\beta \\ (\tau_1 \ldots \to \tau_2)\overline{\triangleright^\alpha + \tau} & = & \tau_1\overline{\triangleright^\alpha + \tau} \ldots \to \tau_2\overline{\triangleright^\alpha + \tau} & \\ \tau_b\overline{\triangleright^\alpha + \tau} & = & \tau_b & \end{array}$$

For expressions, the first interesting case is for extendable applications where the arguments hole matches that of the extension. In this case, a hole with the type of the extension is added onto the arguments of the application, just like when we added a typed hole argument to the application of f in the previous example. The second interesting case is for extendable lambdas whose arguments hole matches that of the extension. In this case we must extend the parameter list of the lambda with a new variable. This rule is what makes this method *almost* not "useless": in this one case we are forced to add a variable which we cannot guarantee will be used, since its creation did not originate from the lambda's body.

Expressions

$$\begin{array}{lcll} (e_1\ e \ldots \triangleright^\alpha)\overline{\triangleright^\alpha + \tau} & = & e_1\overline{\triangleright^\alpha + \tau}\ e\overline{\triangleright^\alpha + \tau} \ldots \boldsymbol{\square_\tau}\ \triangleright^\alpha & \\ (e_1\ e \ldots \triangleright^\beta)\overline{\triangleright^\alpha + \tau} & = & e_1\overline{\triangleright^\alpha + \tau}\ e\overline{\triangleright^\alpha + \tau} \ldots \triangleright^\beta & \text{where } \triangleright^\alpha \neq \triangleright^\beta \\ (e_1\ e \ldots)\overline{\triangleright^\alpha + \tau} & = & e_1\overline{\triangleright^\alpha + \tau}\ e\overline{\triangleright^\alpha + \tau} \ldots & \\ (\lambda\ (x : \tau_1 \ldots \triangleright^\alpha)\ e)\overline{\triangleright^\alpha + \tau} & = & \lambda\ (x : \tau_1\overline{\triangleright^\alpha + \tau} \ldots \boldsymbol{y : \tau}\ \triangleright^\alpha)\ e\overline{\triangleright^\alpha + \tau} & \text{where } y \text{ is } \textit{fresh} \\ (\lambda\ (x : \tau_1 \ldots \triangleright^\beta)\ e)\overline{\triangleright^\alpha + \tau} & = & \lambda\ (x : \tau_1\overline{\triangleright^\alpha + \tau} \ldots \triangleright^\beta)\ e\overline{\triangleright^\alpha + \tau} & \text{where } \triangleright^\alpha \neq \triangleright^\beta \\ (\lambda\ (x : \tau_1 \ldots)\ e)\overline{\triangleright^\alpha + \tau} & = & \lambda\ (x : \tau_1\overline{\triangleright^\alpha + \tau} \ldots)\ e\overline{\triangleright^\alpha + \tau} & \\ x\overline{\triangleright^\alpha + \tau} & = & x & \\ \square_{\tau_1}\overline{\triangleright^\alpha + \tau} & = & \square_{\tau_1\overline{\triangleright^\alpha + \tau}} & \end{array}$$

Finally, extending arguments holes in contexts behaves the same ways as in expressions.

Contexts

$$\begin{array}{lcll} (e_1 \ldots C\ e_2 \ldots \triangleright^\alpha)\overline{\triangleright^\alpha + \tau} & = & e_1\overline{\triangleright^\alpha + \tau} \ldots C\overline{\triangleright^\alpha + \tau}\ e_2\overline{\triangleright^\alpha + \tau} \ldots \boldsymbol{\square_\tau}\ \triangleright^\alpha & \\ (e_1 \ldots C\ e_2 \ldots \triangleright^\beta)\overline{\triangleright^\alpha + \tau} & = & e_1\overline{\triangleright^\alpha + \tau} \ldots C\overline{\triangleright^\alpha + \tau}\ e_2\overline{\triangleright^\alpha + \tau} \ldots \triangleright^\beta & \text{where } \triangleright^\alpha \neq \triangleright^\beta \\ (e_1 \ldots C\ e_2 \ldots)\overline{\triangleright^\alpha + \tau} & = & e_1\overline{\triangleright^\alpha + \tau} \ldots C\overline{\triangleright^\alpha + \tau}\ e_2\overline{\triangleright^\alpha + \tau} \ldots & \\ (\lambda\ (x : \tau_1 \ldots \triangleright^\alpha)\ C)\overline{\triangleright^\alpha + \tau} & = & \lambda\ (x : \tau_1\overline{\triangleright^\alpha + \tau} \ldots \boldsymbol{y : \tau}\ \triangleright^\alpha)\ C\overline{\triangleright^\alpha + \tau} & \text{where } y \text{ is } \textit{fresh} \\ (\lambda\ (x : \tau_1 \ldots \triangleright^\beta)\ C)\overline{\triangleright^\alpha + \tau} & = & \lambda\ (x : \tau_1\overline{\triangleright^\alpha + \tau} \ldots \triangleright^\beta)\ C\overline{\triangleright^\alpha + \tau} & \text{where } \triangleright^\alpha \neq \triangleright^\beta \\ (\lambda\ (x : \tau_1 \ldots)\ C)\overline{\triangleright^\alpha + \tau} & = & \lambda\ (x : \tau_1\overline{\triangleright^\alpha + \tau} \ldots)\ C\overline{\triangleright^\alpha + \tau} & \\ \Diamond\overline{\triangleright^\alpha + \tau} & = & \Diamond & \end{array}$$

[1] For this and following definitions, we emphasize changes.

