ABSTRACT

Title of Dissertation: PROGRAM SYNTHESIS WITH LIGHTWEIGHT ABSTRACTIONS

Sankha Narayan Guria
Doctor of Philosophy, 2023

Dissertation Directed by: Professor David Van Horn
Professor Jeffrey S. Foster

The world is reliant on software systems of unprecedented scale, while our methods for developing software still require programmers to manually write code with little help toward ensuring the software correctly meets its intent. *Program synthesis*, which automatically generates correct programs from specifications, offers a hopeful path forward. While program synthesis has had many successes in recent years, these have mostly been in restricted domains; synthesis has not yet proved useful for the practicing software engineer.

This dissertation aims to advance program synthesis to meet the challenges posed by the use of modern general-purpose languages, tools, and frameworks. This dissertation presents work towards an automated programming stack that uses specifications and expressive test cases written by programmers to scale synthesis tools to diverse domains. Specifically, it demonstrates that types enriched with effect descriptors inferred from test cases are a potent means to guide the synthesis of real Ruby on Rails web apps, and that types enriched with logical predicates can be used to synthesize verified privacy preserving queries. The key to both projects, and most other successful synthesis work, is the proper choice of abstraction for the problem domain at hand. Based on this insight, this dissertation contributes a new synthesis framework that takes as a

parameter an abstract interpreter and automatically guides the search with it. This framework captures many different synthesis approaches from the literature, making it easier to build the synthesis tools of the future. The dissertation concludes with a vision for an automated programming stack that uses specifications and expressive test cases written by programmers to scale synthesis tools to diverse domains, moving us closer to a world in which correct programs are constructed automatically based on programmer's intent.

PROGRAM SYNTHESIS WITH LIGHTWEIGHT ABSTRACTIONS

by

Sankha Narayan Guria

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2023

Advisory Committee:
Professor David Van Horn, Co-chair/Co-advisor
Professor Jeffrey S. Foster, Co-chair/Co-advisor
Professor Lawrence Washington, Dean's Representative
Professor Michael Hicks
Professor Leonidas Lampropoulos

कर्मण्येवाधिकारस्ते मा फलेषु कदाचन ।
मा कर्मफलहेतुर्भूर्मा ते सङ्गोऽस्त्वकर्मणि ॥ ४७ ॥

Karmanye vadhikaraste Ma Phaleshu Kadachana,
Ma Karmaphalaheturbhurma Te Sangostvakarmani

You have the right to work only but never to its fruits.
Let not the fruits of action be your motive, nor let it be your attachment to inaction.
— Bhagavad Gita, Chapter 2, Verse: 47

Acknowledgements

It is unfortunate that a PhD is given to one person, when going through graduate school and doing research takes a village, and more so amidst a global pandemic.

I am indebted to my advisors David Van Horn and Jeff Foster for their unwavering support and encouragement in every new research idea I decided to pursue. David's uncanny ability to distill hard technical problems down to core simple ideas inspired me. He has guided me to write and present my ideas effectively. I still cherish the semester we spent reading papers about verification of neural networks. Though that effort went nowhere, it was an important lesson that not all research projects will be successful, but it is still worth trying. Jeff is a person who cares foremost about problems, rather than holding himself to one technique. He has repeatedly given me the perspective to pursue the right research ideas and the right insights needed to overcome hurdles that a research project invariably presents. Most notably, he continued to advise me remotely even after he moved to Tufts, and took out time to teach me invaluable skills in writing and articulation. I will consider myself successful if I can be a fraction of the mentor to someone as David and Jeff were to me.

I would like to acknowledge my committee – Mike Hicks, Leo Lampropoulos, and Larry Washington for their help in shaping this dissertation. A special thanks goes to Mike and Leo, who have given me valuable advise not only on research, but also on career at various points through my time in grad school. Mike taught me the first programming languages class, and without him my programming language foundations

would not be where it is today.

I would also like to thank Niki Vazou and Marco Guarnieri who were fun collaborators. Niki's resourcefulness and Liquid Haskell wizardry continues to surprise me to this day. Needless to say, whatever Haskell I know is because of her. Marco helped me understand a research area in which I had no prior experience. I have benefited immensely from his patient and kindness.

I am grateful to my PLUM friends for thoughtful conversations, camaraderie, and last minute presentation feedback, without which grad school would not have been possible. Thanks to Milod, who quickly got me upto speed on doing research and being kind to answer any questions I had along the way. Ian, James, and Kesha are my academic siblings. PLUM would not have been the same without you! Shoutout to Alp, Ben Mariano, Ben Quiring, Deena, Finn, Henry, Jacob, Justin, Mingwei, Phil, Pierce, and Yiyun — our conversations about programming languages and the world has always left me feeling enriched.

My undergraduate friends, who are now spread across six timezones were a great source of support. While COVID-19 pandemic caused the entire world to shutdown, our regular calls were always mentally rejuvinating. Thank you Amruta, Divya, Deekshitha, Jenil, Kedar, Rakshit, Sanchit, and Srikar! It definitely helped that someone was always awake somewhere on earth. A few other friends — Tagde, Hatte, Pushon, and Prajanma for the endless laughs on our regular catch-up conversations.

Life in grad school would have been impossible without friends. Thanks to Alejandro, Devesh, Jack, Nirat, and Nishant for giving company in hikes and board game sessions. Shoutout to Adway, Ahana, Bidisha, Darshan, Gargi, Kunal, Nikhil, Radhika, Rishov, Roni, Spandan, Upamanyu, Veeresh, and some more people I am invariably missing for their friendship. I truly cherish my time in College Park because of you!

To my family—Maa, Baba, and Kutu—thank you for your boundless love, gen-

erosity, strength, and encouragement. Moving halfway across the world for a PhD is a truly an insane thing to do. Thank you for believing in me, and encouraging me to be resilient in the face of adversity.

Finally, to Sohini. I enjoyed the work I did over the last few years, but I enjoyed coming back home to be with you even more. Thank you for your love, encouragement, patience, and support; I could not have done it without you.

Table of contents

Chapter 1

Introduction

Programming has become an essential skill for an increasing number of people ranging from data scientists to spreadsheet users. The need for programmers is driven by our world's increasing reliance on software systems. These systems have grown to an unprecedented scale, while our methods for developing software still require expert programmers to manually write code with little help for ensuring software correctly meets its intent. *Program synthesis*, which automatically generates correct programs from specifications, offers a hopeful path forward. It enables non-experts to write code from familiar input/output examples, and professional programmers to describe high-level properties and automatically generate a program that exhibits them without worrying about implementation details.

Synthesis tools need to overcome two primary challenges before they can be practical. The first is *intractability*, i.e., synthesis tools must search through the enormous space of possible programs efficiently to find the program that the user intended. The second is enabling the *specfications* written by the user to be easier to write than the intended program. A popular solution to both these problems is the domain specialization of synthesis tools, which works by effectively reducing the space of programs to specify and that the synthesizer has to search through. For example, the

FlashFill [45, 47] system in Microsoft Excel synthesizes data transformation programs in a specialized domain-specific language (DSL) using input/output examples or Falx [109] synthesizes data visualizations from a visualization sketch and a user dataset.

Much of the prior work, however, requires a complete and accurate embedding of the source language in the logic of the underlying solver the synthesis tool uses. Such synthesis tools use techniques often ranging from symbolic execution [102], counter-example guided synthesis [97], eliminating classes of programs that compute the same value [1], or over-approximate semantics as predicates [58, 80, 32] (often requiring termination measures and additional predicates for verification). Such precise embeddings of source language is infeasible for many industrial-grade languages such as Ruby or Python used by software engineers or data scientists in their daily workflow. Other prior work is strongly coupled with the semantics of the source language using purpose-built solvers like CVC4 [90] for SyGuS, but this coupling necessarily ties the synthesis engine to the particular language model used.

This dissertation accepts the reality that behavior of programs is hard to specify, and proposes to use lightweight specifications users already write. It proposes to use simple abstractions such as types to efficiently guide the search for programs. More generally, we show that such a program search can be guided by any *abstract interpreter* [25], i.e., a tool that soundly approximates program behavior. However, to verify correctness, it proposes testing the programs in the canonical concrete intrepreter of a language. Abstractions can be coarse- or fine-grained as long as they guide the search. Testing, on the other hand, guarantees correctness for programs that use arbitrary libraries which lack precise formal models. In summary, this dissertation aims to demonstrate that:

> *Techniques from abstract interpretation can be combined with program testing to build program synthesis tools using lightweight specifications, to*

generate correct programs in languages or libraries of choice.

1.1 Synthesis of Effectful Programs

A key task in modern software development is writing code that composes calls to existing APIs, such as from a library or framework. *Component-based synthesis* aims to carry out this task automatically, and researchers have shown how to perform component-based synthesis using types [31] or special properties of the synthesis domain [55], which is critical to achieving good performance. However, prior work does not explicitly consider side effects, which are pervasive in many domains. For example, consider web apps, a centerpiece of daily online interactions. Such apps act as the liaison between the database, network, and global state of the program itself. Any operation to read or modify data from such an external systems are side effects. However, these apps are not amenable to formal specifications primarily due to the overhead of formally modeling external systems like the database or the network. Such apps often relying on type systems and testing to enforce correctness, making it hard to use existing synthesis methods.

We address this limitation of applying synthesis to such web apps by introducing RbSyn [49], a new tool for synthesizing Ruby methods. In RbSyn, the user specifies the desired method by its type signature and a series of test cases it must pass. RbSyn then searches for a solution by enumerating candidates and checking them against the tests. The key novelty of RbSyn is that the search is both *type- and effect-guided*. Specifically, the search begins with a *typed hole*, i.e., a placeholder program, tagged with the method's return type. Each step either replaces a typed hole with an expression of that type, possibly introducing more typed holes; inserts an effect hole, annotated with a write effect that may be needed to satisfy a test assertion; or replaces an effect hole with an expression with the given write effect,

possibly inserting another effect hole. Once this process finds a set of method bodies that cumulatively pass all tests, RbSyn uses a novel merging strategy to construct a complete solution: it creates a method whose body branches among the conditions, executing the corresponding (passing) code, thus yielding a single method that passes all tests.

We observe, in practice, programmers test an effectful method by triggering a complementary side-effect in their test. For example, to check if a method *writes* to a database correctly, a corresponding test will *read* from that database location to assert expected behavior. This led to the key insight that test executions can be monitored for errors to automatically infer the effect holes and their desired effect labels. We formalized this inference, and implemented a practical tool for synthesizing programs in Ruby. RbSyn was evaluated by synthesizing database model methods from production-grade Ruby applications like GitLab, Discourse, and Diaspora using their original tests. Our evaluation showed effect guidance is useful for outperformance when compared to just type-guided synthesis. Additionally, RbSyn synthesizes `if-then-else` branches using unit tests, often used to encode data validation checks and business logic in web apps. The branches synthesized by RbSyn are at parity with code written by software engineers of those projects. Moreover, RbSyn's lightweight effects allows programmers to explore a spectrum between highly precise to very coarse grained effects–allowing flexibility to tune the synthesis performance based on specification burden vs. synthesis time budget, while still being correct because of testing.

1.2 Synthesizing Privacy Preserving Queries

While RbSyn with its coarse grained type-and-effects is a good fit for database access methods in web apps, there are domains (like security) that are safety-critical and

require precision and stronger correctness guarantees. For example, a common security task is to ensure that private or sensitive data cannot be accessed by unauthorized third parties, a property formalized as *non-interference* [41]. Information flow control (IFC) [92] systems are designed to protect the confidentiality of such secrets during program execution thus enforcing non-interference. However, in practice, non-interference is too strong for most programs, as most occasionally need to reveal information about sensitive data. For example, a password check routine needs to display the result of checking if the user-input password is correct or not, which leaks a bit of information about the password. To support such use cases, programmers use *declassification* statements in IFC applications that weaken non-interference by allowing selective disclosure. Declassification statements, however, are typically part of an application's trusted computing base and developers are responsible for properly vetting them. In particular, mistakes in declassifications can easily compromise a system's security because declassified information bypasses non-interference checks.

To address this problem, we design ANOSY [50], a framework for enforcing declassification policies that regulate what information can be declassified by limiting the amount of information an attacker could learn from the declassification statements. Specifically, declassification policies are expressed as constraints over *knowledge*, which semantically characterizes the set of secrets an attacker considers possible given the observations. To enforce such policies, first we develop a novel encoding of attacker knowledge approximations using LiquidHaskell's refinement types to produce machine-checked proofs of correctness. Second, the constraints generated by these refinement types are combined with numerical optimization in ANOSY to automatically synthesize functions to compute correct-by-construction knowledge approximations for queries on secret data. The key innovation of ANOSY is the synthesis of a function that given any prior knowledge of the attacker, computes the attacker's posterior knowledge if the query on secrets were to be executed. As a result, such a function can be directly

integrated in the application, eliminating the need to run an expensive static analysis (such as Prob [68]) for each query computation.

ANOSY was evaluated using domains such as intervals [24] or powersets of intervals [84, 10] on diverse set of queries from past work inspired by targeted advertising on Facebook. ANOSY's synthesis being a compile time operation has a one time upfront cost, but the subsequent estimation of adversary's knowledge incurs no cost. In contrast, tools like Prob need to run an expensive static analysis each time adversary knowledge is to be computed. Moreover, ANOSY's synthesis algorithm is more precise for queries containing disjunctions than Prob for higher precision domains like powersets. ANOSY ships with a monad, `AnosyT` that allows safe declassifications to guarantee end-to-end policy compliance in applications. We show that such declassifications support sequence of multiple queries to be answered securely in an end-to-end application, while the precision and time taken for synthesis can be tuned using the cardinality of powersets.

1.3 Generalized program search with abstract interpretation

RBSYN and ANOSY both are separate tools that rely on very different abstractions to guide the synthesis search. Their search algorithms are inherently tied to the abstract domain defintion and the semantics. In general, developing a synthesis tool requires designing the search algorithm tied to the language semantics using abstractions of our choice. However, often the key insight is in designing the right abstraction for the problem domain, like types and effects for Ruby programs or refinement types for Haskell programs. A natural question arises: is there a better way to design synthesis tools that can guide the search based on abstraction while keeping the search algorithms independent of the used domain?

To answer this question, we develop ABSYNTHE [48], an approach based on user-defined abstract semantics that aims to be both lightweight and language agnostic. The synthesis engine is parameterized by the abstract semantics and independent of the source language. In ABSYNTHE, users define a synthesis problem via concrete test cases and an abstract specification in some user-specified abstract domain. These abstract domains, and the semantics of the target language in terms of the abstract domains, are written by the user in a domain-specific language. Moreover, the user can define multiple simple domains, each defining a partial semantics of the language, which they can combine together as a product domain automatically. ABSYNTHE uses these abstract specifications to automatically guide the search for the program using the abstract semantics. The abstract semantics are lightweight to design, simplifying away inconsequential language details, yet effective in guiding the search for programs.

The key novelty of ABSYNTHE is that it separates the search procedure from the definition of abstract domains, allowing the search to be guided by any user-defined domain that fits the synthesis task. More specifically, the program search in ABSYNTHE begins with a hole tagged with an abstract value representing the method's expected return value. At each step, ABSYNTHE substitutes this hole with expressions, potentially containing more holes, until it builds a concrete expression without any holes. Each concrete expression generated is finally tested in the reference interpreter to check if it passes all test cases. A program that passes all test cases is considered the solution.

