Improving the Stability of Type Soundness Proofs in Dafny

Joseph W. Cutler
jwc@seas.upenn.edu
University of Pennsylvania
Philadelphia, Pennsylvania, USA

Emina Torlak
emina@cs.washington.edu
Amazon Web Services
Seattle, Washington, USA

Michael Hicks
mwh@cs.umd.edu
Amazon Web Services
Arlington, Virginia, USA

Abstract

In this extended abstract, we present a method for structuring type soundness proofs in Dafny to improve proof stability. As a case study, we apply the method to proving type soundness for a small expression language, and demonstrate empirically how it improves resource usage metrics known to correlate with stability. Our method can scale to realistic proofs, as demonstrated by its use in the type soundness proof of the Cedar language.

ACM Reference Format:
Joseph W. Cutler, Emina Torlak, and Michael Hicks. 2023. Improving the Stability of Type Soundness Proofs in Dafny. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Type soundness proofs for strongly-typed programming languages are often carried out in type-theory based proof assistants like Coq or Agda. However, recent small-scale experiements [4] and large-scale mechanized metatheory developments [7]. have shown that Dafny can also encode type soundness proofs. In these developments, Dafny serves both as the host language in which the target language is implemented, and a program verifier in which the mechanized proof is completed.

The target language semantics is implemented in Dafny is by way of a definitional interpreter [5], a function `eval` which takes a term `e` in the target language, and returns either an error, or the value `v` the term evaluates to. In principle, this implements some abstract semantics defined by inference rules on paper: a semantic judgment $\mathtt{e} \Downarrow \mathtt{v}$ holds if and only if `eval(e) = Ok(v)`.

The other half of a language implementation is a type-checker: a boolean function `check` that takes a term `e` and a type `t`, and returns true or false if `e` has type `t`. Again, this implements some abstract typing judgment $\vdash \mathtt{e} : \mathtt{t}$, which is defined by way of inference rules on paper.

The goal of a type soundness proof is then to demonstrate that well-typed programs do not go wrong:

$$\vdash \mathtt{e} : \mathtt{t} \implies \exists \mathtt{v}.\, (\mathtt{e} \Downarrow \mathtt{v} \land \mathtt{v} : \mathtt{t})$$

for some value `v` of type `t`. When proven on paper, this proof proceeds by an induction on the typing derivation of $\vdash \mathtt{e} : \mathtt{t}$. For each rule defining the typing judgment, we prove that if the premises do not go wrong, then the conclusion also does not go wrong. Meanwhile, a *mechanized* proof of this fact in Dafny takes the form of a lemma which requires `check(e,t)`, and ensures that `eval(e) = Ok(v)` and that `v` has type `t`. The proof of this lemma necessarily looks like a pattern-match on the term `e`, making recursive calls on the subterms to establish the inductive hyoptheses. For syntax-directed type systems of the sort we will consider in this extended abstract[1], the structure of these two kinds of proofs are the same: each case of the pattern match encodes the proof case for the typing rule of the corresponding syntactic form.

The size of a type soundness proof in Dafny grows linearly with the size of the language; more language features and more expressions mean more cases. This can make it hard to ensure that the proof is *stable*. Unlike mechanized metatheory developments in tools backed by a type-theoretic proof checker, Dafny's SMT-backed verification can be *unstable*: the result of verifying the type soundness theorem can change, from verified to unverified, due to minor (and even unrelated) changes to the code. Unfortunately, folowing the proof structure described above leads to very unstable proofs in Dafny. Anecdotally, we have found it impossible to scale this kind of proof to anything approximating a realistic language without encountering enormous proof instability barriers.

In this extended abstract, we give a recipe for mechanized type soundness proofs in Dafny which ensures proof stability. By giving partial specifications of the evaluator and type checker, and proving type soundness relative to those specifications, we can eliminate much of the variance in resource usage. Our technique also scales to realistic-size languages, as witnessed by its use to verify the type soundness proof of the Cedar language [6].

In Section 2, we present the technique by walking through a traditional-style proof of type soundness in Dafny. Then in Section 3, we evaluate the technique comparing empirical

[1]Strictly speaking, this is not a restriction of our technique. By far the most common way to implement a non-syntax-directed type system is to build a syntax-directed version, prove the two versions equivalent, and then implement the syntax-directed one.