Backwards compatibility. A useful feature of this extended small-step relation is that it is fully backwards compatible with the top-down approach. That is, when choosing how to instantiate a typed hole, we can opt to use one of the $\lambda_\triangleright$-specific rules, or one of the productions rules of the previous section, with the choice being weighted by annotations under user control. Although fully replacing function and application generation with their $\lambda_\triangleright$ variants would restrict the space of generated programs, this is not a problem in practice as these rules only give people crafting generators the flexibility to bias generation towards creating functions that use their arguments, a bias that cannot be expressed so straightforwardly in a top-down setting.

Nonlocal let-insertion. The next nonlocal rule we introduce is to create a fresh variable when needed, but to insert it as a let-binding rather than a function parameter. We use the isomorphism

$$\texttt{let}\ x : \tau\ =\ e\ \texttt{in}\ e' \cong (\lambda\ (x : \tau)\ e')\ e$$

and will use `let` rather than writing the application and lambda in the following. We can insert a let-binding by breaking the surrounding context of a hole into two pieces as before, and inserting a `let` between them.

GenLet
$$C_1 \blacklozenge C_2 \blacklozenge \square_\tau \Longrightarrow C_1 \blacklozenge (\texttt{let}\ x : \tau\ =\ \square_\tau\ \texttt{in}\ \lozenge) \blacklozenge C_2 \blacklozenge x \quad \text{where } x \text{ is } \textit{fresh}$$

This generation rule can be viewed as a combination of previously added rules, but not as a simple sequencing of choices. This rule is equivalent to a generation step of GenApp and GenAbs, if they had been chosen earlier in generation, before the generation of C_2, along with a final GenVar. What we've done is delayed the decision to insert this let-binding to the point where we've decided to use the binding.

3.3 More Types

Polymorphism. In the original work by Pałka et al. (2011), generated terms did not *produce* polymorphic functions. However, they were able to *use* polymorphic constants, such as `seq` and `map`, filling in type variables that are free in the codomain with randomly generated types in the GenFunVar cases for such variables. We can similarly use polymorphic constants in our approach. Still, *generating* polymorphic functions is seemingly out of reach.

Imagine how we might extend the top-down approach to polymorphism: we could extend the GenApp case to instead generate a polymorphic function type whose codomain is compatible with the generation target type. However, when generating the type up front it is surprisingly easy to back the algorithm into a corner by attempting to generate a term of an uninhabited type. A pitfall that our framework provides a potential, though incomplete, way out from.

The idea is when generating an extensible function type, e.g. $\triangleright^\alpha \to \texttt{List Int}$, we can first *weaken* the output type to a polymorphic variable, making the type $\forall z, \triangleright^\alpha \to \texttt{List}\ z$, and then proceed with generation as normal.

To generate a polymorphic function we produce a lambda abstraction whose body produces the weakened polymorphic type, and in the expansion of that hole, new variables whose types contain the type variable z can be added to the parameter list as needed. To see how this guarantees no backtracking will be needed let's step through an example generation of the safe list head function, where a default value is provided for the empty list case. The example is shown in Figure 2.

(1) We begin with a type-abstraction Λ introducing the polymorphic type z, and an extensible function with $\triangleright^\alpha$ whose body is a hole of type z.

$$
\begin{array}{lll}
\Lambda z.\lambda\ (\qquad\qquad\qquad \triangleright^{\alpha})\ \square_{z} & & (1)\\
\qquad\qquad \Longrightarrow \textsc{GenFunVar} & & \\
\Lambda z.\lambda\ (\qquad\qquad\qquad \triangleright^{\alpha})\ \texttt{listCase}\ \square_{\texttt{List}\ z}\ \square_{z}\ \square_{z\ (\texttt{List}\ z)\to z} & & (2)\\
\qquad\qquad \Longrightarrow \textsc{GenParam}\triangleright & & \\
\Lambda z.\lambda\ (l : (\texttt{List}\ z) \qquad \triangleright^{\alpha})\ \texttt{listCase}\ l\ \square_{z}\ \square_{z\ (\texttt{List}\ z)\to z} & & (3)\\
\qquad\qquad \Longrightarrow \textsc{GenAbs} & & \\
\Lambda z.\lambda\ (l : (\texttt{List}\ z) \qquad \triangleright^{\alpha})\ \texttt{listCase}\ l\ \square_{z}\ (\lambda\ (x : z\ xs : (\texttt{List}\ z))\ \square_{z}) & & (4)\\
\qquad\qquad \Longrightarrow \textsc{GenVar} & & \\
\Lambda z.\lambda\ (l : (\texttt{List}\ z) \qquad \triangleright^{\alpha})\ \texttt{listCase}\ l\ \square_{z}\ (\lambda\ (x : z\ xs : (\texttt{List}\ z))\ x) & & (5)\\
\qquad\qquad \Longrightarrow \textsc{GenParam}\triangleright & & \\
\Lambda z.\lambda\ (l : (\texttt{List}\ z)\ \ c : z\ \triangleright^{\alpha})\ \texttt{listCase}\ l\ c\ (\lambda\ (x : z\ xs : (\texttt{List}\ z))\ x) & & (6)
\end{array}
$$

Fig. 2. Generation of a polymorphic safe list head function

(2) From here we use the GenFunVar rule to create an application of the eliminator function for `List`, `listCase`. This gives three holes for the list being eliminated, a value for the empty case, and a function with arguments for both of the components of a cons.
(3) First we focus on the list hole, replacing it with a parameter that is added at $\triangleright^{\alpha}$.
(4) Next we fill in the cons case with a function. This adds another hole of type z, can reference the newly introduced variables x and xs.
(5) We can fill in this newest hole with a reference to x, leaving only the empty case of `listCase`.
(6) This final step is asking us to create a value for a purely polymorphic type without any variables of that type in the context. From this point we could choose to create an application, but that would only be delaying the inevitable since that path would still eventually require filling in a hole of type z. So instead we take the only route out and use GenParam$\triangleright$ to insert an argument for the base value at $\triangleright^{\alpha}$, finishing the generation.