For a baseline comparison against other tools, we evaluated ABSYNTHE on the SyGuS strings benchmark—a standard program synthesis benchmark in a small functional language. Unlike tools like CVC4 [89] (when used in SyGuS synthesizer mode), that have complete background theory for strings and linear integer arithmetic, ABSYNTHE works with more lightweight semantics like string length, string prefix, and string suffix domains. It does not have any additional semantic knowledge of integers

or strings outside these domains. Moreover, ABSYNTHE's guidance using the abstract semantics allows it to outperform other enumerative synthesis tools designed for SyGuS programs [3]. ABSYNTHE also allows the programmer to mix-and-match abstract domains for each synthesis task based on the expressiveness of the domains used for specification. This allows the user to move on a performance and expressiveness spectrum, where using a simple domain searches through more programs but a more expressive domain prunes more programs. The generalizability of ABSYNTHE to different synthesis problems is evaluated by synthesizing benchmarks unrelated to SyGuS. We use the AUTOPANDAS benchmark [13]—a suite of Python data frame manipulation programs using the Pandas library sourced from StackOverflow questions. Pandas data frames are commonly used by data scientists for data wrangling before any downstream analysis is done. AUTOPANDAS uses a graph neural network models trained on dedicated hardware (4 Nvidia Titan V GPUs) over a 48 hour period to solve 17 out of 26 benchmarks. In contrast, ABSYNTHE solves the same number of benchmarks (with some overlap in the set of benchmarks solved) using just a type system and sets of data frame column labels on a 2016 Macbook Pro.

We believe ABSYNTHE represents an important step forward in the design of practical synthesis tools that provide lightweight formal guarantees while ensuring correctness from tests.

Chapter 2

RbSyn: Type- and Effect-Guided Program Synthesis

2.1 Introduction

A key task in modern software development is writing code that composes calls to existing APIs, such as from a library or framework. *Component-based synthesis* aims to carry out this task automatically, and researchers have shown how to perform component-based synthesis using SMT solvers [55]; how to synthesize branch conditions [78]; and how to perform synthesis given a very large number of components [31].

This prior work guides the synthesis process using types or special properties of the synthesis domain, which is critical to achieving good performance. However, prior work does not explicitly consider *side effects*, which are pervasive in many domains. For example, consider synthesizing a method that updates a database. Without reasoning about effects—in this case, that the method body needs to change the database—synthesis of such a method reduces to brute-force search, limiting its performance.

In this chapter, we address this issue by introducing RbSyn, a new tool for

synthesizing Ruby methods. In RbSyn, the user specifies the desired method by its type signature and a series of test cases it must pass. RbSyn then searches for a solution by enumerating candidates and checking them against the tests. The key novelty of RbSyn is that the search is both *type- and effect-guided*. Specifically, the search begins with a *typed hole* tagged with the method's return type. Each step either replaces a typed hole with an expression of that type, possibly introducing more typed holes; inserts an *effect hole*, annotated with a write effect that may be needed to satisfy a test assertion; or replaces an effect hole with an expression with the given write effect, possibly inserting another effect hole. Once this process finds a set of method bodies that cumulatively pass all tests, RbSyn uses a novel merging strategy to construct a complete solution: It creates a method whose body branches among the conditions, executing the corresponding (passing) code, thus yielding a single method that passes all tests. (§ 2.2 gives a complete example of RbSyn's synthesis process.)

We formalize RbSyn for λ_{syn}, a core object-oriented language. The synthesis algorithm is comprised of three parts. The first part, type-guided synthesis, is similar to prior work [76, 37, 80], but is geared towards imperative, object-oriented programs. The second part is *effect-guided synthesis*, which tries to fill an effect hole $\Diamond : \epsilon$ with an expression with effect ϵ. In λ_{syn}, an effect accesses a *region* $A.r$, where A is a class and r is an uninterpreted identifier. For example, `Post.author` might indicate reading instance field `author` of class `Post`. This notion of effects balances precision and tractability: effects are precise enough to guide synthesis effectively, yet coarse enough that reasoning about them is simple. The last part of the synthesis algorithm synthesizes branch conditions to create a *merged* program that combines solutions for individual tests into an overall solution for the complete problem. (§ 2.3 discusses our formalism.)

Our implementation of RbSyn is built on top of RDL, a Ruby type system [36]. Our implementation extends RDL to include effect annotations, including a `self`

region to give more precise effect information in the presence of inheritance. Our implementation also makes use of RDL's *type-level computations* [57] to provide precise typing during synthesis. Finally, when searching for solutions, our implementation heuristically prioritizes further exploration of candidates that are small and have passed more assertions. (§ 2.4 describes our implementation.)

We evaluated RbSyn on a suite of 19 benchmarks, including seven benchmarks we wrote and 12 benchmarks extracted from three widely used, open-source Ruby apps: Discourse, Gitlab, and Disaspora. For the former, we wrote our own specifications. For the latter, we used unit tests that came with the benchmarks. We found that RbSyn synthesizes correct solutions for all benchmarks and does so quickly, taking less than 9 seconds each for 15 of the benchmarks, and 83 seconds for the slowest benchmark. Moreover, type- and effect-guidance is critical. Without it, a majority of the benchmarks time out after five minutes. Finally, we examine the tradeoff of effect precision versus performance. We found that restricting effects to class names only causes 3 benchmarks to time out, and restricting effects to only purity/impurity causes 10 benchmarks to time out. (§ 2.5 discusses the evaluation in detail.)

We believe that RbSyn is an important step forward in synthesis of effectful methods from test cases.

2.2 Overview

In this section, we illustrate RbSyn by using it to synthesize a method from a hypothetical web blogging app. This app makes heavy use of ActiveRecord, a popular database access library for Ruby on Rails. It is the ActiveRecord methods whose side effects RbSyn uses to guide synthesis.

Figure 2.1 shows the synthesis problem. This particular app includes database tables for users and posts. In ActiveRecord, rows of these tables are represented as

```
# User schema {name: Str, username: Str}
# Post schema {author: Str, title: Str, slug: Str}

define :update_post, "(Str, Str, {author: ?Str, title: ?Str, slug:
    ?Str}) → Post", [User, Post] do

  spec "author can only change titles" do
    setup {
      seed_db # add some users and their posts to db
      @post = Post.create(author: 'author', slug: 'hello-world', title:
        'Hello World')
      update_post('author', 'hello-world', author: 'dummy', title:
        'Foo Bar', slug: 'foobar')
    }
    postcond { |updated|
      assert { updated.id == @post.id }
      assert { updated.author == "author" }
      assert { updated.title == "Foo Bar" }
      assert { updated.slug == 'hello-world' }
    }
  end

  spec "other users cannot change anything" do
    setup { ... # same setup as above except next line
      update_post('dummy', ...) # other args same
    }
    postcond { |updated| ... # same other three asserts
      assert { updated.title == "Hello World" }
    }
  end
end
```

Figure 2.1: Specification for `update_post` method

instances of classes `User` and `Post`, respectively. For reference, the table schemas are shown in lines 1 and 2. Each user has a `name` and `username`. Each post has the `author`'s username, the post's `title`, and a `slug`, used to compute a permalink.

The goal of this particular synthesis problem, given by the call to `define`, is to create a method `update_post` that allows users to change the information about a post. Lines 4 and 5 specify the method's type signature in the format of RDL [36], a Ruby type system that RbSyn uses for types and type checking. Here, the first two arguments are strings, and the last is a *finite hash type* that describes an instance of `Hash` with optional (indicated by `?`) keys `author`, `title`, and `slug` (all *symbols*, which are just interned strings) that map to strings. The method itself returns a `Post`.

In addition to the type signature, the synthesis problem also includes a list of constants that can be used in the target method. In this case, those constants are the classes `User` and `Post`, as given by the last argument to `define` on line 5. These classes can then be used to invoke singleton (class) methods in the synthesized method. For simplicity, we assume that RbSyn can use only these constants for this example. In practice, RbSyn can synthesize predefined numeric or string constants like 0, 1 or the empty string.

Finally, the synthesis problem includes a number of *specs*, which are just test cases. Each spec has a title, for human convenience; a `setup` block to establish any necessary preconditions and call the synthesized method; and a `postcond` block with assertions that must hold after the synthesized method runs. As we will see below, separating the pre- and postconditions allows RbSyn to more easily use effects to guide synthesis. In this example, both specs add a few users and a post created by each of them to the database (call to `seed_db`, details not shown) and then create a post titled "Hello World" by the user `author`. The first spec asserts that `update_post` allows `author` to update a post's title. The second spec asserts that a `dummy` user cannot update the post. The check for `id` ensures that only existing posts are updated (any new posts

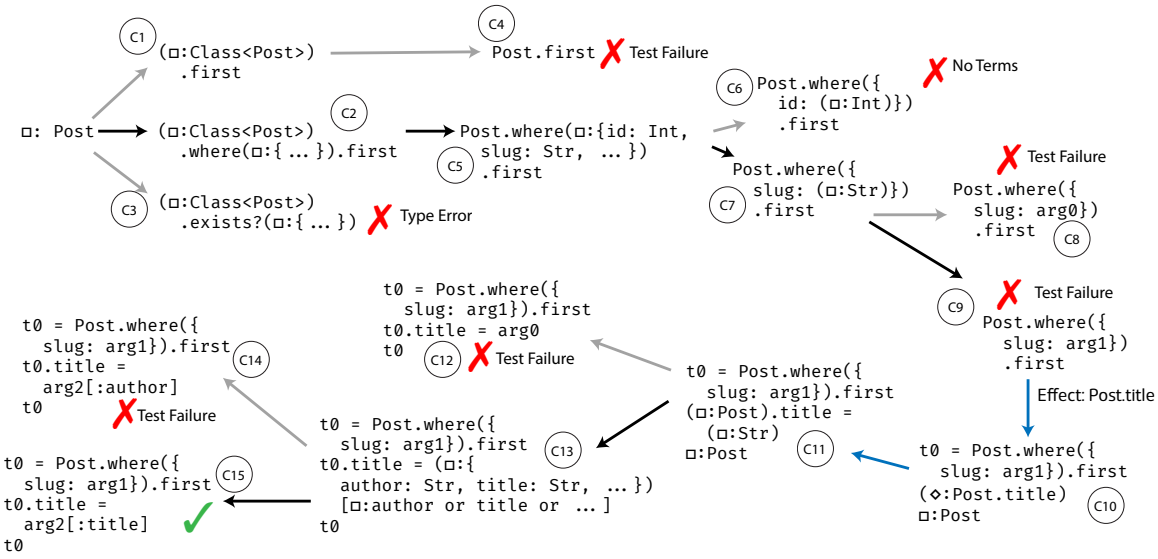


Figure 2.2: Steps in the synthesis of solution to the first specification. Note C2 takes two steps to synthesize but is shown as a single composite step. Some choices available to the synthesis algorithm have been omitted for simplicity.

will have a new unique id).

The final, synthesized solution is shown in Figure 2.3. Notice the synthesized code calls several ActiveRecord methods (`exists?`, `where`, and `first`) as well as the hash access method `[]`. Applying solver-aided synthesis to this problem would require developing accurate models of these methods, which is a difficult challenge [71]. To address this limitation, RbSyn instead enumerates candidates, which can then be run to check them against the specs. As the search space is vast, RbSyn uses `update_post`'s type signature and the effects from the specs' `postcond`s the guide the search. Finally, RbSyn uses a novel merging algorithm to synthesize the necessary branch condition to yield a solution that satisfies both specs.

2.2.1 Synthesizing Spec Solutions

Figure 2.2 shows the search process RbSyn uses to solve this synthesis problem. To begin, RbSyn observes that the return type of `update_post` is `Post`. Thus, the search begins (upper left) by creating a candidate method body □:`Post`, which is a

```
def update_post(arg0, arg1, arg2)
  if Post.exists?(author: arg0, slug: arg1)
    t0 = Post.where(slug:arg1).first
    t0.title=arg2[:title]
    t0
  else
    Post.where(slug: arg1).first
  end
end
```

Figure 2.3: Synthesized `update_post` method.

typed hole that must be filled by an expression of type `Post`. RbSyn then iteratively expands holes in candidates, running the specs whenever it produces fully concretized candidates with no holes.

In general, RbSyn can fill a typed hole with a local variable, a constant, or a method call. As there are no local variables (which so far are just parameters) or constants of the appropriate type, RbSyn chooses a method call. To do so, it searches through the available method type annotations to find those that could return `Post`. In this case, RbSyn takes advantage of RDL's type annotations for ActiveRecord [57] to synthesize candidates `C1` and `C2`, among others (not shown). It is straightforward for the user to add type annotations for any other library methods that might be needed by the synthesized method. For illustration purposes, we also show a candidate `C3` that returns the wrong type. Such candidates are discarded by RbSyn, vastly reducing the search space. Note that C2 contains two method calls, and thus would take two steps to produce, but we show it here as a single step for conciseness.

Next, RbSyn tries to fill holes in candidate expressions, starting with smaller candidates. In this case, it first considers `C1`, which has a hole of type `Class<Post>`, which is the singleton type for the constant `Post`. Thus, there is only one choice for the hole, yielding candidate `C4`. Since `C4` has no holes, RbSyn runs it against the specs. More specifically, it runs it against the first spec—as we will discuss shortly, RbSyn synthesizes solutions for each spec independently, and then combines them.

In this case, C4 fails the spec (because the first post in the database is not the one to be updated, due to the initial database seeding) and hence is rejected.

Continuing with C5, RbSyn fills in the (finite hash-typed) hole, yielding choices that include C6 and C7. RbSyn rejects C6 since there is no way to construct an expression of type `Int`. However, for C7, there are two local variables of type `Str` from the method arguments. Substituting these yields C8 and C9. C8 uses `arg0`, the username, to query the `Post` table's slug, so it fails. C9 queries the `Post` table with the correct slug value `arg1`. This passes the first two assertions (line 16 onwards) but fails the third, which expects the post title to be updated from "Hello World" to "Foo Bar."

RbSyn extends RDL's type annotations to include read and write effects. When the expression inside an `assert` evaluates to `false`, RbSyn infers the `assert`'s read and write effects based on those of the methods it calls. For example, we can give the `Post#title`[1] method, used by the third assertion, the following signature:

```
type Post, :title, '() → Str', read: ['Post.title']
```

Thus, RbSyn sees that the failing assertion reads `Post.title`, an abstract effect label. To make the assertion succeed, RbSyn inserts an *effect hole* $\lozenge$: `Post.title` in the candidate program (C10). It also saves the value of the previous candidate expression in a temporary variable, and inserts a hole with the candidate's type at the end. RbSyn then continues the search, trying to fill the effect hole with a call to a method whose *write* effect matches the hole—such a call could potentially satisfy the failed assertion. Here, RbSyn replaces the effect hole (C11) with a call to `Post#title=`, which is such a method. (We should note that all previous candidates that failed a spec due to a side effect will also have effect holes added in a similar fashion. We omit these candidates from the discussion as they do not lead to a solution.)

RbSyn continues by using type-guided synthesis for the typed holes of C11—

[1] `A#m` indicates instance method `m` of class `A`.

yielding C12, rejected due to assertion failures—and then C13. After several steps (not shown), RbSyn arrives at C14, which fails the spec, and C15, which fully satisfies the first spec. Indeed, we see this exact expression in lines 3–5 of the solution in Figure 2.3.

2.2.2 Merging Solutions

RbSyn next uses the same technique to synthesize an expression that satisfies the second spec, yielding the expression shown on line 7. Now RbSyn needs to merge these individual solutions into a single solution that passes all specs. At a high-level, it does so by constructing a program `if` b_1 `then` e_1 `else if` b_2 `then` e_2 `end`, where the e_i are the solutions for the specs and the b_i are *branch conditions* capturing the conditions under which those expressions pass the specs.