Conference'17, July 2017, Washington, DC, USA
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
function eval(env : Env, e : Term) :
    Result<Val,EvErr>
{
  match e {
    ...
    case Add(e1,e2) =>
      var n1 :- evalInt(env,e1);
      var n2 :- evalInt(env,e2);
      Ok(IntVal(n1 + n2))
  }
}
```

Figure 1. The `eval` function, with the case for `Add`

```
function check(ctx : Ctx, e : Term, t : Ty)
    : Result<(),TckErr> {
  var t' :- infer(ctx,e);
  if t == t' then Ok(()) else Err(TckErr)
}

function infer(ctx : Ctx, e : Term) :
    Result<Ty,TckErr> {
  match e {
    ...
    case Add(e1,e2) =>
      var _ :- inferIntTy(ctx,e1);
      var _ :- inferIntTy(ctx,e2);
      Ok(IntTy)
    }
  }
```

Figure 2. The `check` and `infer` functions, with the case for `Add`

metrics of proof stability against other methods of structuring type soundness proofs, and other ways of writing a typechecker. Lastly, in Section 4, we discuss how our technique has been deployed in practice, speculate about how it can be applied to even more complex languages including Cedar, and discuss the limitations and drawbacks.

2 Stable Type Soundness Proofs

For a running example, let's consider the first-order expression language we use for our evaluation in Section 3. The full language definition, as well as the interpreter and all of the different typecheckers we evaluate can be found in the attached supplementary materials. The language has types for integers, booleans, and records, all given as the Dafny type `Ty`. For terms, we have variables (represented as strings, since the langauge has no binders), addition, subtraction, division, and, or, conditionals, and record expressions and projections. These are all represented by a Dafny ADT `Term`, which has a variant for each syntactic form. To demonstrate how the language works, and how our technique applies, we focus on just the behavior of the addition operation.

The language's semantics is implemented by a function `eval(env,e)` which takes an environment mapping variables to their values and a term, and returns either (a) the value the term evaluates to, or (b) an evaluation error, which can be a division by zero error or a runtime type error.

The function header for the `eval` function is shown in Figure 1. The function `evalInt` is a helper that calls `eval`, and then pattern matches on the result, returning `n` if it was `IntVal(n)`, and throwing a runtime type error otherwise.

Terms are typed in a context mapping variables to their types, with a typechecking implemented by a program `check(ctx,e,t)` which calls an inference function `infer`, and then checks if the result is equal to `t`. Code for `check` and the `Add` case for `infer` are shown in Figure 2, where `inferIntTy` calls `infer`, and throws a type error if the result is anything but `IntTy`.

A First Cut

For this language, type soundness means the following. If `e` checks against `t` in context `ctx` and the environment `env` agrees with the context `ctx`, then `e` either evaluates under the environment `env` to a value of type `t`, or it results in a division by zero error (but not a runtime type error). This lemma is encoded in dafny as the `sound` lemma in Figure 3, along with auxiliary definitions: `envHasCtx` is the predicate saying that the environment agrees with the context, and `isSafe(env,e,t)` encodes the conclusion of the theorem.

To prove the `sound` lemma, we induct on the term `e`. The case of `sound` for `Add(e1,e2)` is illustrative. We make two recursive calls to the lemma to introduce the inductive hypotheses, and the solver takes care of the rest.

The reasoning that the solver takes care of under the hood is somewhat involved. Getting from the assumption `check(ctx,Add(e1,e2),t).Some?` to the IH preconditions `check(ctx,e1,IntTy).Some?` and `check(ctx,e2,IntTy).Some?` requires reasoning about the (potentially complex) code for `check` and `infer`. Similarly, getting from the IH results `isSafe(env,e1,IntTy)` and `isSafe(env,e2,IntTy)` to the conclusion `isSafe(env,Add(e1,e2),t)` requires reasoning through the code for `eval` to determine the possible ways `Add(e1,e2)` evaluates when `e1` and `e2` might either evaluate to values or raise division-by-zero errors.

Finding Stability

Unfortunately, as we'll see in Section 3, this proof approach is not stable. We realized this while attemping to scale this style of proof up to a realistic language. With enough operations and language features, the proof resource usage varies wildly between runs, enough that the verification will fail unpredictably. At its core, this instability arises from all of the afformentioned unguided reasoning the solver has to