The resulting polymorphic type for the function is inhabited *by construction*, since the creation of the type occurs concurrently with the creation of a witness. This technique works to generate polymorphic terms when we focus generation on just the polymorphic function. However, it breaks down when the generation of the polymorphic term becomes non-trivially constrained with the rest of the program.

More precisely, crucial to the simplicity of our generation rules is the idea that the type extension operation can be applied globally — we can scan over the whole program and extend every occurrence of an arguments hole with a constant type. This is no longer the case when generating polymorphic functions because they may be applied with multiple different types: the extension operation now needs to consider how the polymorphic type is being instantiated locally. More concerningly, however, extending a polymorphic function can induce an extension in non-polymorphic functions that are unified with the polymorphic function arbitrarily far away in the program.

For instance, consider the following partially generated program:

$$
\begin{array}{l}
\texttt{let}\ f : \forall z.\ z \triangleright^{\alpha} \to \texttt{List}\ z\ =\ \Lambda z.\ \lambda\ (x : z \triangleright^{\alpha})\ \texttt{cons}\ x\ \square_{\texttt{List}\ z}\ \texttt{in}\\
\texttt{let}\ g : \texttt{Int} \triangleright^{\alpha} \to \texttt{List Int}\ =\ \lambda\ (n : \texttt{Int} \triangleright^{\alpha})\ \square_{\texttt{List Int}}\ \texttt{in}\\
(\texttt{if}\ \ldots\ \texttt{then}\ f\ \texttt{Int else}\ g)\ 0\ \triangleright^{\alpha}
\end{array}
$$

Here the generation has defined two functions; f and g. f has a polymorphic type $\forall z.\ z \triangleright^{\alpha} \to \texttt{List}\ z$. At the moment, its body is partially filled in with a call to `cons`, adding its argument x onto an

expression hole. The function g has the same type but specialized to `Int`. Below the `lets` there is an application that applies either f or g to 0.

What happens when we extend the hole in f with a variable bound at $\triangleright^{\alpha}$? How do we figure out what type to extend the application with? With the rules as written we would be extending $\triangleright^{\alpha}$ with `List` z, but this isn't a well-formed type since z is outside its binding! We need to find some mapping from the free type variables to types that are well defined at the current location. Going back to the current example the mapping we want is $z \mapsto$ `List Int`. Unfortunately finding this out requires peeking into the function position of the application and finding the constraint from g.

There is also an inverse problem: if g is extended with an argument of type `List Int`, the easiest thing to do is to add an argument with that type to f, which indeed has no issues type-wise. However, since f produces a polymorphic type, an argument of type `List Int` is likely too specific to be used within the generation of f. Instead, it might be wise to extend f with the weakened type `List` z. Although for arbitrary types there could be many such weakenings with no clear rule for what choices would be be more desirable. For example if instead we consider an extension with `Int` $\times$ `Int` we could consider any of $z \times$ `Int`, `Int` $\times z$, $z \times z$, $\forall y.\ z \times y$, etc. However, since all of these types are perfectly valid to fill in here it is not a serious issue to generating polymorphic functions.

While it seems like there should always be a way to deduce what a valid type substitution to be used at a given location would be, the fact that it requires a nontrivial search of the context adds concerns that such a process might not be practical or guaranteed to terminate, so further exploration of generating polymorphic functions is left to future work.

Lists and Matching. Extending our algorithm to a more type-rich setting, comes with another instance of the problem in the form of match-bound variables. For a concrete example, consider the type of lists: we can operate on them using `match` expressions of the form:

$$\texttt{match}\ e\ \texttt{with}\ [] \implies e \mid x :: xs \implies e$$

To generate matches, we could use typing rule derived local generation rule:

GenMatchLocal
$$C \blacklozenge \square_{\tau} \Longrightarrow C \blacklozenge (\texttt{match}\ \square_{\texttt{List}\ \tau'}\ \texttt{with}\ [] \implies \square_{\tau} \mid x :: xs \implies \square_{\tau})$$
where x, xs are *fresh* and τ' is some type

This rule will replace a hole $\square_{\tau}$ with three new ones: one for the list expression, one for the empty case, and one for the non-empty case. However, once again, this rule suffers from the problem of guessing an arbitrary type. This can be partially addressed by adding a rule analogous to GenFunVar:

GenMatchVar
$$C \blacklozenge \square_{\tau} \Longrightarrow C \blacklozenge \texttt{match}\ x\ \texttt{with}\ [] \implies \square_{\tau} \mid y :: ys \implies \square_{\tau}$$
where $[x \mapsto \texttt{List}\ \tau] \in \Gamma(C)$ and y, ys are *fresh*