To create the b_i, RbSyn uses the same technique again, this time synthesizing a boolean-valued expression that evaluates to `true` under the `setup` of spec i. In this case, this process results in the same branch condition `true` for both specs. However, since this trivially holds for both specs, this branch condition does not work—we need to find a branch condition that distinguishes the two cases.

Next RbSyn tries to synthesize a branch condition b_1' that evaluates to `true` for the `setup` of the first spec and `false` for the `setup` of the second. This yields the more precise branch condition b_1' `= Post.exists?(author: arg0, slug: arg1)`. This is a sufficient condition, as the `update_post` method is supposed to update a post only if a post with slug `arg1` is authored by `arg0`. It solves an analogous synthesis problem for the second spec, yielding b_2' `= !Post.exists?(author: arg0, slug: arg1)`. As these are the negation of each other, RbSyn then merges these two together as `if-then-else` (rather than an `if-then-else if-then-else`), yielding the final synthesized program in Figure 2.3.

$$
\begin{array}{llll}
\textit{Values} & v & ::= & \texttt{nil} \mid \texttt{true} \mid \texttt{false} \mid [A] \\
\textit{Expressions} & e & ::= & v \mid x \mid e; e \mid e.m(e) \\
 & & \mid & \texttt{if}\ b\ \texttt{then}\ e\ \texttt{else}\ e \\
 & & \mid & \texttt{let}\ x = e\ \texttt{in}\ e \mid \square : \tau \mid \lozenge : \epsilon \\
\textit{Conditionals} & b & ::= & e \mid !b \mid b \vee b \\
\textit{Types} & \tau & ::= & A \mid \tau \cup \tau \\
\textit{Programs} & P & ::= & \texttt{def}\ m(x) = e \\
 & & & \\
\textit{Specs} & s & ::= & \langle S, Q \rangle \\
\textit{Setup} & S & ::= & e;\ x_r = P(e) \\
\textit{Postconditions} & Q & ::= & \texttt{assert}\ e \mid Q; Q \\
\textit{Spec Set} & \Psi & := & \{s_i\} \\
\textit{Synthesis Goal} & G & ::= & \langle \tau \to \tau, \Psi \rangle \\
 & & & \\
\textit{Class Table} & CT & ::= & \emptyset \mid A.m : \sigma, CT \\
\textit{Method Types} & \sigma & ::= & \tau \xrightarrow{\langle \epsilon_r, \epsilon_w \rangle} \tau \\
\textit{Type Env.} & \Gamma & ::= & \emptyset \mid x : \tau, \Gamma \\
\textit{Dynamic Env.} & E & ::= & x \to v \\
\textit{Constants} & \Sigma & ::= & \emptyset \mid v : \tau, \Sigma \\
\textit{Effect} & \epsilon & ::= & \bullet \mid * \mid A.* \mid A.r \mid \epsilon \cup \epsilon
\end{array}
$$

$$
r \in \text{effect regions} \quad \bullet \subseteq \epsilon \quad \epsilon \subseteq *
$$

$$
A_1.* \subseteq A_2.* \text{ and } A_1.r \subseteq A_2.r \text{ and } A_1.r \subseteq A_2.* \text{ if } A_1 \leq A_2
$$

$$
\epsilon^1 \subseteq \epsilon^1 \cup \epsilon^2 \quad \epsilon^2 \subseteq \epsilon^1 \cup \epsilon^2
$$

$$
\langle \epsilon_r^1, \epsilon_w^1 \rangle \cup \langle \epsilon_r^2, \epsilon_w^2 \rangle = \langle \epsilon_r^1 \cup \epsilon_r^2, \epsilon_w^1 \cup \epsilon_w^2 \rangle
$$

$$
x \in \text{variables}, \ m \in \text{methods}, \ A \in \text{classes},
$$

$$
\mathit{Nil} \leq \tau \quad \tau \leq \mathit{Obj} \quad \tau_1 \leq \tau_1 \cup \tau_2 \quad \tau_2 \leq \tau_1 \cup \tau_2
$$

Figure 2.4: Syntax and Relations of λ_{syn}.

2.3 Formalism

In this section, we formalize RbSyn on λ_{syn}, a core object-oriented calculus shown in Figure 2.4. Values v include `nil`, `true`, `false`, and objects $[A]$ of class A. Note that we omit fields to keep the presentation simpler. Expressions e include values, variables x, sequences $e; e$, method calls $e.m(e)$, conditionals `if` b `then` e `else` e, and variable bindings `let` $x = e$ `in` e. A conditional guard b can be an expression e, a negation $!b$, or a disjunction $b \vee b$. The grammar for guards is limited to match what

RbSyn can actually synthesize.

Expressions also include *typed holes* $\square : \tau$ and *effect holes* $\lozenge : \epsilon$, which are placeholders that are eventually filled with an expression of the given type, or expression with the given write effect, respectively. We note our synthesis algorithm only inserts effect holes at positions that can have any type. Types are either classes or unions of types, and we assume classes form a lattice with *Nil* (the class of `nil`) as the bottom element and *Obj* as the top element. We write $A \leq B$ when class A is a subclass of B according to the lattice. We defer the definition of effects for the moment. Finally, a synthesized program P is a single method definition `def` $m(x) = e$. We restrict the method to one argument for convenience.

A spec s in λ_{syn} is a pair of setup code S and a postcondition Q. A setup $e_1; x_r = P(e_2)$ includes some initialization e_1 followed by a special form indicating calling the synthesized method in P with argument e_2 and binding the result to x_r. The postcondition is a sequence of assertions that can test x_r and inspect the global state using library methods. We write Ψ for a set of specs, and a *synthesis goal* G is a pair $\langle \tau_1 \to \tau_2, \Psi \rangle$, where τ_1 and τ_2 are the method's domain and range types, respectively, and Ψ are the specs the synthesized method should satisfy.

The next part of Figure 2.4 defines additional notation used in the formalism. Synthesized methods can use classes and methods from a *class table* CT, which maps class and method names to the methods' types. For example, the class table has type information for other methods of a target app and library methods such as those from ActiveRecord. A method type σ has the form $\tau \xrightarrow{\langle \epsilon_r, \epsilon_w \rangle} \tau'$, where τ and τ' are the domain and range types, respectively, and $\langle \epsilon_r, \epsilon_w \rangle$ specifies the method's read effect ϵ_r and write effect ϵ_w (discussed shortly). During type-guided synthesis, RbSyn maintains a type environment Γ mapping variables to their types. When executing a synthesized program, the operational semantics (omitted) uses a dynamic environment E mapping variables to their values. During synthesis, Σ is a list of user-supplied

constants that can fill holes.

Effects. The last part of Figure 2.4 defines effects ϵ. In RbSyn, effects are hierarchical names that abstractly label the program state. The empty effect $\bullet$ denotes no side effect, used for pure computations. The effect $*$ is the top effect, indicating a computation that might touch any state in the program. Lastly, effect $A.*$ denotes code that touches any state within class A, and $A.r$ denotes code that touches the region labeled r in A, where region names are completely abstract. Effects can also be unioned together.

We define subsumption $\epsilon_1 \subseteq \epsilon_2$ on effects to hold when ϵ_2 includes ϵ_1. Effects $\bullet$ and $*$ are the bottom and top, respectively, of the $\subseteq$ relation, and if $A_1 \leq A_2$ then $A_1.r \subseteq A_2.r$ and $A_1.r \subseteq A_2.*$ and $A_1.* \subseteq A_2.*$. We also have standard rules for subsumption with effect unions.

In RbSyn, all effects arise from calling methods from the class table CT, which have effect annotations of the form $\langle \epsilon_r, \epsilon_w \rangle$, where ϵ_r and ϵ_w are the method's read and write effects, respectively. We extend subsumption to such paired effects in the natural way. During synthesis, if RbSyn observes the failure of an assertion with some read effect ϵ_r, it tries to fix the failure by inserting a call to some method with write effect ϵ_w such that $\epsilon_r \subseteq \epsilon_w$, i.e., it tries writing to the state that is read. For example, in Section 2.2, this technique generated a call to `Post#title`.

Our effect language is inspired by the region path lists approach of [15], but is much simpler. We opted for coarse-grained, abstract effects to make it easier to write annotations for library methods. Although class names are included in the effect language, such names are for human convenience only—nothing precludes a method in class A being annotated with an effect to $B.r$ for some other class B. We found that this approach works well for our problem setting of synthesizing code for Ruby apps, where trying to precisely model heap and database state would be difficult.

$$\boxed{\Sigma, \Gamma \vdash_{CT} e \rightsquigarrow e : \tau}$$

$$\frac{\Gamma(x) = \tau}{\Sigma, \Gamma \vdash_{CT} x \rightsquigarrow x : \tau} \text{ T-Var}$$

$$\frac{\begin{array}{c}\Sigma, \Gamma \vdash_{CT} e_1 \rightsquigarrow e_1' : \tau_1 \\ \Sigma, \Gamma[x \mapsto \tau_1] \vdash_{CT} e_2 \rightsquigarrow e_2' : \tau_2\end{array}}{\Sigma, \Gamma \vdash_{CT} \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \rightsquigarrow \texttt{let}\ x = e_1'\ \texttt{in}\ e_2' : \tau_2} \text{ T-Let}$$

$$\frac{}{\Sigma, \Gamma \vdash_{CT} \square : \tau \rightsquigarrow (\square : \tau) : \tau} \text{ T-Hole} \qquad \frac{v : \tau_1 \in \Sigma \qquad \tau_1 \leq \tau_2}{\Sigma, \Gamma \vdash_{CT} \square : \tau_2 \rightsquigarrow v : \tau_1} \text{ S-Const}$$

$$\frac{\Gamma(x) = \tau_1 \qquad \tau_1 \leq \tau_2}{\Sigma, \Gamma \vdash_{CT} \square : \tau_2 \rightsquigarrow x : \tau_1} \text{ S-Var}$$

$$\frac{m : \tau_1 \to \tau_2 \in CT(A) \qquad \tau_2 \leq \tau_3}{\Sigma, \Gamma \vdash_{CT} \square : \tau_3 \rightsquigarrow (\square : A).m(\square : \tau_1) : \tau_2} \text{ S-App}$$

Figure 2.5: Type-guided synthesis rules (selected).

However, we believe the core of this approach—pairing effects (in our case, reads and writes) and then creating candidates using the opposing element of such a pair—can be generalized to more complex effect systems.

Synthesis Problem. We can now formally specify the synthesis problem. Given a synthesis goal $\langle \tau_1 \to \tau_2, \{\langle S_i, Q_i \rangle\} \rangle$, RbSyn searches for a program P such that, for all i, assuming that S_i calls P with an argument of type τ_1, evaluating to x_r of type τ_2, it is the case that $P \vdash S_i; Q_i \Downarrow v$. In other words, evaluating the setup followed by the postcondition yields some value rather than aborting with a failed assertion. We omit the evaluation rules as they are standard.

2.3.1 Type-Guided Synthesis

The first component of RbSyn is type-guided synthesis, which creates candidate expressions of a given type by trying to fill a hole $\square : \tau_2$ where τ_2 is the method return type. Figure 2.5 shows a subset of the type-guided synthesis rules; the full

set can be found in the companion technical report [49]. These rules have the form $\Sigma, \Gamma \vdash_{CT} e_1 \rightsquigarrow e_2 : \tau$, meaning with constants Σ, in type environment Γ, under class table CT, the holes in e_1 can be rewritten to yield e_2, which has type τ.

The rules in Figure 2.5 have two forms. The T- rules apply to expressions whose outermost form is not rewritten. Thus these rules perform standard type checking. For example, T-Var type checks a variable x by checking its type against the type environment Γ, leaving the term unchanged. T-Let typechecks and recursively rewrites (or not) the subexpressions and then rewrites those new expressions into a let-binding, ensuring the resulting term is type-correct. Finally, T-Hole applies to a typed hole that is not being rewritten, in which case it remains the same and has the given type.

The S- rules rewrite typed holes. S-Const replaces a hole by a constant of the correct type from Σ. S-Var is similar, replacing a hole by a variable from Γ. Finally, S-App replaces a hole with a call to a method with the right return type, inserting typed holes for the method receiver and argument.

Type Narrowing. Notice that in these three S-rules, the term replacing the hole may actually have a subtype of the original hole's type. Thus, type-guided synthesis could *narrow* types in a synthesized program, potentially also narrowing the search space. For example, consider an expression $(\square_1 : \texttt{Str}).\texttt{append}(\square_2 : \texttt{Str})$ that joins two strings, and assume the set of constants Σ includes `nil`. Notice that `nil` is a valid substitution for $\square_1$, which will then cause the type of the receiver to narrow to *Nil*. But then the typing derivation fails because the *Nil* type has no `append` method, stopping further exploration along this path. In contrast, if we had typed the replacement term at `Str`, then RbSyn would have fruitlessly continued the search, trying various replacements for $\square_2$ only to reject them due to a runtime failure for invoking a method on `nil`.

$$\boxed{\Sigma, \Gamma, \epsilon_r \vdash_{CT} e \twoheadrightarrow e}$$

$$\frac{\Sigma, \Gamma \vdash_{CT} e \rightsquigarrow e : \tau}{\Sigma, \Gamma, \epsilon_r \vdash_{CT} e \twoheadrightarrow \texttt{let}\ x = e\ \texttt{in}\ (\lozenge : \epsilon_r; \square : \tau)}\ \text{S-Eff}$$

$$\boxed{\Sigma, \Gamma \vdash_{CT} e \rightsquigarrow e : \tau}$$

$$\frac{}{\Sigma, \Gamma \vdash_{CT} \lozenge : \epsilon \rightsquigarrow (\lozenge : \epsilon) : \mathit{Obj}}\ \text{T-EffObj}$$

$$\frac{\epsilon_r \subseteq \epsilon'_w \qquad m : \tau_1 \xrightarrow{\langle \epsilon'_r, \epsilon'_w \rangle} \tau_2 \in CT(A)}{\Sigma, \Gamma \vdash_{CT} \lozenge : \epsilon_r \rightsquigarrow \square : \epsilon'_r; (\square : A).m(\square : \tau_1) : \tau_2}\ \text{S-EffApp}$$

$$\frac{}{\Sigma, \Gamma \vdash_{CT} \lozenge : \epsilon \rightsquigarrow \texttt{nil} : \mathit{Nil}}\ \text{S-EffNil}$$

Figure 2.6: Effect guided synthesis rule

2.3.2 Effect-Guided Synthesis

The second component of RbSyn is effect-guided synthesis, used when type-guided synthesis creates a candidate that does not satisfy the postcondition of the tests. If this happens, RbSyn computes the effect $\langle \epsilon_r, \epsilon_w \rangle$ of the failed assertion in the postcondition. (We defer the formal rules for computing this effect to the technical report [49], as they simply union the effects of method calls in the assertion.) Then, we hypothesize that the assertion may have failed because the region denoted by ϵ_r is in the wrong state.

To potentially fix the state, RbSyn applies a new rule S-Eff, shown in Figure 2.6. The hypothesis computes the type τ of e, the candidate expression that failed the postcondition. In the conclusion, e is rewritten to $\texttt{let}\ x = e\ \texttt{in}\ (\lozenge : \epsilon_r; \square : \tau)$, i.e., e is computed, bound to x, and two holes are sequenced. The first must be filled with an expression of the desired effect ϵ_r. The second must have e's type τ, to preserve type-correctness. For example, it could be filled by x, as happened in Figure 2.2 when `t0` is returned.