```
lemma sound(env : Env, ctx : Ctx, e : term,
    t : Ty)
  requires envHasCtx(env,ctx)
  requires check(ctx,e,t).Some?
  ensures isSafe(env,e,t)
{
  match e {
    ...
    case Add(e1,e2) =>
      sound(env,ctx,e1,IntTy);
      sound(env,ctx,e2,IntTy);
  }
}

predicate envHasCtx(env : Env, ctx : Ctx) {
  forall x :: x in ctx ==>
    x in env &&
    valHasType(env[x],ctx[x])
}

predicate isSafe(env : Env, e  : Term, t :
    Ty){
  (eval(env,e).Ok? &&
    valHasType(eval(env,e).value,t))
  ||
  (eval(env,e).Err? && eval(env,e).error ==
    DivByZero)
}
```

Figure 3. The sound lemma with the case for Add, and auxiliary definitions

do about the code of the typechecker and evaluator, to get from the premise of the lemma to the premises of the IH, and then from the conclusions of the IH to the conclusion of the case. A patchwork solution is to add guidance in the form of assertions around the recursive calls, to spell out the intermediate steps more directly.

The common solution to dealing with proof instability is to specify the functions involved in the proof, and then make the functions themselves opaque. This way, the solver can only interact with the functions though the specification, cutting down the search space of possible proofs and hence improving stability. Moreover, if the specifications are written in the form of separate lemmas, the programmer can control how and when different parts of the function's specification is revealed to the solver. In this case, the dilemma is that it's not at all clear which functions to make opaque, and how we should specify them.

Inspiration comes from noticing that sound depends only on facts about the safety predicate `isSafe`, and not directly on facts about evaluation. In fact, the only facts it needs about `isSafe` are that in every case, the results of the IH calls jointly imply the conclusion. To illustrate, the addition case for type soundness holds for *any* safety predicate

```
lemma addIsSafe(env : Env, e1 : Expr, e2 :
    Expr)
  requires isSafe(env,e1,IntTy)
  requires isSafe(env,e2,IntTy)
  ensures isSafe(env,Add(e1,e2),IntTy)
{reveal isSafe(); ...}
```

Figure 4. Add Compatibility Lemma

$$\frac{\mathsf{ctx} \vdash \mathsf{e1} : \mathsf{IntTy} \qquad \mathsf{ctx} \vdash \mathsf{e2} : \mathsf{IntTy}}{\mathsf{ctx} \vdash \mathsf{Add(e1,e2)} : \mathsf{IntTy}}$$

$$\frac{\mathtt{isSafe(env,e1,IntTy)} \qquad \mathtt{isSafe(env,e1,IntTy)}}{\mathtt{isSafe(env,Add(e1,e2),IntTy)}}$$

Figure 5. Addition Typing Rule, and the Corresponding Compatibility Lemma

`isSafe` — even an opaque one — which has the property that `isSafe(env,e1,IntTy)` and `isSafe(env,e2,IntTy)` together imply `isSafe(env,Add(e1,e2),IntTy)`.

This lemma, shown as `addIsSafe` in Figure 4 is a kind of "compatibility lemma", stating that safe terms can be built from smaller safe terms. The upshot from this is that if we make the `isSafe` predicate opaque, the Add case of the soundess theorem goes through with only minor modification, adding a call to `addIsSafe` after the uses of IH.