But this suffers from the same limitation as that one: it requires the generation to have already just happened to produce a variable of the right type. Using the same guiding principle as the beginning of this Section, we can devise the following rule to avoid that, where we write $\tau(C_2 \blacklozenge \square_{\tau})$ for the type of $C_2 \blacklozenge \square_{\tau}$:

GenMatch
$$C_1 \blacklozenge C_2 \blacklozenge \square_{\tau} \Longrightarrow C_1 \blacklozenge (\texttt{match}\ \square_{\texttt{List}\ \tau}\ \texttt{with}\ [] \implies C_2 \blacklozenge \square_{\tau} \mid x :: xs \implies \Diamond) \blacklozenge x$$
where x and xs are *fresh*

We could define a similar rule that binds xs instead. This rule is very similar to GenLet; in that rule we observed that it would be equivalent to make the right guesses at an earlier point in generation. In this rule we have same relationship: we could have decided to use GenMatchLocal

earlier with the right guesses for types. However, by delaying this decision we have ensured that we don't need to make any guesses for types at all. These rules may seem odd at first in that they are still introducing variable bindings that are not guaranteed to be used, only ensuring that one of x or xs is used depending on which match generation rule was chosen. We argue that this type of variable non-usage is not as important as that of function parameter non-usage. That is because the act of pattern matching against a value is itself a useful step of computation, with the control flow of the program depending on the value being matched against.

However, we found that letting the generator create too many control branches can be problematic as often only one control path will ever be taken by the generated program. This in effect means the generation time spent on the other branches is wasted in a similar manner to time spent on the generation of function arguments without uses.

3.4 Implementation

We implemented a generator in OCaml using the production rules discussed in Section 3.2, in addition to GenConst and GenVar. A naïve implementation of the formal framework where, for example, every time an arguments hole was extended we iterated over the whole program term, would not be efficient. For efficiency purposes, our implementation leverages the following key techniques:

Expressions are mutable cells and encode their context: Mutable expressions make the insertion of parameters, arguments, etc. efficient because we can make a local change to an expression without reconstructing the entire program term to account for the new changes. Additionally, sub-expression terms contain a pointer "upwards" to their predecessor (i.e. encoding their context) so that the global program term can be traversed in both directions — when creating a new variable, we traverse upwards from the hole until we find the place we'd like to insert the binding.

Tracking the available holes: We maintain a list of the cells which are currently holes in a worklist. This means that we never have to search the program term to find existing holes; we always know every hole's location.

Tracking the locations of extensions: We have maps from arguments holes to all types, lambdas, and applications that use that arguments hole. This means that the global operation of searching the program term for the particular arguments hole being extended can be made efficient: we simply modify the appropriate cells from the map.

Rule selection logic: We have implemented a framework for easily expressing and modifying complex weight functions for choosing generation steps. This was in large part enabled by the small-step approach to generation, since each step can produce the generated sub-terms as holes rather than recursive calls. Our framework uses urns (Lampropoulos et al. 2017b) to efficiently sample from each valid distinct generation step.

In addition to the performance-oriented techniques above, we also focused our attention towards low-level choices that impact the posterior distribution of generated terms:

Functions with empty domains: A key difference between our generator and Pałka et al. (2011) is that the terms produced by our generator are in the multary lambda calculus. One of the caveats of this means that our generator risks generating functions that never get a parameter added by generation, and so are left with no parameters. While this is not necessarily an issue, if too many functions turn out this way generated programs will primarily be thunking and forcing expressions rather than passing data around in complex ways. In practice, however, we found that weighting

the insertion of arguments into functions with fewer parameters more heavily was sufficient to ensure that almost all functions have parameters.

Extraction to plain lambda calculus. Because we are generating in the multary lambda calculus, we also need to devise a method of extraction to test languages with only single arity. The easiest choice is to encode parameters as tuples, making all function calls uncurried. Another choice is to extract to curried functions, treating empty parameters as thunks.

No direct control over function domains. When generating a function of type $\triangleright^{\alpha} \rightarrow \tau$, we're not able to place constraints on how complicated the domain will get — the body and non-local changes to the program term decide what the domain looks like. Even crude measures such as preventing GenParam▷ from being chosen if the type is too complex fall short when there are a lot of higher-order usages of extensible functions. This can become an issue for generation because it is often the case that the generator will run out of fuel before filling in holes for these complex types, resulting in many complex function types that just throw out all their values - exactly what we were trying to avoid! In practice, the same distribution tuning as before—deprioritizing insertions into functions with too many parameters—greatly lessens this tendency.

4 EVALUATION

This whole project began when we tried to use random testing to evaluate the correctness of student-written compilers for the needs of an undergraduate compilers course. We reimplemented the state-of-the-art approach to generating well-typed lambda terms (Pałka et al. 2011), and we soon realized that we were having trouble consistently finding certain bugs dealing with intricate function argument usage—such as miscompilations that led to stack clobbering. Upon further inspection, we noticed that our top-down generator was rarely generating functions that actually *used* their arguments, which pointed us towards the nonlocal generation approach.

Naturally, to *evaluate* our approach we will not pit our implementation against our own re-implementation of Pałka et al., but rather turn to their original implementation. In particular, we aim to answer the following three questions:

(1) Is the new generator more effective at finding bugs in compilers?
(2) Does our generator succeed in its goal of generating well-typed terms that use their arguments?
(3) Does that bring it closer in behavior to real human-written programs?