The rules for working with effect holes are shown in the bottom of Figure 2.6,

which extends Figure 2.5. T-EffObj gives an effect hole, that is not rewritten, type *Obj*. Since this is the top of the type hierarchy, this ensures an effect hole can safely be replaced by a term with any type. In other words, effect holes are filled for their effects, not their types. S-EffApp does the heavy lifting, filling an effect hole with a call to a method m with a write effect ϵ'_w that subsumes the desired effect ϵ_r. Of course, this call may itself read state ϵ'_r, so the rule precedes the method call with a hole with that effect, in case said state needs to change. Finally, S-EffNil replaces an effect hole with `nil`, which removes it from the program. This is used in case some extra effect holes are added that are not actually needed.

2.3.3 Merging Solutions

The last component of RbSyn combines expressions that pass individual specs into a final program that passes all specs. More specifically, given a synthesis goal $\langle \tau_1 \to \tau_2, \{s_i\} \rangle$, RbSyn first uses type- and effect-guided synthesis to create expressions e_i such that e_i is the solution for spec s_i. Then, RbSyn combines the e_i into a branching program roughly of the form `if` b_1 `then` e_1 `else if` b_2 `then` $e_2 \ldots$ for some b_i.

For each i, RbSyn uses the type-guided synthesis rules in § 2.3.1 to synthesize a b_i such that under the setup S_i of spec s_i, conditional b_i evaluates to `true`, i.e., `def` $m(x) = b_i \vdash S_i;$ `assert` $x_r \Downarrow v$. Note effect-guided synthesis is not used here as the asserted expression x_r is pure.

Notice that while each initial b_i evaluates to `true` under the precondition, there is no guarantee it is a sufficient condition for s_i to satisfy the postcondition—especially because RbSyn aims to synthesize small expressions, as discussed further in § 2.4. Moreover, there may be multiple e_i that are actually the same expression, and therefore could be combined to yield a smaller solution.

Thus, RbSyn next performs a *merging* step to create the final solution. This process operates on tuples of the form $\langle e, b, \Psi \rangle$, which is a hypothesis that the program

$$\langle e_1, b_1, \Psi_1\rangle \oplus \langle e_2, b_2, \Psi_2\rangle = \langle e_1, b_1, \Psi_1 \cup \Psi_2\rangle \quad \text{if } e_1 \equiv e_2 \text{ and } b_1 \implies b_2 \tag{2.1}$$

$$\langle e_1, b_1, \Psi_1\rangle \oplus \langle e_2, b_2, \Psi_2\rangle = \langle e_1, b_1 \vee b_2, \Psi_1 \cup \Psi_2\rangle \quad \text{if } e_1 \equiv e_2 \text{ and } b_1 \not\implies b_2 \tag{2.2}$$

$$\begin{aligned} \langle e_1, b_1, \Psi_1\rangle \oplus \langle e_2, b_2, \Psi_2\rangle &= \langle e_1, b_1^{syn}, \Psi_1\rangle \oplus \langle e_2, b_2^{syn}, \Psi_2\rangle \\ &\text{if } e_1 \not\equiv e_2 \text{ and } b_1 \implies b_2 \\ \text{where } \forall \langle S_i, Q_i\rangle \in \Psi_1.\texttt{def } m(x) = b_1^{syn} \vdash S_i; \texttt{assert } x_r \Downarrow v \\ \wedge \quad \forall \langle S_j, Q_j\rangle \in \Psi_2.\texttt{def } m(x) = b_1^{syn} \vdash S_j; \texttt{assert } !x_r \Downarrow v \\ \text{and } \forall \langle S_i, Q_i\rangle \in \Psi_1.\texttt{def } m(x) = b_2^{syn} \vdash S_i; \texttt{assert } !x_r \Downarrow v \\ \wedge \quad \forall \langle S_j, Q_j\rangle \in \Psi_2.\texttt{def } m(x) = b_2^{syn} \vdash S_j; \texttt{assert } x_r \Downarrow v \end{aligned} \tag{2.3}$$

Figure 2.7: Rewriting rules.

fragment `if` b `then` e satisfies the specs Ψ. RbSyn repeatedly merges such tuples using an operation $\langle e_1, b_1, \Psi_1\rangle \oplus \langle e_2, b_2, \Psi_2\rangle$ to represent that `if` b_1 `then` e_1 `else if` b_2 `then` e_2 satisfies the specs $\Psi_1 \cup \Psi_2$. We define $\textsc{Specs}(\langle e_1, b_1, \Psi_1\rangle \oplus ...) = \bigcup \Psi_i$, i.e., the specs from merged tuples, and $\textsc{Prog}(\langle e_1, b_1, \Psi_1\rangle \oplus ...) =$ `def` $m(x) =$ `if` b_1 `then` e_1 `else` ..., a definition with the expression represented by the merged tuples.

Figure 2.7 defines rewriting rules that are applied to create the final solution. Rule 2.1 simplifies the case where e_1 and e_2 are the same and b_1 implies b_2, yielding a single expression and branch that satisfy $\Psi_1 \cup \Psi_2$. Note we omit the symmetric case for all rules due to space limitations. Rule 2.2 applies when b_1 does not imply b_2 but e_1 and e_2 are the same. In this case, e_1 satisfies the union of the specs under the disjunction of the branch conditions. (Note this rule could also applied if $b_1 \Rightarrow b_2$, but the resulting solution would be longer than Rule 2.1 generates.) Finally, Rule 2.3 applies when e_1 and e_2 differ but b_1 implies b_2. In such a scenario, b_2 holds for both e_1 and e_2 and thus it must be that b_1 and b_2 are insufficient to branch among e_1 and e_2. Thus, RbSyn synthesizes a stronger conditional b_1^{syn} that hold for all specs in Ψ_1 and does not hold for the specs in Ψ_2, and the reverse for b_2^{syn}. For example, recall the application of this rule in the example of § 2.2, to synthesize a more precise branch

Algorithm 1 Merge programs

1: **procedure** MERGEPROGRAM(candidates $= \{\langle e_i, b_i, \Psi_i \rangle\}$)
2: merged $\leftarrow \{\bigoplus \langle e_i, b_i, \Psi_i \rangle\}$
3: final $\leftarrow \{\}$
4: **for all** $m \in$ merged **do**
5: $m \leftarrow$ apply (2.1)-(2.3) to m until no rewrites possible
6: final $\leftarrow$ final $\cup \{m\}$ if $\forall \langle S_i, Q_i \rangle \in \textsc{Specs}(m)$.
7: $\bigwedge_i \textsc{Prog}(m) \vdash S_i; Q_i \Downarrow v$
8: **end for**
9: **return** $\textsc{Prog}(m)$ s.t. $m \in$ final
10: **end procedure**

condition because the initial condition `true` was the same for both branches.

RbSyn also includes a number of other merging rules, deferred to the Appendix A.4, for further simplifying expressions. Like, `if` b_1 `then` e_1 `else if` $!b_1$ `then` e_2 `else nil` can be rewritten as `if` b_1 `then` e_1 `else` e_2, which was used to generate the solution in Figure 2.3.

Checking Implication. Checking the implications in Figure 2.7 is challenging since branch conditions may include method calls whose semantics is hard to reason about. To solve this problem, RbSyn checks implications using a heuristic approach that is effective in practice. Each unique branch condition b is mapped to a fresh boolean variable z. Similarly, $!b$ is encoded as $\neg z$, and $b_1 \vee b_2$ is encoded as $z_1 \vee z_2$. Then to check an implication $b_1 \Rightarrow b_2$, RbSyn uses a SAT solver to check the implication of the encoding. While this check could err in either direction (due to not modeling the semantics of the b_i precisely), we found it works surprisingly well in practice. In case the implication check fails due to lack of precision, we fall back on the original $\oplus$ form which represents the complete program `if` b_1 `then` e_1 `else if` ... without loss of precision. Should the implication check incorrectly succeed, it will be caught by running the merged program against the assertions.

Constructing the Final Program. Finally, notice that the merge operation $\oplus$ is not associative, and it may yield different results depending on the order in which it is applied. Thus, to get the best solution, RbSyn uses Algorithm 1. It builds the set of all possible merged fragments (line 2). Then it simplifies each candidate solution using the rewrite rules and only considers a candidate valid if it passes all tests. It returns any such program as the solution. This branch merging strategy tries all combinations, so it is less sensitive to spec order than other component based synthesis approaches [78]. In practice, we found that reordering the specs does not have much effect.

2.3.4 Discussion

Before discussing our implementation in the next section, we briefly discuss some design choices in our algorithm.

Our effect system uses pairs of read and write effects in regions. As mentioned, this core idea could be extended to any effects in a test assertion that can be paired with an effect in the synthesized method body. For example, throwing and catching exceptions, I/O to disk or network, or enabling/disabling features in a UI could all be expressed this way. We leave exploring such effect pairs to future work.

One convenient feature of our algorithm is that correctness is determined by passing specs, which are directly executed. Thus, the synthesizer can generate as many candidates as it likes—i.e., be as over approximate as it likes—as long as its set of candidates includes the solution. This feature enables RbSyn to use a fairly simple effect annotation system compared to effect analysis tools [15].

We could potentially adapt our algorithm to work in a capability-based setting, using the observation that capabilities and effects are related [27, 17, 42]. In this setting, assertion failures in tests would indicate specific capabilities needed by the synthesized code. We leave exploring this idea further to future work.

Finally, we distinguish typed holes from effect holes, rather than have a single type-and-effect hole, to control where to use type-guidance and where to use effect-guidance. When initially trying to synthesize a method body, we omit effects because it is unclear which effects are needed. For example, in Figure 2.1, the second spec has read effects on all fields of the post, and yet the target method does not write any fields, as the spec is checking the case when the post is not modified. Thus, we cannot simply compute the union of all read effects in all assertions and use those for effect guidance. Moreover, type-guided synthesis often will synthesize effectful expressions, e.g., the call to `Post.where` in Figure 2.3. Conversely, our algorithm only places effect holes in positions where the type does not matter—hence type information for such a hole would not add anything. Nonetheless, type-and-effect holes would be a simple extension of our approach, and we leave exploration of them to future work in other synthesis domains.

2.4 Implementation

RbSyn is implemented in approximately 3,600 lines of Ruby, excluding its dependencies.

Synthesis specifications, as discussed in § 2.2, are written in a custom domain-specific language. Each has the form:

```
define :name, "method-sig", [consts,...] do
  spec "spec1" do setup { ... } postcond { ... } end ...
end
```

where `:name` names the method to be synthesized; `method-sig` is its type signature; and `consts` lists constants that can be used in the synthesized method. Each `spec` is a test case the method must pass: `setup` describes the test case setup, and `postcond` makes assertions about the results.

In Ruby, `do...end` and `{...}` are equivalent syntax for creating *code blocks*, i.e., closures. Having the setup and postcondition in separate code blocks allows RbSyn to run the setup code and check the postcondition independently.

RbSyn also has optional hooks for resetting the global state before any `setup` block is run. This ensures candidate programs are tested in a clean slate without being affected by side-effects from previous runs. In our experiments, RbSyn resets the global state by clearing the database.

Program Exploration Order. While our synthesis rules are non-deterministic, our implementation is completely deterministic. This makes it sensitive to the order in which expressions are explored. RbSyn uses two metrics to prioritize search. First, programs are explored in order of their size; smaller programs are preferred over larger ones. Program size is calculated as the number of AST nodes in the program.

Second, RbSyn prefers trying effect-guided synthesis for expressions that have passed more assertions rather than fewer. (The technical report [49] formally describes counting passed assertions.) Untested candidates are assumed to have passed zero assertions. In general, expressions are explored in decreasing order of number of passed assertions, then in increasing order of program size.

These metrics combined also help when RbSyn synthesizes a candidate that does not make any progress towards a solution: after running tests and effect-guided synthesis on such candidates, their size increases, but if they do not pass more assertions, they are pushed further down the search queue. We leave experimenting with other search strategies to future work.

Effect Annotations. We extended RDL to support effect annotations along with type annotations for library methods. Programmers specify read and write effects following the grammar in § 2.3. For example a method annotated with a write effect `Post.author` writes to some region `author` in some object of class `Post`. Here `author`

is an uninterpreted string, selected by the programmer. Similarly the labels "." and "$*$" stand for pure and any region (or simply "impure"), respectively. A region `Post.*` is written as `Post` for convenience. One important extension is a `self` effect region, which indicates a read or write to the class of the receiver. This is essential for supporting ActiveRecord, whose query methods are inherited by the actual Rails model classes. For example, we use the `self` effect on the `exists?` query method of `ActiveRecord::Base`. Then at a call `Post.exists?`, where `Post` inherits from `ActiveRecord::Base`, we know the query reads the `Post` table and not any other table.

Effect annotations are similar to frame conditions [16, 70, 35] used in verification literature. More precise effect annotations help RbSyn find a solution faster because it will have fewer methods with subsumed effects than an imprecise one, shrinking the search space. But effect precision does not affect the correctness of the synthesized program, since correctness is ensured by the specs. For example, if the effect annotation for the method Post#title= shown in § 2.2.1 had just `Post` as its write annotation, synthesis would still work, but would try more candidate programs. In some cases, coarse effects are required, e.g. the Post.where method queries records from the Post table. It has the coarser Post annotation because which columns such a query will access cannot be statically specified: it depends on the arguments. We evaluate some of the tradeoffs in effect precision in § 2.5.4.

Type Level Computations. RbSyn uses RDL [36, 88] to reason about types, e.g., checking if one type is a subtype of another, and using the type environment and class table to find terms that can fill holes. RDL includes *type-level computations* [57], or *comp types*, in which certain methods' types include computations that run during type checking. For example, a comp type for the `ActiveRecord#joins` method can compute that `A.joins(B)` returns a model that includes all columns of tables `A` and `B`

combined. Using a comp type for `joins` encodes a quadratic number of type signatures, for different combinations of receivers and arguments, into a single type, and more for joins of more than two tables [57].

RbSyn uses RDL's comp types, but with new type signatures designed for synthesis. In particular, the previous version of RDL's comp types gave precise types when the receiver and arguments were known, e.g., in `A.joins(B)`, RDL knows exactly which two classes are being joined. But this may not hold during synthesis, e.g., if `B` is replaced by a hole in the example, then the exact return type of the `joins` call cannot be computed.

To address this issue, we modified RDL's existing comp type signatures for `ActiveRecord` methods like `joins` so that they compute all possible types. For example, if a hole is an argument to `joins`, then the type finds all models `B1`, `B2`, ... that could be joined (i.e., those with associations); gives the hole type $\texttt{B1} \cup \texttt{B2} \cup \ldots$; and sets the return type of joins to a table containing the columns of $\texttt{A}, \texttt{B1}, \texttt{B2}, \ldots$. This over-approximation is narrowed as the argument terms are synthesized, leading to cascading narrowing of types throughout the program as discussed in § 2.3.1.

Optimizations. Synthesis of terms that pass a spec is an expensive procedure. In practice, we found solutions to a single spec often satisfy others. Thus, when confronted with a new spec, RbSyn first tries existing solutions and conditionals to see if they hold for the spec, before falling back on synthesis from scratch if needed. This makes the bottleneck for synthesis not the number of tests, but the number of unique paths through the program. Moreover, this reduces the number of tuples for merging, as a single expression and conditional tuple can represent multiple specs Ψ.

Finally, we found that in practice, the condition in one spec often turns out to be the negation of the condition in another. Thus during synthesis of conditionals, RbSyn tries the negation of already synthesized conditionals before falling back on

synthesis from scratch.