```
lemma sound(env : Env, ctx : Ctx, e : term,
    t : Ty)
  requires envHasCtx(env,ctx)
  requires check(ctx,e,t).Some?
  ensures isSafe(env,e,t)
{
  match e {
    ...
    case Add(e1,e2) =>
      sound(env,ctx,e1,IntTy);
      sound(env,ctx,e2,IntTy);
      addIsSafe(env,e1,e2);
  }
}
```

One way of thinking about the `addIsSafe` lemma is that it says that the safety predicate *interprets the typing rule* for addition. By replacing all instances of the typing judgent in the rule for Addition that the case of `infer` for `Add(e1,e2)` implements, we arrive at the required `addIsSafe` compatibility lemma: this is demonstrated in Figure 5. In short, all that's required for a given safety predicate to hold for every well-typed term is that it interprets every typing rule in the language. This suggests the following technique:

```
  lemma invertAddCheck(ctx : Ctx, e1 :
    Term, e2 : Term)
  requires invert(ctx,Add(e1,e2),t).Ok?
  ensures check(ctx,e1,IntTy).Ok?
  ensures check(ctx,e2,IntTy).Ok?
  ensures t == IntTy
{ reveal infer(); reveal check(); }
```

Figure 6. Inversion Lemma for Addition

1. Make the safety predicate opaque, preventing the solver from directly reasoning about the interpreter in the type safety proof.
2. Prove "safety compatibility lemmas" for every typing rule, demonstrating that if the safety predicate holds of all the premises, it holds of the conclusion.
3. Write the type soundness proof, inserting calls to each case's corresponding compatibility lemma, just after the recursive calls to IH.

Going Further with Inversion Lemmas

Even with this modification, the type soundness proof still requires the solver to do complex reasoning about the code of the typechecker. To eliminate this unguided reasoning, we must find a way to specify the typechecker enough to make it opaque, and have the main soundness proof refer only to the specification lemmas.

Our solution again comes from an analysis of what the solver needs to know about `check` in each case. In the `Add(e1,e2)` case, the solver must reason from `check(ctx,Add(e1,e2),t).Ok?` to `check(ctx,e1,IntTy).Ok?` and `check(ctx,e2,IntTy).Ok?` for the preconditions of the IH to hold, essentially running the code of `infer` and `check` in reverse. This kind of reasoning corresponds to what's known in the type systems literature as *inversion principles*: theorems that state that if the a compound syntactic form (like `Add(e1,e2)`) is well-typed, then its component forms (here, `e1` and `e2`) are well-typed. Moreover, this inversion lemma, shown in Figure 6 is slightly stronger, saying that the type `t` in question must have been `IntTy`.

This then allows us to make the `check` and `infer` functions opaque, and modify the proof of the soundness theorem once more to add a call to this inversion lemma before the calls to IH.