4.1 Replicating the GHC Strictness Analyzer Case Study

To answer the first question, we replicate to the major case study of the original Pałka et al. paper: finding bugs in the strictness analyzer of GHC-6.12. Pałka et al. generated programs of type `List Int → List Int` in an environment seeded with multiple functions from the Haskell standard library, pretty-printed them all to create a single Haskell module, and applied them all to a sequence of partial lists of the form:

```
1:2:undefined
```

GHC-6.12 contained a series of bugs in its strictness analyzer, where the optimizer would sometimes optimize away an expression with a visible side effect. For example, the following term:

```
seq (id (a -> seq a id) (undefined::Int))
```

outputs the following when evaluated on the aforementioned list:

```
[1,2,*** Exception:
```

However, the forcing of the `undefined` is optimized away when compiled with optimizations leading to a different output:

```
*** Exception:
```

As a result, such bugs can be identified using *differential testing*: compiling the same program with and without optimizations and checking that the outputs agree.

To evaluate the performance of our method we reuse the (Pałka et al. 2011) test harness and setup, substituting our implementation to populate the Haskell module. We run both our method and that of Pałka et al. 100 times, with each run consisting of 50 tests, and with each test consisting of generating 1000 functions. For each batch of 1000 functions, we compile twice — once with and once without optimizations, run the compiled output, and check that outputs agree.

We found that our method found a bug with a bit over half the number of generated examples, and about a quarter of the CPU time with a comparable failure rate. The Pałka et al. generation code found bugs in 98 of the runs and took an average of 19.56 tests and 1092 seconds of CPU time to find a bug. Our method found bugs in 99 runs and took an average of 10.08 tests and 237.4 seconds to find a bug. That is, our method was roughly 2× more *effective* at finding bugs and 4× more *efficient* on average. A Mann-Whitney-U test (Mann and Whitney 1947) confirms that our method is statistically better, as the p-value that our method is finding bugs in less tests by chance is $p = 3.014 \cdot 10^{-13}$, and in less CPU time by chance is $p = 1.147 \cdot 10^{-28}$.

An extension of the original work (Pałka 2014) by Pałka et al. describes three other distinct bugs discovered in GHC, which are triggered and detected by using slightly modified generators and test harnesses. Out of the three, we could manually trigger two using the counterexample provided in the thesis, and our generators were also able to find them—although we could not obtain the code accompanying the thesis to compare against.

4.2 Parameter usage

To answer the last two questions, we introduce a straightforward usage metric on programs: we simply calculate the percentage of function parameters that are used by their bodies. In addition, we turn to an independent benchmark suite of functional programs (Quiring et al. 2022), and calculate this metric on those, to get an estimate of what things looks like in practice.

The average parameter usage across all programs in the suite is 94.9%, which validates our premise that functions should use their arguments. Table 1 provides a detailed description of the programs in the benchmark suite and their parameter usage. Finally, we measure the usage rate of both the local and the nonlocal approach. Across 10000 random generation tests, the generator of Pałka et al. produced an average of 30% parameter usage. Our method produced an average of 99.9%—but this can be tuned if desired by increasing the probability that the original generation rule for application (GenApp) is selected.

5 RELATED WORK

We've heavily discussed the pioneering work of Pałka et al. (2011) throughout this paper, as they first introduced the local algorithm and its various refinements. Since then, a number of other papers have tackled the same problem of generating well-typed terms, with varying approaches, goals, and outcomes.

Midtgaard et al. (2017) built upon the local algorithm of Pałka et al. by extending the type system with an effect for identifying programs whose output depends on the evaluation order. They then generalized the local algorithm to take effects into account, generating well-typed programs without such a dependence. Thus, they were able to weed out undefined behaviors that arise from under-specification of the evaluation order and find multiple bugs in OCaml's optimizing

Tab. 1. Benchmark descriptions and usage statistics

Name	LOC	# Used / Total	Description
nucleic	3326	158 / 165 (95%)	The SML version of the "Pseudoknot" program (Hartel et al. 1996).
boyer	838	26 / 26 (100%)	The Boyer-Moore benchmark from SML/NJ.
ratio-regions	548	108 / 117 (92%)	An image segmentation benchmark from MLton.
mc-ray	487	108 / 121 (89%)	A SML port of the "Weekend Raytracer" (Shirley 2020).
knuth-bendix	450	162 / 166 (97%)	The Knuth-Bendix benchmark from SML/NJ.
raytracer	333	102 / 102 (100%)	The Ray tracer from Impala benchmarks.
SNF	290	50 / 52 (96%)	The Smith-Normal-Form benchmark from MLton.
cps-convert	199	29 / 32 (90%)	A SML implementation of the Danvy-Filinski CPS conversion (Danvy and Filinski 1992).
interpreter	178	25 / 25 (100%)	An interpreter for a simple language.
parser-comb	168	70 / 72 (97%)	Using parser combinators to parse a simple expression syntax.
life	122	56 / 57 (98%)	The Life benchmark from SML/NJ.
derivative	113	20 / 21 (95%)	A SML port of the symbolic derivation benchmark from Larceny.
nqueens	45	10 / 10 (100%)	The classic N-Queens benchmark.
quicksort	44	4 / 4 (100%)	Quicksort of integer lists.
mandelbrot	43	6 / 6 (100%)	The Mandelbrot benchmark from SML/NJ
safe-for-space	27	4 / 7 (57%)	An example that tests safe-for-space closure conversion (Shao and Appel 2000)
cpstak	21	6 / 6 (100%)	The continuation-passing-style implementation of the Takeuchi (`tak`) function ported from Larceny.
filter	11	6 / 6 (100%)	Filter a list.
tak	10	1 / 1 (100%)	The Takeuchi (`tak`) function.
ack	8	1 / 1 (100%)	Ackermann's function.

native-code back end. Rocha (2019) further built upon this work by encoding exceptions as an effect in the type system, and using the same approach to find additional bugs in various OCaml compilers. Extending a type system with effects is largely orthogonal to generating functions that use their arguments, so we would expect such approaches to also benefit from the results presented in this paper.