Limitations. While RbSyn works on a wide range of programs, as we will demonstrate next, it does have several key limitations. First, RbSyn currently only synthesizes code that does not need type casts to be well-typed. This ensures programs do not have type errors at run time, but eliminates some valid programs from consideration. Second, the set of constants RbSyn can use during synthesis is fixed ahead of time. This places programs that use unlikely constants out of reach, e.g., we have encountered Rails model methods that include raw SQL query strings (instead of only using ActiveRecord). Finally, because RbSyn uses enumerative search, it can face a combinatorial explosion when searching for nested method calls, e.g., if there are n possible method calls, available, synthesizing $A.m(A.m(A.m(x)))$ may require an $O(n^3)$ search. In practice, we did not face this problem as deeply nested method calls are rarely used in Rails apps.

2.5 Evaluation

We evaluated RbSyn by using it to synthesize a range of benchmarks extracted from widely used open source applications that use a variety of libraries. We pose the following questions in our evaluation:

- How does RbSyn perform using code based on existing unit tests in widely deployed applications? (§ 2.5.2)
- How much improvement is type-and-effect guidance compared to alternatives such as only type-guidance or only effect-guidance? (§ 2.5.3)
- How does the precision of effect annotations affect synthesis performance? (§ 2.5.4)

2.5.1 Benchmarks

To answer the questions above, we collected a benchmark suite comprised of programs from the following sources:

- *Synthetic benchmarks* is a set of minimal examples that demonstrate features of RbSyn.
- *Discourse* [53] is a Rails-based discussion platform used by over 1,500 companies and online communities.
- *Gitlab* [40] is a web-based Git repository manager with wiki, issue tracking, and CI/CD tools built on Rails.
- *Diaspora* [30] is a distributed social network, with groups of independent nodes (called Pods), also built on Rails.

We selected these apps because they are popular, well-maintained, widely used, and representative of programs that are written with supporting unit tests. We selected a subset of the app's methods for synthesis, choosing ones that fall into the Ruby grammar we can synthesize: method calls, hashes, sequences of statements and branches. We currently do not synthesize blocks (lambdas), for/while loops, case statements, or meta-programming in the synthesized code. All benchmarks from apps have side effects due to either database accesses or reading and writing globals.

Table 2.1: Synthesis benchmarks and results. *# Specs* is the number of specs used to synthesize the method; *Asserts* reports the minimum and maximum number of assertions over all specs for every benchmark; *# Orig Paths* is the number of paths through the method as written in the app; *# Lib Meth* is the number of library methods used for every benchmark; *Time* shows the median and semi-interquartile range over 11 runs, followed by the median time for synthesis using only types, only effects and naive term enumeration (*Neither*). *Meth Size* is the number of AST nodes in the synthesized method; *# Syn Paths* shows the number of paths through the synthesized method.

Group	ID	Name	# Specs	Asserts		# Orig Paths	# Lib Meth	Time (sec)				Meth Size	# Syn Paths
				Min	Max			Median ± SIQR	Types	Effects	Neither		
Synthetic	S1	lvar	1	1	1	1	164	0.34 ± 0.01	1.36	11.97	-	4	1
	S2	false	1	1	1	1	164	0.35 ± 0.01	1.37	12.19	-	4	1
	S3	method chains	2	1	1	1	164	0.98 ± 0.01	9.56	-	-	10	1
	S4	user exists	2	1	1	1	164	0.98 ± 0.02	9.52	-	-	9	1
	S5	branching	3	1	1	2	165	2.49 ± 0.07	38.37	-	-	17	2
	S6	overview (ext)	3	4	4	3	164	12.78 ± 0.09	-	-	-	72	3
	S7	fold branches	3	1	1	1	164	82.44 ± 0.95	218.51	-	-	13	1
Discourse	A1	User#clear_glob…	3	2	2	3	169	2.11 ± 0.04	-	-	-	24	3
	A2	User#activate	2 (3)	1	4	2	170	8.95 ± 0.23	-	-	-	28	2
	A3	User#unstage	3 (4)	1	5	2	164	50.02 ± 0.55	-	-	-	31	2
	A4	User#check_site…	5	1	1	2	168	51.6 ± 0.23	-	-	-	28	3
Gitlab	A5	Discussion#build	1	4	4	1	167	0.24 ± 0.01	-	-	-	18	1
	A6	User#disable_two…	1	10	10	1	164	0.25 ± 0.01	-	0.44	-	22	1
	A7	Issue#close	1 (2)	3	3	1	166	0.77 ± 0.03	25.99	0.13	0.37	15	1
	A8	Issue#reopen	1 (3)	5	5	1	166	3.68 ± 0.1	-	0.55	45.66	17	1
Diaspora	A9	Pod#schedule_…	3 (4)	1	1	2	161	2.44 ± 0.04	-	-	-	19	2
	A10	User#process_inv…	1	2	2	2	165	2.64 ± 0.05	0.81	-	0.85	12	1
	A11	InvitationCode#use!	1	1	1	1	165	4.23 ± 0.06	-	-	-	12	1
	A12	User#confirm_email	7	4	4	2	166	7.28 ± 0.11	-	-	-	31	3

Table 2.1 lists the benchmarks. The first column group lists the app name (or *Synthetic* for the synthetic benchmarks); the benchmark id; the benchmark name; and the number of specs. The synthetic benchmarks exercise features of RbSyn by synthesizing pure methods, methods with side effects, methods in which multiple branches are folded into a single line program, etc. The Discourse benchmarks include a number of effectful methods in the User model, such as methods to activate an user account, unstage a placeholder account created for email integration, etc. The Gitlab benchmarks include methods that disable two factor authentication for a user, methods to close and reopen issues, etc. Finally, the Disaspora benchmarks include methods to confirm a user's email, accept a user invitation, etc.

We derived the specs for the non-synthetic benchmarks directly from the unit tests included in the app. We split each test into *setup* and *postcondition* blocks in the obvious way, and we added an appropriate type annotation to the synthesis goal. Across all benchmarks, we started with a base set of constants (Σ in § 2.3) to be `true`, `false`, 0, 1 and the empty string. Then we added `nil` and singleton classes (for calling class methods) on a per benchmark basis as needed. (As with many enumerative search based methods, we rely on the user to provide the right set of constants.)

A few apps have several different unit tests with exactly the same setup but different assertions in the postcondition. We merged any such group of tests into a single spec with that setup and the union of the assertions as the postcondition, to ensure that every spec setup can be distinguished with a unique branch condition, if necessary. We indicate this in the # *Specs* column of Table 2.1 by listing the final number of specs followed by the original number of tests in parentheses if they differ. We report the minimum and maximum number of assertions over all specs per benchmark in the *Asserts* columns and the number of paths through the method in the true canonical solution (from the app) in the # *Orig Paths* column.

Annotations for Benchmarks. Finally, the *# Lib Meth* column lists the number of library methods available during synthesis. These are methods for which we provided type-and-effect annotations. In total, 164 such methods are shared across all benchmarks, including, e.g., ActiveRecord and core Ruby libraries. Since our benchmarks are sourced from full apps, they often also depend on some other methods in the app. We wrote type-and-effect annotations for such methods and included those annotations only when synthesizing that app. Since RbSyn needs to run the synthesized code, when running specs we include the code for both general-purpose methods, such as those from ActiveRecord, and required app-specific methods. We slightly modify the set of library methods for A9, as discussed further below.

To find effect labels for app-specific methods, we found examining the method name and quickly scanning its code was typically quite helpful. Often it was clear if a method was pure or impure. For impure methods, there were a few cases. Sometimes, methods access the same object fields irrespective of how the method is called, so we give such methods the most precise labels, e.g., the effect `InvitationCode.count` was used for benchmark A10. Other times, it is apparent the method accesses different fields of a class depending on the method's arguments or the global state, so we give these class effect labels, e.g., `User` (equivalent to `User.*`). Overall, the simplicity of the effect system helped here, as we could use human-readable region identifiers even without any object references, e.g., the effect `InvitationCode.count` abstracts over all possible instances of `InvitationCode` class.

The other main category of effect labels was for Rails libraries such as ActiveRecord. We constructed these labels by following the documentation. For metaprogramming-generated column accessor methods, we extended RDL's existing type generating annotations [88] to also generate effects. For example, when RDL creates the type signature for an accessor method `Post#title` for the title column of the Post table, it now also creates a read effect annotation `Post.title` for it.

Overall, we found writing effect annotations to be easier than our previous efforts writing type annotations for Ruby [88, 57], though of course we relied on that previous experience. We leave a systematic evaluation of the effort of writing effect annotations to future work.

2.5.2 Synthesis Correctness and Performance

RbSyn successfully synthesized methods that pass the specs for every benchmark. We manually examined the output and found that the synthesized code is equivalent to the original, human-written code, modulo minor differences that do not change the code's behavior in practice. For example, one such difference occurs with original code that updates multiple database columns with a single ActiveRecord call, and then has a sequence of asserts to check that each updated column is correct. Because RbSyn considers the effects of assertions in the postcondition one by one, it instead synthesizes a sequence of database updates, one per column. Another difference occurs in Gitlab, which uses the `state_machine` gem (an external package) to maintain an issue's state (closed, reopened, etc). RbSyn synthesizes correct implementations that work without the gem.

The middle group of columns in Table 2.1 summarizes RbSyn's running time. We set a timeout of 300 seconds on all experiments. The first column reports performance numbers for the full system as the median and semi-interquartile range (SIQR) of 11 runs on a 2016 Macbook Pro with a 2.7GHz Intel Core i7 processor and 16GB RAM. The next three columns show the median performance when RbSyn uses only type-guidance, only effect-guidance, and naive enumeration, respectively. The SIQRs (omitted due to space constraints) for these runs are very small compared to the median runtime, similar to the performance numbers with all features enabled. We discuss the runs with certain guidance disabled in detail in § 2.5.3. The right-most group of columns shows the synthesized method size (in terms of number of AST

nodes) and the number of paths through the method (1 for straight-line code).

Overall, RbSyn runs quickly, with around 80% of benchmarks solving in less than 9s. Benchmarks like A3 take longer because it requires synthesis of `nil` terms—recall `nil` is the bottom element of our type lattice, causing RbSyn to synthesize `nil` at every typed hole for method arguments. Consequently, this requires testing all completed candidates—even though they eventually fail—consuming significant time.

For one benchmark, A9, we changed the set of default library methods slightly due to some pathological behavior. This benchmark includes an assertion that invokes ActiveRecord's `reload` method, which touches all fields of that record. But then when RbSyn tries to find matching write effects, it explores a combinatorial explosion of writes to different subsets of the fields. This effort is almost entirely wasted, because the remainder of the assertion looks at only one particular field—but that one read is subsumed by the effect of the `reload`, making it invisible to RbSyn's search. As a result, synthesis for A9 slows down by two orders of magnitude. We addressed this by removing four ActiveRecord methods that manipulate specific fields and adding ActiveRecord's `update!` method as the only way to write a field back to the database. An alternative approach would have been to move the `reload` call to be outside the assertion.

As this example shows, and as is common with many synthesis problems, performance is very hard to predict. Indeed, we can see from Table 2.1 that performance is generally not well correlated with either the size of the output program or with the number of branches. The number of assertions (which direct the side effect guided synthesis) does not correlate with the synthesis time. We do observe that RbSyn's branch merging strategy is effective, often producing fewer conditionals than there are specs, e.g., in A12 there are seven specs but only three conditionals. Though, sometimes the results are not always optimal if the branch merging strategy finds a program that passes all tests, but a program with fewer branches exists, e.g., for A4

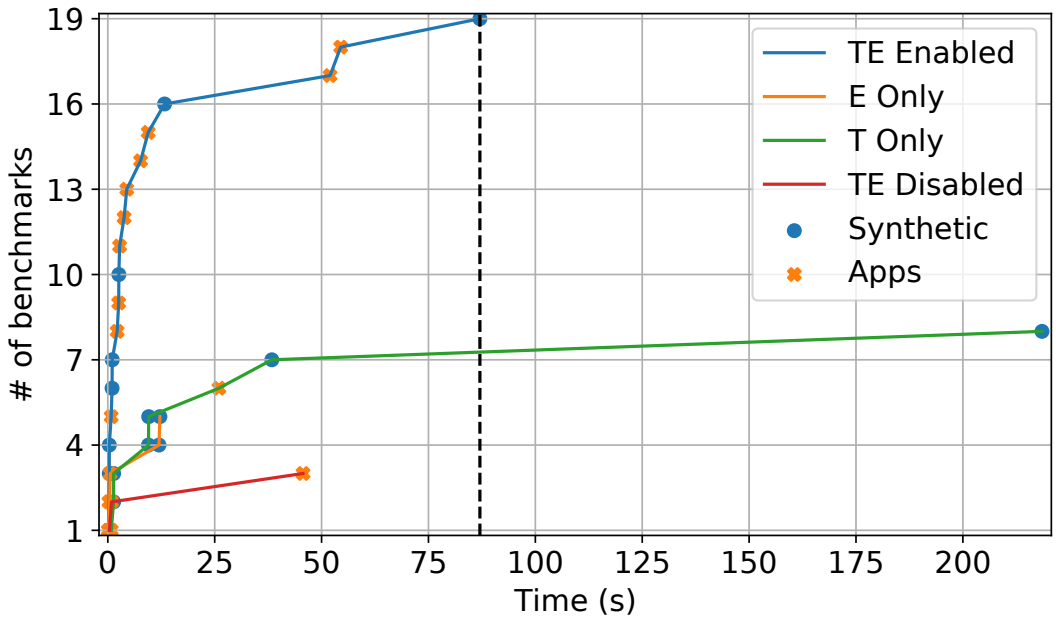


Figure 2.8: Number of benchmarks synthesized using type-and-effect (*TE Enabled*) guided synthesis relative to using only type (*T Only*) or effect (*E Only*) guidance separately and naive enumeration (*TE Disabled*). Higher is better.

and A12, RbSyn produces a program with one more branch than the hand-written.

2.5.3 Performance of Type- and Effect-Guidance

Next, we explore the performance benefits of type- and effect-guidance. Figure 2.8 plots the running times from Table 2.1 when all features of RbSyn are enabled (*TE Enabled*), with only type-guidance (*T Only*), with only effect-guidance (*E Only*) and with neither (*TE Disabled*). The plot shows the number of benchmarks that complete (y-axis) in a given amount of time (x-axis), based on the median running times. This experiment serves as a proxy to show how a synthesis procedure that uses type-guidance but not effect-guidance, such as SyPet [31] or Myth [76, 37], may have performed if adapted for Ruby.