```
lemma sound(env : Env, ctx : Ctx, e : term,
    t : Ty)
  requires envHasCtx(env,ctx)
  requires check(ctx,e,t).Some?
  ensures isSafe(env,e,t)
{
  match e {
    ...
    case Add(e1,e2) =>
      invertAddCheck(ctx,e1,e2);
      sound(env,ctx,e1,IntTy);
      sound(env,ctx,e2,IntTy);
      addIsSafe(env,e1,e2);
  }
}
```

This gives us the final recipe for our technique:

1. Make the safety predicate opaque, preventing the solver from directly reasoning about the interpreter in the type safety proof.
2. Prove safety compatibility lemmas for every typing rule, demonstrating that if the safety predicate holds of all the premises, it holds of the conclusion.
3. Make the typechecker opaque
4. Prove inversion lemmas for every typing rule, showing that if the typechecker can compute that the concluson holds, it must also be able to compute that the premises.
5. Write the type soundness proof, inserting calls to the inversion lemmas before IH calls, and inserting calls to the compatibility lemma after.

3 Evaluation

In this section, we demonstrate empirically that our technique as described in Section 2 does yield more stable type soundness proofs. As a quantative proxy for stability, we measure the resource usage of our proofs. If a proof is expensive, or its cost varies wildly between runs, this can be an indicator for future proof instability [3] [9]. While resource usage is the most important indicator of future proof instability, we also base our conclusions on how much verification duration — the wall-clock time it takes for the solver to prove the VCs — varies between runs.

Experimental Setup

We evaluate five different typecheckers and type soundness proofs for the same language to compare their proof stability. The code for these typecheckers, proofs, and the language's evaluator, can be found in the supplementary materials. The five different typecheckers, as well as the contents of the rest of the files, are described below.

- (`lang.dfy`): The langauge and type definitions, common to all typecheckers and the evaluator.
- (`eval.dfy`): Contains the evaluator for our expressiong language. All of the type soundness proofs are relative to this evaluator.
- (`basic.dfy`): This is the typechecker and type safety proof described in the first part of Section 2. The typechecker uses one main method `infer` and some helper functions to infer a type for a term, and then checks that it's equal to the given type.
- (`unfolded.dfy`): This is the same as the typechecker in `basic.dfy`, with all of the syntactic sugar for `Result`-binds (`:-`) unfolded, and all of the helper functions inlined. We include this file to measure the degree to

which adding layers of abstraction hurts proof stability. The proof is essentially the same as the one in `basic.dfy`.
- (`bidirectional.dfy`): This typechecker is written in the same style as the typechecker in `basic.dfy`. The typing relation that it impelents is slightly different, however, including a subtyping rule which implements record width and depth subtyping. The type soundness proof changes to include calls to reflexivity and transitivity lemmas about subtyping, which cannot be inferred by the solver.
- (`opaque-safe.dfy`): This file implements the first half of our technique, making the safety property opaque, proving compatability lemmas about the evaluator, and calling them in the soundness proof. The typechecker is identical to the one in `bidirectional.dfy`.
- (`opaque-safe-invert.dfy`): This file implements the full stabilizing technique, adding inversion lemmas to the code in `opaque-safe.dfy`, and making the typechecker itself opaque.
- (`std.dfy` and `util.dfy`): Auxiliary functions and definitions not specific to the language's evaluator or typechecker.

We run the experiment by running the built-in `dafny measure-complexity` command on the files containing each type soundness proof, with the flags `-iterations:250` and `-log-format csv`. Each invocation of `measure-complexity` command verifies the file 250 times, and dumps the verification results, durations, and resource usages to a CSV file. We then parse and analyze the CSV file with a python script, which computes the means and standard deviations of verification duration and resource usage for the solver's task of proving correctness of the VCs in the file's main soundness theorem. The experiments were run on a 2023 MacBook Pro with an Apple Silicon M2 processor, and 32GB memory, runnning Dafny v4.3.0, and z3 v4.12.1.

Results