In the same vein, Hoang et al. (2022) extend the algorithm of Pałka et al. to System F. In System F the problem of early generation of types is only exacerbated: it also contains *type* applications! Hoang et al. introduce the notion of *unsubstitution*, a process for generating some types τ, τ' and some type variable α for a given type σ such that $\sigma = \tau[\tau'/a]$. We discussed how our approach interacts with polymorphism in Section 3.3, but it would be interesting future work to explore precisely how it interacts with the particular intricacies of System F.

The local algorithm has also been adapted to an imperative language setting. da Silva Feitosa et al. (2019) use such an approach to generate well-typed Featherweight Java (Igarashi et al. 2001) programs, while more recently Li et al. (2022) did the same for their core calculus for Checked C (Ruef et al. 2019). Both approaches still first target a model of the imperative language that is embedded in a functional one, and therefore the ideas from Pałka et al. carry over seamlessly. On

the other hand, the CSmith project, the crown jewel of imperative compiler testing (Yang et al. 2011), follows a different, more ad-hoc, approach developing a fine-tuned hand-written generator to generate C programs that don't exhibit undefined behavior. Interestingly, it appears that their work also suffers from a similar problem as the one we identified and addressed in this paper, as Barany (2018) showed that one can leverage static analysis during generation to eliminate some forms of dead code, leading to improved efficiency and effectiveness.

A different line of work focuses on studying the statistical properties of lambda terms rather than generating terms for testing purposes. By viewing lambda terms as labeled trees, the well-explored literature on generating combinatorial structures is directly applicable, beginning with Flajolet et al. (1994) who systematized the notion of recursive grammar-based generation almost three decades ago. Since then, a number of approaches have directly used this prism for lambda term generation. Grygiel and Lescanne (2012) provide a mapping from (untyped) lambda terms to natural numbers (called a *ranking*) and an inverse mapping (the *unranking*). This in turn gives rise to a natural generation strategy that first samples a natural number before using the unranking function to uniformly generate terms of an exact size. On the other hand, Lescanne (2014) used the notion of Boltzmann samplers (Duchon et al. 2004) to uniformly generate terms of an approximate size. Unfortunately, while using such combinatorial techniques for generating and studying the characteristics of untyped lambda terms has been fruitful, they appear extremely hard to adapt so that they enforce more semantic constraints such as well-typedness, despite some progress in adapting them to take other structural features into account (Feldt and Poulding 2013).

On the other hand, the work of Kennedy and Vytiniotis (2012) also uses a ranking-unranking like approach, elegantly phrased in the form of games, but they succeed in providing encoding and decoding functions directly from and to well-typed lambda terms. While testing is mentioned as a potential application of their functional pearl, we are not aware of any work that has followed up to demonstrate its viability.

An alternative line of work harnesses the declarative power of logic programming to concisely enumerate lambda terms (Tarau 2015). In his work, Tarau describes a Prolog-based generator and type inference algorithm for simply-typed terms in DeBruijn notation. Taking full advantage of Prolog's unification and backtracking capabilities, he is able to enumerate lambda terms of slightly larger sizes compared to a rejection sampling approach backed by combinatorial techniques for generation of untyped terms. A follow up to this work bridges the gap, exploring a synergy between logic programming and Boltzmann samplers (Bendkowski et al. 2017). By interleaving the two approaches, the authors are able to achieve uniform generation of closed λ calculus terms of a size an order of magnitude higher than before, and use it to gauge statistical properties of lambda terms at scale. It still, however, remains an open question whether testing with a uniform generation of inputs (and the associated overhead that comes with) is an effective approach. For example, Grygiel and Lescanne (2012) observed that "almost all typeable terms start with an abstraction". Unfortunately, a generator that almost always produces such abstractions is completely ineffective at finding bugs in properties that involve reduction, such as progress, preservation, or most specifications of optimization correctness, as no reductions can actually take place.

An interesting thread of related work attempts to automatically derive an algorithm such as the one that appears in Pałka et al. (2011), based on a declarative specification of the type system. Claessen et al. (2015) use an idea similar to the ranking-unranking approach of Grygiel and Lescanne (2012), as implemented in the Haskell FEAT library (Duregård et al. 2012) for enumeration of algebraic datatypes. They then rely on Haskell's laziness to prune large parts of the input space and avoid generating them in the future. Fetscher et al. (2015) extend this work for PLT Redex, introducing a disunification solver to more effectively handle pattern matching constraints. Lampropoulos et al. (2017a) develop a language for writing programs that can be viewed as both generators and

checkers, allowing for lightweight annotations to control both the resulting probability distribution and the amount of constraint solving that happens prior to variable instantiation. All three papers replicate the case study finding bugs in GHC's strictness analyzer, and report an order of magnitude overhead compared to the original implementation. Finally, the case study of Pałka et al. was also replicated in a recent work by Goldstein et al. (2021), where the authors improved upon the original's efficiency and effectiveness by generating multiple inputs and selecting the ones that maximizes the interactions between different constructors—an idea inspired by the combinatorial testing literature. Once again, at a high level these threads are all orthogonal to our contributions.