We can clearly see that type- and effect-guided synthesis performs best, successfully synthesizing all benchmarks; the slowest takes 83s. In contrast, with both strategies disabled, all but three small benchmarks time out. Performance with only type- or only

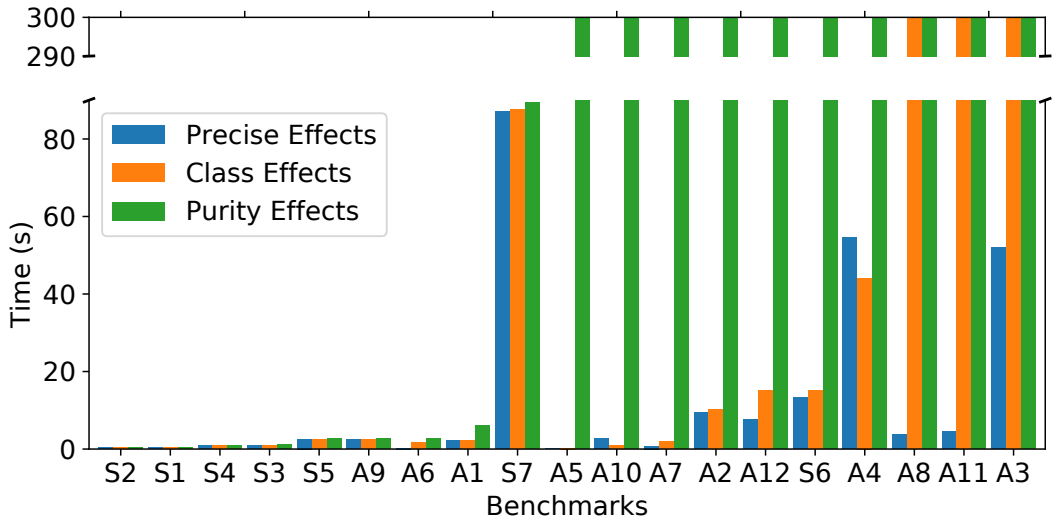


Figure 2.9: Performance of RbSyn with varying effect annotation precision: full, class effects only, and purity annotations on library methods. Lower is better. Full height indicates timeout.

effect-guidance lies in between. With only type-guidance, synthesis completes on eight benchmarks, of which the majority are pure methods from the synthetic benchmarks. From apps, it only synthesize A7 and A10. In these benchmarks, the needed effectful expressions are small and hence can be found with essentially brute-force search. With only effect-guidance, synthesis performance significantly worse, completing only five benchmarks, of which only three are from apps. These benchmarks succeeded because effect-guided synthesis quickly generates the template for the effectful method calls and then correctly fills them since they are small and can be found quickly by naive enumeration.

2.5.4 Effect Annotation Precision vs. Performance

Finally, we explore the tradeoff between effect annotation precision and synthesis performance. Recall that we found writing effect annotations easier for our benchmarks than writing type annotations. However, the effort can be further minimized by writing less precise annotations. This will not affect correctness, since RbSyn only accepts

synthesis candidates that pass all specs, but it does affect performance.

Figure 2.9 plots the median of synthesis times for benchmarks over 11 runs under three conditions: *Precise Effects*, which are the effects used above; *Class Effects*, in which annotations include only class names and eliminate region labels (e.g., `Post.title` becomes `Post`); and *Purity Effects*, in which the only effect annotations are pure or impure (the $\bullet$ and $*$ effects, respectively, in our formalism). The benchmarks (x-axis) are ordered in increasing order of time for *Purity Effects*, then *Class Effects*, and finally *Precise Effects*.

From these experiments, we see that synthesis time increases as effect annotation precision decreases, often leading to a timeout. Class labels were sufficient to synthesize 16 of 19 benchmarks. Overall, class labels take time similar to precise labels, except for the three cases (A8, A11, and A3) where side-effecting method calls require precise labels to quickly find the candidate. As all precise effects are reduced to class effects, RbSyn must try many candidates with class effect before finding the correct one, leading to timeouts.

We note that A1 and A4 are slightly faster when using class effects. The reason is an implementation detail. The effect holes in these benchmarks can only be correctly filled by methods whose regular annotations are class annotations (more precise annotations are not possible). However, when trying to fill holes, RbSyn first tries all methods with precise annotations, only afterward trying methods with class annotations. Since the precise annotations never match, this yields worse performance under the precise effect condition than under the class effect condition, when the search could by chance find the matching methods sooner.

Purity labels only enabled synthesis of 9 benchmarks, including just 3 of 12 app benchmarks. The purity annotations are slow in general and only effective in the cases where the number of impure library methods is small.

2.6 Related Work

Component-Based Synthesis. Several researchers have proposed component-based synthesis, which creates code by composing calls to existing APIs, as RbSyn does. For example, [55] propose synthesis of loop-free programs for bit-vector manipulation. Their approach uses formal specifications for synthesis, in contrast to RbSyn, which uses unit tests. Hoogle+ [54] uses Haskell tests and types to synthesize potential solutions, primarily geared towards API discovery. CodeHint [39] synthesizes Java programs, using a probabilistic model to guide the search towards expressions more often used in practice. SyPet [31] also synthesizes programs that use Java APIs, by modeling them as a petri net and using SAT-based techniques to find a solution. These approaches do not support synthesis of programs with branches, which are common in the domain of web apps. While SyPet supports synthesis with side-effecting methods and CodeHint detects undesirable side effects during the search and avoids them, RbSyn uses side effect information from test cases to guide the search.

Programming by Example. Myth [76, 37] uses bidirectional type checking to synthesize programs, using input/output examples as the specification. However, Myth expects examples to be *trace complete*, meaning the user has to provide input/output examples for any recursive calls on the function arguments. RbSyn does not synthesize recursive functions, as they are rarely needed in our target domain of Ruby web apps. Escher [1] and spreadsheet manipulation tools [46, 51, 47] all accept input/output examples as a partial specification for synthesis. These tools primarily target users who cannot program, whereas RbSyn is targeted towards programmers. In addition, RbSyn's specs are full unit tests, so they can check both return values and side effects. λ^2 [34] synthesizes data structure transformations using higher-order functions, a feature not handled by RbSyn because of our target domain of Rails web apps, which rarely use such functions. STUN [2] uses a program merging

strategy that is similar to ours, but it depends on defining domain-specific unification operators to safely combine programs under branches. In contrast, our approach may be more domain-independent, using preconditions and tests to find correct branch conditions. There have been multiple approaches to synthesizing database programs [20, 32]. Perhaps the closest in purpose to RbSyn is Scythe [108], which synthesizes SQL queries based on input/output examples. Scythe uses a two-phased synthesis process to synthesize an abstract query, after which enumeration is used to concretize the abstract query. In contrast, the use of comp types [57] allows RbSyn to quickly construct a template for a database query. With precise types for the method argument holes, this essentially builds abstract queries for free, whose holes are then filled later during synthesis.

Solver-Aided Synthesis. In solver-aided synthesis, synthesis specifications are transformed to a set of constraints for a SAT or SMT solver. Synquid [80] uses polymorphic refinement types as the specification for synthesis. Lifty [82] is a similar type system that verifies information flow control policies and synthesizes program repairs as needed to satisfy the policies. Both Synquid and Lifty synthesize conditionals using logical abduction. In contrast, RbSyn uses branch merging to synthesize conditionals, since translating Rails code and libraries into logical formulas is impractical.

Sketch [98] allows users to write partial programs, called sketches, where the omitted parts are then synthesized by the tool. Migrator [111] uses *conflict-driven learning* [33] to synthesize raw SQL queries, for use in database programs for schema refactoring. In contrast, programs synthesized by RbSyn use ActiveRecord to access the database. Rosette [102, 101] is a solver-aided language that provides access to verification and synthesis. It relies on symbolic execution, and thus requires significant modeling of external libraries for synthesizing programs that use such libraries.

EUSOLVER [3] synthesizes programs with branches, using an information-gain heuristic via decision tree learning. While, the decision tree learning procedure can produce branches in an enumerative search setting (provided the input/output example set is complete), we leave an exploration of how it compares to our rule-based merging to future work. However, EUSOLVER requires a SMT solver to produce counterexamples to build the input/output example set which has the additional cost of requiring formal specifications of library method semantics, an impractical task in the Rails setting. SuSLik [81] synthesizes heap-manipulating programs using separation logic to precisely model the the heap. RBSYN, in contrast, uses very coarse effects to track accesses that can go beyond the heap, such as database reads and writes.

2.7 Conclusion

We presented RBSYN, a system for type- and effect-guided program synthesis for Ruby. In RBSYN, the synthesis goal is described by the target method type and a series of specs comprising preconditions followed by postconditions that use assertions. The user also supplies the set of constants the synthesized method can use, and type-and-effect annotations for any library methods it can call. RBSYN then searches for a solution starting from a hole $\square : \tau$ typed with the method's return type, inserting (write) effect holes $\lozenge : \epsilon$ derived from the read effects of failing assertions. Finally, RBSYN merges together solutions for individual specs by synthesizing branch conditions to select among the different solutions as needed. We evaluated RBSYN by running it on a suite of 19 benchmarks, 12 of which are representative programs from popular open-source Ruby on Rails apps. RBSYN synthesized correct solutions to all benchmarks, completing synthesis of 15 of the 19 benchmarks in under 9s, with the slowest benchmark solving in 83s. We believe RBSYN demonstrates a promising new approach to synthesizing effectful programs.

Chapter 3

ANOSY: Approximated Knowledge Synthesis with Refinement Types for Declassification

3.1 Introduction

Information flow control (IFC) [92] systems protect the confidentiality of sensitive data during program execution. They do so by enforcing a property called *non-interference* which ensures the absence of leaks of secret information (say, a user location) through public observations (say, information being sent to the network socket).

Real-world programs, however, often need to reveal information about sensitive data. For instance, a location based web application needs to suggest restaurants or friends that are nearby the `Secret` user location. Such computations, which leak information about the `Secret` location, would be prevented by IFC systems that enforce non-interference. To support them, IFC systems provide *declassification* statements [93] that can be used to weaken non-interference by allowing the selective disclosure of some `Secret` information.

Declassification statements, however, are typically part of an application's trusted computing base and developers are responsible for properly declassifying information. In particular, mistakes in declassification statements can easily compromise a system's security because declassified information bypasses standard IFC checks. Implementing declassification statements can be difficult for developers to implement correctly. For example, [19] showed that non-Personally Identifiable Information (PII) in an advertising system could be combined to uniquely identify and target an individual. Developers may declassify seemingly non-sensitive non-PII, but accidentally leak sensitive information about a person's identity. Instead of trusting the developer to correctly declassify information, an alternative approach is to enforce *declassification policies* [21] that regulate the use of declassification statements.

In this chapter, we present ANOSY, a framework for enforcing *declassification policies* on IFC systems where policies regulate *what* information can be declassified [93] by limiting the amount of information an attacker could learn from the declassification statements. Specifically, declassification policies are expressed as constraints over *knowledge* [8], which semantically characterizes the set of secrets an attacker considers possible given the prior declassification statements. To enforce such policies, we develop (1) a novel encoding of knowledge approximations using Liquid Haskell's [104] refinement types which we use to (2) automatically synthesize correct-by-construction knowledge approximations for Haskell queries. We then (3) implement and (4) evaluate a knowledge tracking and policy enforcing declassification function that can easily extend existing IFC monadic systems. Next, we discuss these four contributions in detail.

Verified knowledge approximations We define a novel encoding for knowledge approximations over abstract domains using Liquid Haskell (§ 3.4). The novelty of our encoding is that approximation data types are indexed by two predicates that

respectively capture the properties of elements inside and outside of the domain. Using these indexes, we encode correctness of over- and under-approximations, without using quantification, permitting SMT-decidable verification. With this encoding, we implement and machine check Haskell approximations of two abstract domains: intervals over multi-dimensional spaces (where each dimension is abstracted using an interval) and powersets on these intervals, that increase the precision of our approximations. This verified knowledge encoding is general and can be used, beyond declassification, also as building block for dynamic [44, 28], probabilistic [100, 68, 43, 62], and quantitative policies [9, 61].

Synthesis of knowledge approximations We develop a novel approach for automatically synthesizing correct-by-construction posteriors given any prior knowledge and user-specified boolean query over multi-dimensional integer secret values (§ 3.5). Our approach combines type-based sketching with SMT-based synthesis and it is implemented as a Haskell compiler plugin, i.e., it operates at compile-time on Haskell programs. Given a user-defined query, ANOSY generates a synthesis template (a so-called sketch) where the values of the abstract domain elements are left as *holes* to be filled later with values, combined with the correctness specification encoded as refinement types. It then reduces the high-level correctness property into integer constraints on bounds of the abstract domain elements and uses an SMT solver to synthesize *optimal* correct-by-construction values. Replacing these values in the sketch, ANOSY synthesizes Haskell executable programs of the approximated knowledge and automatically checks their correctness with Liquid Haskell.

Enforcing declassification policies We implement a policy-based declassification function that can be used by any monadic Haskell IFC framework (§ 3.2, § 3.3). In this setting, users write declassification policies as Haskell functions that constrain the (approximated) attacker knowledge, whereas declassification queries are written

as regular Haskell functions over secret data. At compile time, ANOSY synthesizes and verifies the knowledge approximations for all declassification queries. At runtime, declassification is called in the `AnosyT` monad that tracks knowledge over multiple declassification queries and checks, using the synthesized knowledge approximations, whether performing the declassification would lead to violating the user-specified policy. Importantly, `AnosyT` is defined as a monad transformer, thus can be staged on top of existing IFC monads like LIO [99] and STORM [64].

Evaluation We evaluated precision and running time of ANOSY using two benchmarks (§ 3.6). First, we compared with `Prob`'s [68] benchmark suite to conclude that ANOSY is slower but more precise. Second, to demonstrate ANOSY enables secure declassification of sequential queries, we evaluate how many queries ANOSY allows to declassify before a policy violation. For the interval abstract domain, we found a policy violation was detected after a maximum of 7 queries and after 14 queries for the more precise powerset domain.

3.2 Overview

We start by motivating the need for declassification policies (§ 3.2.1): repeated downgrades can weaken non-interference until leaking the secret is allowed. Next, we present how the knowledge revealed by queries can be computed (§ 3.2.2). Finally (§ 3.2.3), we describe how ANOSY synthesizes correct-by-construction knowledge, by combining refinement types, SMT-based synthesis, and metaprogramming.

3.2.1 Motivation: Bounded Downgrades

Secure Monads IFC systems, *e.g.,* LIO [99] and LWeb [77], define a *secure* monad to ensure that security policies are enforced over sensitive data, like a user's physical

location. For instance here, we define the data type `UserLoc` to capture the user location as its `x` and `y` coordinates.

```
data UserLoc = UserLoc {x :: Int, y :: Int}
```

A `Secure` monad will return such a location wrapped in a protected "box" to ensure that only code with sufficient privileges can inspect it. For example, a function that gets the user's location will return a protected value:

```
getUserLoc :: User → Secure (Protected UserLoc)
```

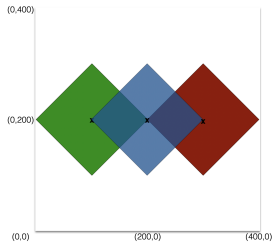
In the `LIO` monad, for example, data are protected by a security label data type and the monad ensures, based on the application, that only the intended agents can observe (or unlabel) the user's exact location.

Queries A *query* is any boolean function over *secret* values. As an example, we consider the user location to be the secret value and the `nearby` function below checks proximity to this secret value from `(x_org, y_org)`.

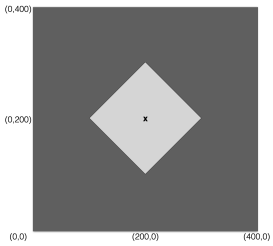
```
type 𝒮 = UserLoc


nearby :: (Int, Int) → 𝒮 → Bool
nearby (x_org, y_org) (UserLoc x y)
  = abs (x - x_org) + abs (y - y_org) ≤ 100
  where abs i = if i < 0 then -i else i
```

The `nearby` query is using Manhattan distance to check if a user is located within `100` units of the input origin location.

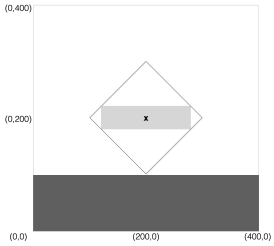
Downgrades Even though locations protected by the `Secure` monad cannot be inspected by unprivileged code, in practice many applications need to allow selective leaks of secret information to unprivileged code. For instance, many web applications need to check location of users to provide useful information, such as restaurant, friend, or dating suggestions that are physically `nearby` the user.



(a) Posteriors on x = 200, 300, 400 and y = 200.



(b) Indistinguishability Sets



(c) Under-Approximation

Figure 3.1: Posteriors, Indistinguishability Sets and their Approximations with respect to `nearby` query.