The graph in Figure 8 shows the mean verification *resource usage* (and standard deviation, with whiskers), over the 250 verifications of each soundness theorem, while the graph in Figure 7 shows verification *durations*, in miliseconds. The two graphs tell essentially the same story. The highest cost and highest variance proof in all cases is the soundness proof for the typechecker in `basic.dfy`. It strikes an unfortunate balance of being complicated code — heavy use of monadic bind and lots of helper functions — with little guidance in the proof. The typechecker in `unfolded.dfy` is slightly cheaper to verify than the one in `basic.dfy`, but it still has enormous variance. The cost seems to be lower because of all of the inlined definitions, meaning that the verifier must reason about fewer functions while proving soundness. The proof for the typechecker in `bidirectional.dfy` is cheaper and

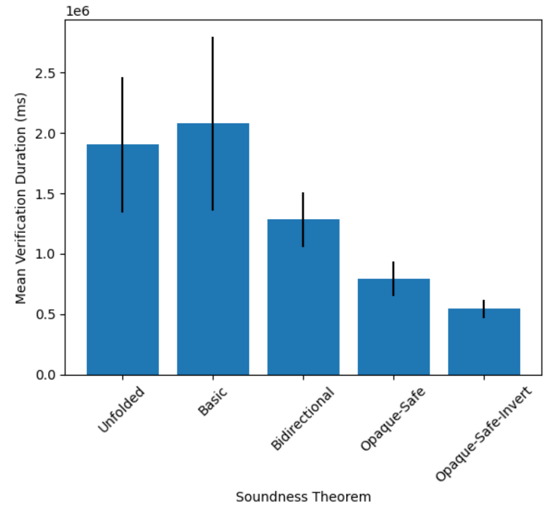

Figure 7. Verification Duration

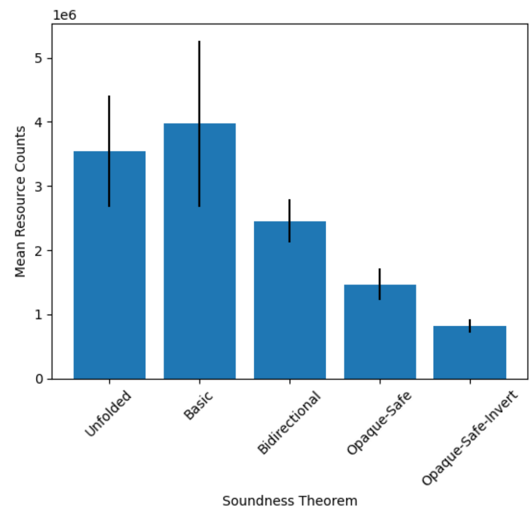

Figure 8. Verification Resource Usages

has lower variance than either of the previous two. This may be because while the bidirectionality increases the complexity of the typechecking algorithm, this complexity *requires* further guidance to the verifier for it to accept the proof at all, thereby improving stability on the whole. As expected, the two proofs using the first and second halves of our technique are by far the best. The soundness theorem using the fully specified typechecker and safety property (`opaque-safe-invert.dfy`) is the best of all, with negligable variance in resource usage between runs of the verifier.

4 Discussion

Scaling It Up. A version of this proof technique was developed for the purposes of mechanizing the proof of type soundness of the Cedar language [6] the language which underlies the Amazon Verified Permissions serivce [2]. The technique described in this abstract is currently used in the type soundness proof of the reference typechecker with respect to the reference interpreter. In Cedar, programs express access control policies, and the evaluator outputs an authorization decision: allow or deny. Cedar's type system is a great deal more complex than that of our toy langauge, and includes a number of advanced type system features like ocurrence typing [8] and singleton types [2].

Semantically though, Cedar is not *all* that much more complex to model than the toy language we evaluate for this extended abstract. Like our toy language, Cedar's evaluator is simplified by the fact that it is terminating, first order, and has no binders. Although we have not tested it, our technique should scale to languages with nontermination — by step-indexing the evaluator and proving type-safety with failure to converge in n steps being considered "safe" [1] — and higher-order functions — by applying standard tricks to handle variable binding.

Manual Effort & Benefit. All of this benefit does come at the cost of some of the automation that Dafny users expect. In taking possible work away from the solver to make the proof more stable, we are necessarily creating more work for ourselves! As we saw in the previous section, the type soundness proofs in the last two cases spell out many more steps along the way. Anecdotally, however, we have found it practically impossible to scale a proof that doesn't use this technique. So while this technique does require more manual effort, we have found it to be the only way to structure a type soundness proof that does not encounter impassable instability obstacles.

For language simpler than the one we evaluate here, the benefit of using this technique decreases. In particular, in languages where the safety property does not include a disjunction — safety means that every term evaluates to a value of the right type — the benefit shrinks dramatically. The kind of straight-line reasoning required when every term can only evaluate in one way seems to be much easier for the verifier.

References

[1] Andrew W. Appel and David McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-Carrying Code. *ACM Trans. Program. Lang. Syst.* 23, 5 (sep 2001), 657–683. https://doi.org/10.1145/504709.504712

[2] Susumu Hayashi. 1991. Singleton, Union and Intersection Types for Program Extraction. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software (TACS '91)*. Springer-Verlag, Berlin, Heidelberg, 701–730.

[3] K. R. M. Leino and Clément Pit-Claudel. 2016. Trigger Selection Strategies to Stabilize Program Verifiers. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 361–381.

[4] Mikael Mayer. 2023. *How to use Dafny to Prove Type Safety.* Accessed: 2023-10-17.

[2]https://aws.amazon.com/verified-permissions/

[5] John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference - Volume 2* (Boston, Massachusetts, USA) *(ACM '72)*. Association for Computing Machinery, New York, NY, USA, 717–740. https://doi.org/10.1145/800194.805852

[6] Amazon Web Services. 2023. Cedar Language. https://www.cedarpolicy.com/en. Accessed: 2023-10-17.

[7] Amazon Web Services. 2023. Cedar Language Dafny Reference. https://github.com/cedar-policy/cedar-spec/tree/main/cedar-dafny. Accessed: 2023-10-17.

[8] Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. *SIGPLAN Not.* 43, 1 (jan 2008), 395–406. https://doi.org/10.1145/1328897.1328486

[9] Aaron Tomb. 2023. Personal Communication.