Finally, we would be remiss to not discuss the related field of program synthesis (Gulwani et al. 2017), which also deals with the problem of producing well-typed programs. In particular, there is a sequence of approaches that attempt to synthesize functions that adhere to a specification following a top-down enumeration approach—not too dissimilar from the Pałka et al. one (Feser et al. 2015; Osera and Zdancewic 2015; Polikarpova et al. 2016). There is, however, a key distinction: the goal in program synthesis is to find *the one* or one of a few handful programs that satisfies a given specification. In other words, the desired solution is over constrained, and randomly producing that is extremely unlikely. Thinking back to the example derivation of the safe head function in Section 3, there were a lot of random choices that had to line up for that outcome. As a result, program synthesis turns to exhaustive techniques (such as SMT solvers) to look for solutions. In contrast, the goal in random testing is to generate many programs in the hopes of finding bugs in a space that can't be exhaustively explored. This distinctions makes all the difference in the world. In the particular case of lambda terms, generation of well-typed terms is an interesting middle ground between testing properties of syntactically correct terms, such as a parser/printer roundtrip property where grammar-based generation will suffice, and program synthesis, where exhaustive techniques need to be employed. The well-typedness precondition is sparse enough to make a generate-and-test approach impractical, but also contains sufficient inhabitants that randomness is necessary for effective exploration.

6 CONCLUSION AND FUTURE WORK

In this paper, we identified a key limitation in the type-directed approaches to well-typed term generation that are prevalent in the literature, which frequently produces functions that do not use their arguments, which causes large chunks of generated code to be ineffective for testing. We devised a new approach that produces functions that use their arguments by allowing the generation to delay the choice of argument types until it is generating uses for those arguments as well. We formalized this approach as a small-step relation in an extension of the simply-typed lambda calculus that is fully backwards compatible with the top-down approach, meaning that the heuristics developed in the past decade of work on the generation of well-typed lambda terms can still be exploited in this setting. In the future, we're hoping to further explore the vast configuration space that arises from the fine-grained level of control that the new algorithm provides, with an eye out for more complex type systems including dependently typed ones.

ACKNOWLEDGMENTS

NSF blah blah

REFERENCES

Gergö Barany. 2018. Liveness-Driven Random Program Generation. In *Logic-Based Program Synthesis and Transformation*, Fabio Fioravanti and John P. Gallagher (Eds.). Springer International Publishing, Cham, 112–127.

Maciej Bendkowski, Katarzyna Grygiel, and Paul Tarau. 2017. Boltzmann Samplers for Closed Simply-Typed Lambda Terms. In *Practical Aspects of Declarative Languages*, Yuliya Lierler and Walid Taha (Eds.). Springer International Publishing, Cham, 120–135.

Koen Claessen, Jonas Duregård, and Michal H. Palka. 2015. Generating constrained random data with uniform distribution. *J. Funct. Program.* 25 (2015). https://doi.org/10.1017/S0956796815000143

Samuel da Silva Feitosa, Rodrigo Geraldo Ribeiro, and Andre Rauber Du Bois. 2019. Generating Random Well-Typed Featherweight Java Programs Using QuickCheck. *Electronic Notes in Theoretical Computer Science* 342 (2019), 3–20. https://doi.org/10.1016/j.entcs.2019.04.002 The proceedings of CLEI 2018, the XLIV Latin American Computing Conference.

Olivier Danvy and Andrzej Filinski. 1992. Representing Control: A study of the CPS transformation. *Mathematical Structures in Computer Science* 2, 4 (Dec. 1992), 361–391. https://doi.org/10.1017/S0960129500001535

Philippe Duchon, Philippe Flajolet, Guy Louchard, and Gilles Schaeffer. 2004. Boltzmann Samplers for the Random Generation of Combinatorial Structures. *Combinatorics, Probability & Computing* 13, 4-5 (2004), 577–625. https://doi.org/10.1017/S0963548304006315

Jonas Duregård, Patrik Jansson, and Meng Wang. 2012. Feat: Functional Enumeration of Algebraic Types. In *Proceedings of the 2012 Haskell Symposium* (Copenhagen, Denmark) *(Haskell '12)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/2364506.2364515

Robert Feldt and Simon M. Poulding. 2013. Finding test data with specific properties via metaheuristic search. In *24th International Symposium on Software Reliability Engineering*. IEEE, 350–359. http://robertfeldt.net/publications/feldt_poulding_2013_finding_test_data_with_specific_properties.pdf

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. ACM, New York, NY, USA, 229–239. https://doi.org/10.1145/2737924.2737977

Burke Fetscher, Koen Claessen, Michal H. Palka, John Hughes, and Robert Bruce Findler. 2015. Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System. In *24th European Symposium on Programming (Lecture Notes in Computer Science, Vol. 9032)*. Springer, 383–405. http://users.eecs.northwestern.edu/~baf111/random-judgments/

Philippe Flajolet, Paul Zimmermann, and Bernard Van Cutsem. 1994. A Calculus for the Random Generation of Labelled Combinatorial Structures. *Theor. Comput. Sci.* 132, 2 (1994), 1–35. https://doi.org/10.1016/0304-3975(94)90226-7

Harrison Goldstein, John Hughes, Leonidas Lampropoulos, and Benjamin C. Pierce. 2021. Do Judge a Test by its Cover: Combining Combinatorial and Property-Based Testing.