The `showAdNear` function below shows a restaurant advertisement to the user only if they are nearby. To do so, the function uses `downgrade` (from the `Secure` monad) to downgrade (to `public`) the result of the `nearby` check over the protected user location.

```
downgrade  :: Protected S → (S → Bool)
           → Secure Bool


showAd     :: User → Restaurant → Secure ()
showAdNear :: User → Restaurant → Secure ()
showAdNear user res = do
  ul     ← getUserLoc user
  isNear ← downgrade ul (nearby (res_loc res))
  if isNear then showAd user res else return ()
```

Downgrades are a common feature of real-world IFC systems. For example, in LIO downgrades happen with the `unlabelTCB` trusted codebase function, which is exposed to the application developers. At the same time, downgraded information bypasses security checks by design. In the code above, `isNear` is unprotected and can now be leaked to an attacker. Therefore, declassification statements need to be correctly placed to avoid unintended leaks of information that would bypass IFC enforcement.

Declassification knowledge To semantically characterize the information declassified by `downgrade`s, we use the notion of *attacker knowledge* [8], i.e., the set of secrets that are consistent with an attacker's observations, where attackers can observe the results of `downgrade`. That is, we consider the worst-case scenario where any declassified information is *always* leaked to an attacker. This knowledge can be refined by consecutively running downgrade queries and ultimately can reveal the exact value of the secret. In the example below, a piece of code downgrades two queries asking if the user is located nearby to both `(200,200)` and `(400,200)` to infer if the exact user location is `(300,200)`.

```
secret ← getUserLoc user
kn1    ← downgrade secret (nearby (200,200))
kn2    ← downgrade secret (nearby (400,200))
-- if kn1 ∧ kn2, then secret = (300,200)
```

The *posterior* is the knowledge obtained after executing a query. Consider again the code above. If `nearby (200,200)` is true, the knowledge after the first downgrade statement is the green region of Figure 3.1a. Using this information as *prior* knowledge for the second downgrade query, which asks `nearby (400,200)`, might result in a knowledge containing only the user location `(300,200)`, i.e., the intersection of the green and red posterior knowledge regions.

Quantitative Policies A *quantitative policy* is a predicate on knowledge which, for instance, ensures that the accumulated knowledge is not specific enough, i.e., the secret cannot be revealed. As an example, `qpolicy` below states that the knowledge should contain at least `100` values.

```
qpolicy dom = size dom > 100
```

This policy will allow declassifying `nearby (200,200)` and `nearby (300,200)`, since the intersections of the green and blue regions in Figure 3.1a contain at least 100

potential locations, but not `nearby (400,200)` since the resulting knowledge contains exactly one secret.

Bounded Downgrade We define a bounded downgrade operator that allows the computation of queries on secret data, while enforcing quantitative policies. For example, the operator tracks declassification knowledge during an execution and allows downgrading the `nearby (200,200)` and `nearby (300,200)` queries, but terminates with an error on the sequence of `nearby (200,200)` and `nearby (400,200)`.

The downgrade operation is the method of the `AnosyT` monad (§ 3.3) which is defined as a state monad transformer. As a state monad, it preserves the protected secret, the quantitative policy, and the prior declassification knowledge. To `downgrade` a new query, the monad checks if the posterior knowledge of this query satisfies the policy. If not, it terminates with a policy violation error. Otherwise, it updates the knowledge to the posterior and returns the query result. Since `AnosyT` is also a monad transformer, it can be combined with existing security monads, which provide the underlying IFC enforcement mechanism, to enrich them with extra quantitative guarantees on the inevitable downgrades.

3.2.2 Approximating knowledge from queries

Precisely computing, representing, and checking quantitative policies over a (potentially infinite) knowledge requires reasoning about all points in the input space, which is an uncomputable task in general. So, we use abstract domains (here intervals [24]) to approximate knowledge.

Indistinguishability sets The proximity query `nearby (200,200)` partitions the space of secret locations into two partitions (for the two possible responses: `True` and `False`), called *indistinguishability sets* (ind. sets), i.e., all secrets in each partition produce the same result for the query. Figure 3.1b depicts the two ind. sets for our

query. The inner diamond—depicted in light gray—is the ind. set for the result `True`, i.e., all its elements respond `True` to the query. In contrast, the outer region—depicted in dark gray—is the ind. set for `False`. Figure 3.1c depicts the under-approximated (i.e., subset) ind. sets for the query as defined by `under_indset`:

```
data AInt = AInt {lower :: Int, upper :: Int}
data A = A [AInt]

under_indset :: (A, A)
under_indset = (A [AInt 121 279, AInt 179 221],
                A [AInt   0 400, AInt   0  99])
```

The data `AInt` abstracts integers as intervals between a lower and an upper value. $\mathcal{A}$ is our abstract knowledge data type that is defined as a list of abstract integers, which can be used to abstract data with any number of integer fields. The `under_indset` is a tuple, where the first element corresponds to the `True` response and the second element to the `False` response. It says all secrets in $x \in [121, 279]$ and $y \in [179, 221]$ evaluate to `True` for the query and all secrets in $x \in [0, 400]$ and $y \in [0, 99]$ evaluate to `False`.

Knowledge under-approximation. We use ind. sets to compute the *posterior* knowledge after the query, i.e., the set of secrets considered possible after observing the query result. To do so, we simply take the intersection $\cap$ of the prior knowledge with the ind. sets associated with the query [8, 9]. If the intersection happens with the exact ind. sets, then we derive the exact posterior. For our example, we intersect with the under-approximate ind. set to produce an under-approximation of the posterior knowledge i.e., an under-approximation of the information learned when observing the query result.

```
underapprox :: A →  (A, A)
underapprox p = (p ∩ trueInd, p ∩ falseInd)
```

```
where (trueInd, falseInd) = under_indset
```

The intersection ∩ refers to the set-theoretic intersection of two domains. We formally define these operations in § 3.4.

3.2.3 Verification and Correct-by-Construction Synthesis of Knowledge

Our goal is to generate a knowledge approximation for each downgrade query, which as shown by our `nearby` example is a strenuous and error prone process. To automate this process ANOSY uses refinement types, metaprogramming, and SMT-based synthesis to automatically generate correct-by-construction knowledge approximations of queries in four steps. First, for each query ANOSY generates a refinement type specification that denotes knowledge approximation. Next, it uses metaprogramming to generate a partial program, called a sketch, i.e., a function definition with holes (to be eventually substituted with terms) that computes the knowledge. Then, it uses an SMT solver to fill in the integer value holes in the sketch. Finally, it delegates to Liquid Haskell's refinement type checker to verify that our synthesized knowledge indeed satisfies its specification.

Here, we explain a simplified version of these steps for our `nearby (200,200)` example query.

Step I: Refinement Type Specifications ANOSY uses abstract refinement types to index abstract domains with a predicate that all its elements should satisfy (§ 3.4). For example, $\mathcal{A}$ `<{\l → 0 < l}>` denotes the abstract domain whose elements are positive values. Using this abstraction, ANOSY specifies the ind. set and knowledge approximations:

```
under_indset :: (A <{\l →   nearby l}>,
                 A <{\l → ¬ nearby l}>)
```

```
underapprox :: p: 𝒜
            → (𝒜 <{\x →   nearby x ∧ (x ∈ p)}>,
               𝒜 <{\x → ¬ nearby x ∧ (x ∈ p)}>)
```

`under_indset` returns a tuple of abstract domains. The first abstract domain can only contain elements that satisfy the query and the second that falsify it. The function `underapprox` computes the posterior given some prior knowledge `p`. The posterior is further refined to contain only elements that originally existed in the prior knowledge.

Step II: Sketch Generation Using syntax directed meta-programming ANOSY defines `underapprox` as in § 3.2.2 to be the intersection of the ind. set and the prior knowledge. For the definition of the ind. set it relies on the secret type to be translated to generate a sketch with integer value holes. Since `UserLoc` contains two integer fields, the sketch [98] for `under_indset` is the following, where all `l` and `u` are holes:

```
under_indset = (𝒜 [𝒜Int l_t1 u_t1, 𝒜Int l_t2 u_t2],
                𝒜 [𝒜Int l_f1 u_f1, 𝒜Int l_f2 u_f2])
```

Step III: SMT-Based Synthesis Finally, it combines the refinement type with the program sketch to generate, using an SMT solver, solutions for the integer holes (§ 3.5). By combining values from the above sketch for `under_indset` with its refinement type, the below constraints are generated:

$$\forall x, y.\ l_{t1} \leq x \leq u_{t1} \ \wedge\ l_{t2} \leq y \leq u_{t2} \implies nearby(x, y) \qquad \text{(Under-approx, True)}$$

$$\forall x, y.\ l_{f1} \leq x \leq u_{f1} \ \wedge\ l_{f2} \leq y \leq u_{f2} \implies \neg nearby(x, y) \qquad \text{(Under-approx, False)}$$

The first constraint indicates all points in the domain should satisfy the `nearby` function, whereas the second constraint means the all points inside the domain should not satisfy the nearby function. The definition of `nearby (200,200)` and the `abs`

function is mechanically translated to logic as follows:

$$query(x, y) = abs(x - 200) + abs(y - 200) \leq 100$$

$$abs(i) = \texttt{if } i < 0 \texttt{ then } -i \texttt{ else } i$$

These constraints have multiple correct solutions, but, for precision, ANOSY prefers the tightest bounds. Specifically, when under-approximating, it aims for the maximal domain that satisfies the above two constraints. ANOSY uses Z3 [14] as the SMT solver of choice because it supports optimization directives for maximizing $u_1 - l_1$ and $u_2 - l_2$ together, for both the true and false cases. Finally, it uses the SMT synthesized solutions to fill in the holes and derive complete programs, like the `under_indset` of § 3.2.2.

Step IV: Knowledge Verification ANOSY uses LiquidHaskell to verify the synthesized result. To achieve this step, we implemented (§ 3.4) verified abstract domains for intervals and their powersets that, as shown in our evaluation § 3.6, greatly increase the precision of the abstractions. These implementations are independent of the synthesis step and can be used to verify manually user-written, knowledge approximations as well.

3.3 Bounded Downgrade

Here we present the bounded downgrade operation, first by an example that showcases how downgrades that violate the quantitative declassification policy are rejected, next by providing its exact implementation, and finally by showing correctness of policy enforcement.

Bounded Downgrade by Example The bounded downgrade function checks, before running a downgrade query using the underlying `Secure` monad, that the approximation of the revealed knowledge satisfies the quantitative policy. To do so, it preserves a state that maps each secret that has been involved in downgrading operations to its current knowledge. As an example, below we present how the knowledge is updated to prevent the example from § 3.2.1.

```
secret ← lift (getUserLoc user)
-- secret  = Protected (UserLoc 300 200)
-- secrets = []
r1 ← downgrade secret "nearby (200,200)"
-- secrets = [(secret, post1 = {121...279,179...221})], |post1| = 6837
r2 ← downgrade secret "nearby (300,200)"
-- secrets = [(secret, post2 = {221...279,179...221})], |post2| = 2537
r3 ← downgrade secret "nearby (400,200)"
-- secrets = [(secret, post3 = {∅        , 179 ... 221})], |post3| = 0
-- Policy Violation Error
```

The user location is taken by lifting the `getUserLoc` function of the underlying monad (any computation of the underlying monad can be lifted). Assume that the user is located at `(300,200)`. Originally, there is no prior knowledge for this secret (and protected) location, i.e., the `secrets` map associating secrets to knowledge approximations is empty. After downgrading the `nearby (200,200)` query (which as we will explain next, is passed to `downgrade` as a string) we get the posterior `post1` with size `6837`. Since this size is greater than `100`, the `qpolicy` (defined in § 3.2.1) is satisfied and the result of the query (here true) is returned by the bounded downgrade. Similarly, downgrade of the `nearby (300,200)` query refines the posterior to size `2537`. But, when downgrading the `nearby (400,200)` query the posterior size becomes zero, thus our system will refuse to perform the query (and downgrading its result) and return a policy violation error, instead of risking the leak of the secret.

Definition of Bounded Downgrade Figure 3.2 presents the definition of the bounded `downgrade` function. It takes as input a protected secret, which should be able to get `unprotected` by an instance of the `Unprotectable` class, a string that uniquely determines the query to be executed, and returns a boolean value in the `AnosyT` state monad transformer [66]. As discussed in § 3.2.1, we used a transformer to stage our downgrade on top of an existing secure monad.

The state of Anosy `AState` contains the quantitative policy, the map `secrets` of secret values to their current knowledge, and the map `queries` that maps strings that represent queries to query information `QInfo` that, in turn, contain both the query itself and an under-approximation function (like the synthesized `underapprox`) that given the prior knowledge approximates the posterior, after the query is executed. Even though tracking of multiple secrets is permitted, we require all the secrets and abstractions to have the same type; this limitation can be lifted using heterogeneous collections [59].

Having access to this state, `downgrade` will throw an error if it cannot find the query information of the string input, since it has no way to generate the posterior knowledge[1]. Then, it will compute the posterior and throw an error if it violates the quantitative policy. Otherwise, it will update the posterior of the secret and return the result of the query. Note that detection of violations of the quantitative policy is independent of the actual secret value.

Correctness: Policy Enforcement Suppose a secret `s` that has been downgraded n times by the queries $\texttt{query}_1, \ldots, \texttt{query}_n$. After each downgrade, the knowledge is refined. So, starting from the top knowledge ($\mathcal{K}_0 \doteq \top$), after n queries, the knowledge evolves as follows: $\mathcal{K}_0 \subseteq \mathcal{K}_1 \subseteq \cdots \subseteq \mathcal{K}_i \subseteq \cdots \subseteq \mathcal{K}_n$, where $\mathcal{K}_i = \mathcal{K}_{i-1} \cap \{x \mid \text{query}_i\ x = \text{query}_i\ \texttt{s}\}$.

We can show that for each i-th downgrade of the secret `s`, there exists a posterior

[1] On-the-fly synthesis albeit possible would be very expensive.

```
type AnosyT a s m = StateT (AState a s) m

data AState a s = AState {
  policy   :: a → Bool,
  secrets  :: Map s a,
  queries  :: Map String (QInfo a s)}

data QInfo a s = QInfo {
  query  :: s → Bool,
  approx :: p: a
         → (a <{\x →   query x ∧ (x ∈ p)}>,
             a <{\x → ¬ query x ∧ (x ∈ p)}>)}

class Unprotectable p where
  unprotect :: p t → t

downgrade :: ( Monad m, Unprotectable protected
             , AbstractDomain a s) -- Defined in § 4.1
          ⇒ protected s
          → String -- (s → Bool)
          → AnosyT a s m Bool
downgrade secret' qName = do
  st ← get
  let qinfo  = lookup qName (queries st)
  if isJust qinfo then do
    let secret = unprotect secret'
    let prior  = fromMaybe ⊤
               $ lookup secret (secrets st)
    let (QInfo query approx) = fromJust qinfo
    let (postT, postF) = approx prior
    if policy st postT ∧ policy st postF then do
      let response = query secret
      let posterior = if response then postT
                      else postF
      modify $ \st → st {secrets =
        insert secret posterior (secrets st)}
      return $ response
    else throwError "Policy Violation"
  else throwError ("Can't downgrade " ++ qName)
```

Figure 3.2: Implementation of bounded `downgrade`.

$\mathcal{P}_i$ so that `(s,`$\mathcal{P}_i$`)` is in the `secrets` map and also $\mathcal{P}_i$ is an under-approximation of the knowledge $\mathcal{K}_i$, that is $\mathcal{P}_i \subseteq \mathcal{K}_i$. The proof goes by induction on i, assuming that the attacker and the `downgrade` implementation start from the same $\top$ knowledge, and the inductive step relies on the specification of the `approx` function and the way `downgrades` modifies `secrets`, i.e., using `postT` or `postF` depending on the response of the query.