Katarzyna Grygiel and Pierre Lescanne. 2012. Counting and generating lambda terms. *CoRR* abs/1210.2610 (2012). arXiv:1210.2610 http://arxiv.org/abs/1210.2610

Sumit Gulwani, Alex Polozov, and Rishabh Singh. 2017. *Program Synthesis*. Vol. 4. NOW. 1–119 pages.

Pieter H. Hartel, Marc Feeley, Martin Alt, Lennart Augustsson, and et al. 1996. Benchmarking implementations of functional languages with 'Pseudoknot', a float-intensive benchmark. *Journal of Functional Programming* 6, 4 (1996), 621–655. https://doi.org/10.1017/S0956796800001891

Tram Hoang, Anton Trunov, Leonidas Lampropoulos, and Ilya Sergey. 2022. Random Testing of a Higher-Order Blockchain Language (Experience Report). In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (may 2001), 396–450. https://doi.org/10.1145/503502.503505

Andrew J. Kennedy and Dimitrios Vytiniotis. 2012. Every bit counts: The binary representation of typed data and programs. *Journal of Functional Programming* 22, 4-5 (2012), 529–573. https://research.microsoft.com/en-us/people/dimitris/jfpcoding.pdf

Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. 2017a. Beginner's Luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 114–129. http://dl.acm.org/citation.cfm?id=3009868

Leonidas Lampropoulos, Antal Spector-Zabusky, and Kenneth Foner. 2017b. Ode on a Random Urn (Functional Pearl). In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell* (Oxford, UK) *(Haskell 2017)*. ACM, New York,

NY, USA, 26–37. https://doi.org/10.1145/3122955.3122959

Pierre Lescanne. 2014. Boltzmann samplers for random generation of lambda terms. *CoRR* abs/1404.3875 (2014). http://arxiv.org/abs/1404.3875

Liyi Li, Yiyun Liu, Deena L. Postol, Leonidas Lampropoulos, David Van Horn, and Michael Hicks. 2022. A Formal Model of Checked C. In *Proceedings of the Computer Security Foundations Symposium (CSF)*.

H. B. Mann and D. R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* 18, 1 (1947), 50 – 60. https://doi.org/10.1214/aoms/1177730491

Jan Midtgaard, Mathias Nygaard Justesen, Patrick Kasting, Flemming Nielson, and Hanne Riis Nielson. 2017. Effect-Driven QuickChecking of Compilers. *Proc. ACM Program. Lang.* 1, ICFP, Article 15 (aug 2017), 23 pages. https://doi.org/10.1145/3110259

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed Program Synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. ACM, New York, NY, USA, 619–630. https://doi.org/10.1145/2737924.2738007

Michał H. Pałka. 2014. *Random Structured Test Data Generation for Black-Box Testing*. Department of Computer Science and Engineering, Software Technology (Chalmers), Chalmers University of Technology,. http://publications.lib.chalmers.se/records/fulltext/195849/195849.pdf 168.

Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an Optimising Compiler by Generating Random Lambda Terms. In *Proceedings of the 6th International Workshop on Automation of Software Test* (Waikiki, Honolulu, HI, USA) *(AST '11)*. ACM, New York, NY, USA, 91–97. https://doi.org/10.1145/1982595.1982615

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. *SIGPLAN Not.* 51, 6 (June 2016), 522–538. https://doi.org/10.1145/2980983.2908093

Benjamin Quiring, John Reppy, and Olin Shivers. 2022. Analyzing Binding Extent in 3CPS. *Proc. ACM Program. Lang.* 6, ICFP, Article 114 (aug 2022), 29 pages. https://doi.org/10.1145/3547645

Jason S. Reich, Matthew Naylor, and Colin Runciman. 2012. Advances in Lazy SmallCheck. Presented at the 24th Symposium on Implementation and Application of Functional Languages. http://www.cs.york.ac.uk/fp/jason-docs/ReichNaylorRunciman2013.pdf

Murilo Giacometti Rocha. 2019. *Testing of OCaml exceptions by effect-driven generation of programs*. Master's thesis. University of Edinburgh. https://project-archive.inf.ed.ac.uk/msc/20193481/msc_proj.pdf

Andrew Ruef, Leonidas Lampropoulos, Ian Sweet, David Tarditi, and Michael Hicks. 2019. Achieving Safety Incrementally with Checked C. In *Proceedings of the Symposium on Principles of Security and Trust (POST)*.

Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. SmallCheck and Lazy SmallCheck: automatic exhaustive testing for small values. In *1st ACM SIGPLAN Symposium on Haskell*. ACM, 37–48. http://www.cs.york.ac.uk/fp/smallcheck/smallcheck.pdf

Zhong Shao and Andrew W. Appel. 2000. Efficient and Safe-for-Space Closure Conversion. *ACM Transactions on Programming Languages and Systems* 22, 1 (Jan. 2000), 129–161. https://doi.org/10.1145/345099.345125

Peter Shirley. 2020. *Ray Tracing in One Weekend*. https://raytracing.github.io

Paul Tarau. 2015. On Type-directed Generation of Lambda Terms. In *Proceedings of the Technical Communications of the 31st International Conference on Logic Programming (ICLP 2015), Cork, Ireland, August 31 - September 4, 2015*. http://ceur-ws.org/Vol-1433/tc_12.pdf

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. 283–294. https://doi.org/10.1145/1993498.1993532