Thus if our quantitative policy enforces a lower bound on the size of the leaked knowledge, (*e.g.,* `qpolicy dom = size dom > k`) it is correctly enforced by `downgrade`: since $\mathcal{P}_i \subseteq \mathcal{K}_i$, then `qpolicy` $\mathcal{P}_i$ implies `qpolicy` $\mathcal{K}_i$ at each stage of the execution. Note that for correctness of policy enforcement, the policy should be an increasing function in the size of the input for underapproximations. The exact definition of such a policy domain specific language is left as a future work. Further, even though our implementation can trace knowledge overapproximations, we have not yet studied applications or policy enforcement for this case. Last but not least, it is important that the policy is checked irrespective of the query result, i.e., on both `postT` and `postF`, to prevent potential leaks due to the security decision.

Security Guarantees ANOSY enforces declassification policies that limit the amount of information an attacker can learn from declassification statements. For this, ANOSY directly checks that downgrades are bounded (§3.3) and it relies on the underlying security monad to ensure that the adversary's knowledge remains constant, i.e., there are no leaks, between two `downgrade`s. As a result, the underlying security monad needs to enforce termination-sensitive non-interference. Alternatively, one can use a monad enforcing termination-insensitive non-interference, such as LIO [99], and additionally prove termination, *e.g.,* using Liquid Haskell's termination checker.

```
class AbstractDomain a s where
  ⊤    :: a <{\_ → True , \_ → False}>
  ⊥    :: a <{\_ → False, \_ → True }>
  ∈    :: s → a → Bool
  ⊆    :: a → a → Bool
  ∩    :: d1:a <p1, n1> → d2:a <p2, n2>
       → {d3:a <p1∧p2, n1∨n2> | d1 ⊆ d3 ∧ d2 ⊆ d3}
  size :: a → {i:Int | 0 ≤ i}
  -- class laws
  sizeLaw    :: d1:a → {d2:a | d1 ⊆ d2}
             → {size d1 ≤ size d2}
  subsetLaw :: c:s → d1:a → {d2:a | d1 ⊆ d2}
             → {c ∈ d1 ⇒ c ∈ d2}
```

Figure 3.3: Abstract Domain Type Class

3.4 Refinement Types Encoding

We saw that our bounded downgrade function is correct, if each query is coupled with a function `approx` that correctly computes the underapproximation of posterior knowledge. Here, we show how refinement types can specify correctness of `approx`, in a way that permits decidable refinement type checking. First (§ 3.4.1), we define the interface of abstract domains as a refined type class that in § 3.4.2 we use to specify the abstractions of ind. sets and knowledge. Next, we present two concrete instances of our abstract domains: intervals (§ 3.4.3) and powersets of intervals (§ 3.4.4).

3.4.1 Abstract Domains

Figure 3.3 shows the `AbstractDomain` a s refined type class interface stating that a can abstract, i.e., represent a set of values of, s. For example, an instance `instance AbstractDomain` $\mathcal{A}_I$ `UserLoc` states that the data type $\mathcal{A}_I$ (that we will define in § 3.4.3) abstracts `UserLoc` (of § 3.2.1). The interface contains method definitions and class laws, and when required the abstract domain is indexed by abstract refinements.

Class Methods The class contains six, standard, set—theoretic methods. Top ($\top$) and bottom ($\bot$), respectively represent the full and empty domains. Member $c \in d$ tests if the concrete value c is included in the abstract domain d. Subset $d_1 \subseteq d_2$ tests if the abstract domain d_1 is fully included in the abstract domain d_2. Intersect $d_1 \cap d_2$ computes an abstract domain that includes all the concrete values that are included in both its input domains. Finally, `size d` computes the number of concrete values represented by an abstract domain.

Class Laws We use refinement types to specify two class laws that should be satisfied by the $\subseteq$ and `size` methods. `sizeLaw` states that if $\texttt{d}_1$ is a subset of $\texttt{d}_2$, then the size of $\texttt{d}_1$ should be less than or equal to the size of $\texttt{d}_2$. `subsetLaw` states that if $\texttt{d}_1$ is a subset of $\texttt{d}_2$, then any concrete value in $\texttt{d}_1$ is also in $\texttt{d}_2$. These methods have no computational meaning (i.e., they return unit) but should be instantiated by proof terms that satisfy the denoted laws. Even though we could have expressed more set-theoretic properties as laws, these two were the ones required to verify our applications.

Abstract Indexes In the types of top, bottom, and intersection, the type a is indexed by two predicates `p` and `n` (both of type $s \rightarrow$ `Bool`). The positive predicate `p` describes properties of concrete values that are members of the abstract domain. Dually, the negative predicate `n` describes properties of the values that do not belong to the abstract domain. Intuitively, the meaning of these predicates is the following:

a `<p,n> ~ {d:`a `| ∀x. x∈d ⇒ p x ∧ ∀x. x∉d ⇒ n x}`

Yet, the right-hand side definition is using quantifiers which lead to undecidable verification. Instead, we used abstract refinements [103] and the left-hand side encoding, to ensure decidable verification.

The specification of the full domain $\top$ states that the positive predicate is `True`, i.e., all elements belong to the domain, and the negative `False`, i.e., no elements are

outside of the domain. Similarly, the empty domain $\bot$ has a `False` positive predicate, i.e., no elements are in the domain, and `True` negative predicate, i.e., all elements can be outside the domain. Finally, the type signature for intersect $\texttt{d}_1 \cap \texttt{d}_2$ returns a domain $\texttt{d}_3$ whose positive predicate indicates it includes elements included in $\texttt{d}_1$ and $\texttt{d}_2$ i.e., $\texttt{p}_1 \wedge \texttt{p}_2$. The negative predicate indicates points excluded from $\texttt{d}_3$ are points excluded from either $\texttt{d}_1$ or $\texttt{d}_2$, i.e., $\texttt{n}_1 \vee \texttt{n}_2$. The refinement on $\texttt{d}_3$ ensure that $\texttt{d}_3$ is a subset $\subseteq$ of both $\texttt{d}_1$ and $\texttt{d}_2$. For abstract types in which these two predicates are omitted, the `\_ → True` predicate is assumed, which we will from now on abbreviate as `true` and imposes no verification constraints.

3.4.2 Approximations of ind. sets and knowledge

In Figure 3.4, we use the positive and negative abstract indexes to encode the specifications of over- and under-approximations for ind. sets and knowledge. We assume concrete types for a and s with an `instance AbstractDomain` a s and a query on the secret. (In the previous sections for simplicity, we omitted the negative predicates and overapproximations.)

Approximations of ind. sets A query's ind. sets is a tuple whose first element is an abstract domain that represents secrets that satisfying the query and the second element is an abstract domain that represents secrets that falsify the query.

The specification of the ind. sets `under_indset` says the first domain only includes secrets for which the `query` is `True` and the second domain only includes secrets for which the `query` is `False` (the positive predicates). The negative predicates do not impose any constraints on the elements that do not belong to the domain. This means the domains can exclude any number of secrets, as long as the secrets that are included are correct, i.e., it is an under-approximation.

Dually, the over-approximation `over_indset` sets the negative predicate to exclude

```
query :: s → Bool

under_indset :: (a <{\x →   query x, true}>,
                 a <{\x → ¬ query x, true}>)
over_indset  :: (a <{true, \x → ¬ query x}>,
                 a <{true, \x →   query x}>)

underapprox :: p:a →
  (a <{\x →   query x ∧ (x ∈ p), true}>,
   a <{\x → ¬ query x ∧ (x ∈ p), true}>)
underapprox p = (dT ∩ p,dF ∩ p)
  where (dT,dF) = over_indset
overapprox  :: p:a →
  (a <{true, \x → ¬ query x ∨ (x ∉ p)}>,
   a <{true, \x →   query x ∨ (x ∉ p)}>)
overapprox p = (dT ∩ p,dF ∩ p)
  where (dT,dF) = over_indset
```

Figure 3.4: Specifications of Approximations for concrete a and s that instantiate `AbstractDomain`.

all points for which the `query` evaluates to `False` for the domain corresponding to the `True` response and the second domain (corresponding to the `False` response) excludes all points for which the `query` evaluates to `False`. The positive predicates are just true. The domains can include any number of secrets as long as they are not leaving out any secrets that are correct, i.e., it is an over-approximation.

Approximations of knowledge By combining the prior knowledge of the attacker with the ind. set for the query, we derive an approximation of the attacker's knowledge after they observe the query. Figure 3.4 shows the specifications for the knowledge under-approximation `underapprox` and the over-approximation `overapprox`. `underapprox` is similar to the type of `under_indset`, except the positive predicate is strengthened to express that all the elements of the domain should also belong to the prior knowledge `p`. Similarly, `overapprox` specifies that the elements that do not belong in the posterior knowledge, should neither be in the prior nor the ind. set. Each approximation is implemented by a pair-wise intersection with the respective

ind. sets and can be verified thanks to the precise type we gave to intersection.

Precision The refinement types ensure our definitions are correct, but they do not reason about the precision of the abstract domains. For example, the bottom and top domains are vacuously correct solutions for under- and over-approximations, respectively. But, these domains are of little use as ind. sets, since they ignore all the query information. It is unclear if precision can be encoded using refinement types. Instead, we empirically evaluate precision in § 3.6.

3.4.3 The Interval Abstract Domain

Next we define $\mathcal{A}_I$, the interval abstract domain that can abstract any secret type $\mathcal{S}$, constructed as a product of integers (like the `UserLoc` of § 3.2) or types that can be encoded to integers (*e.g.,* booleans or enums). $\mathcal{A}_I$ is defined as follows:

```
-- S = Int × Int × ...
data AInt = AInt {lower :: Int, upper :: Int}
type Proof p x = {v:S<p> | v = x }

data AI <p::S → Bool, n::S → Bool>
  = AI { dom :: [AInt]
       , pos :: x:{S| x ∈ dom } → Proof p x
       , neg :: x:{S| x ∉ dom } → Proof n x }
  | TI { pos :: x:S → Proof p x }
  | BI { neg :: x:S → Proof n x }
```

$\mathcal{A}_I$ has three constructors. $\top_I$ and $\bot_I$ respectively denote the complete and empty domains. $\mathcal{A}_I$ represents the domain of any n-dimensional intervals, where `n` is the length of `dom`. An interval `AInt` represents integers between `lower` and `upper`. For a secret $s = s_1 \times s_2 \times \ldots s_n$, an $\mathcal{A}_I$ represents each s_i by the `i`th element of its `dom` ($s_i \in$ `(dom!i)`) in the `n` dimensional space. For example, `domEx = [(AInt 188`

`212), (AInt 112 288)]` is the rectangle of $x \in [188, 212]$ and $y \in [112, 288]$ in the two dimensional space of `UserLoc`.

Proof Terms The `pos` and `neg` components in the $\mathcal{A}_I$ definition are proof terms that give meaning to the positive `p` and negative `n` abstract refinements. The complete domain $\top_I$ contains the proof field `pos` that states that every secret s should satisfy the positive predicate `p` (i.e., `x: S → Proof p x`) and the empty domain contains only the proof `neg` for the negative predicate `n`. Due to syntactic restrictions that abstract refinements can only be attached to a type for SMT-decidable verification [103], the proof terms are encoded as functions that return the secret, while providing evidence that the respective predicates are inhabited by possible secrets. In $\mathcal{A}_I$ this is encoded by setting preconditions to the proof terms: the type of the `pos` field states that each `s` that belongs to `dom` should satisfy `p`, while the `neg` field states that each `x` that does not belong to `dom` should satisfy `n`.

When an $\mathcal{A}_I$ is constructed via its data constructors, the proof terms should be instantiated by explicit proof functions. For example, below we show that the `domEx` (described above) only represents elements that are `nearby (200,200)`.

```
example :: AI <{\s → nearby (200,200) s, true}>
example = AI domEx exPos (\x → x)

exPos :: s:{UserLoc | s ∈ domEx }
      → {o:UserLoc | nearby (200,200) s ∧ o = s}
exPos (UserLoc x y) = UserLoc x y
```

The proof term `exPos` is an identity function refined to satisfy the `pos` specification. Once the type signature of `exPos` is explicitly written, Liquid Haskell is able to automatically verify it. Automatic verification worked for all non-recursive queries, but for more sophisticated properties (*e.g.,* in the definition of the intersection function)

we used Liquid Haskell's theorem proving facilities [106] to establish the proof terms. Importantly, when $\mathcal{A}_I$ is used opaquely (like in `approx` in Figure 3.4), the proof terms are automatically verified.

AbstractDomain Instance We implemented the methods of the `AbstractDomain` class for the $\mathcal{A}_I$ data type as interval arithmetic functions lifted to n-dimensions. $\in$ checks if any secret is between `lower` and `upper` for every dimension. $\subseteq$ checks if the intervals representing the first argument is included in the intervals representing the second argument. $\cap$ computes a new list of intervals to represent the abstract domain, that includes only the common concrete values of the arguments. Size just computes the number of secrets in the domain, which can be interpreted as the domain's volume. Our implementation consists of 360 lines of (Liquid) Haskell code, the vast majority of which constitutes explicit proof terms for `pos` and `neg` fields and the class law methods. By design, $\mathcal{A}_I$ uses a list to abstract secrets that are sums of any number of elements, thus this class instance can be reused by an ANOSY user to abstract various secret types.

3.4.4 The Powersets of Intervals Abstract Domain

To address the internal imprecision of the interval abstract domains, we follow the technique of [84, 10] and define the powerset abstract domain $\mathcal{A}_{\mathbb{P}}$, i.e., a set of interval domains. Similar to intervals, the powerset $\mathcal{A}_{\mathbb{P}}$ is also parameterized with the positive and negative predicates:

```
data A_P <p::S → Bool, n::S → Bool> = A_P {
    dom_i :: [A_I] , dom_o :: [A_I]
  , pos :: x:{S| x ∈ dom_i ∧ x ∉ dom_o} → Proof p x
  , neg :: x:{S| x ∉ dom_i ∨ x ∈ dom_o} → Proof n x }
```

$\mathcal{A}_{\mathbb{P}}$ contains four fields. $\texttt{dom}_i$ is the set (represented as a list) of intervals that are

contained *in* the powerset. $\texttt{dom}_o$ is the set of intervals that are *excluded* from the powerset. This representation backed by two lists gives flexibility to define powersets by writing regions that should be included and excluded, without sacrificing generality or correctness (as guaranteed by our proofs). Moreover, this encoding of the powerset makes our synthesis algorithm simpler (§ 3.5). The proof terms provide the boolean predicates that give semantics to the secrets contained in the powerset, similar to the interval abstract domain (§ 3.4.3). We do not need a separate top $\top$ and bottom $\bot$ for $\mathcal{A}_{\mathbb{P}}$ as they can be represented using $\top_I$ or $\bot_I$ in the `pos` list.

AbstractDomain Instance We implemented the methods of the `AbstractDomain` class for the powerset abstraction in 171 lines of code. A concrete value belongs to ($\in$) the powerset $\mathcal{A}_{\mathbb{P}}$ if it belongs to any individual interval of the $\texttt{dom}_i$ list but not to any individual interval of the $\texttt{dom}_o$ list. The subset $\texttt{d}_1 \subseteq \texttt{d}_2$ operation checks if each individual interval in the inclusion list $\texttt{dom}_i$ of $\texttt{d}_1$ is a subset of at least one interval in the inclusion list $\texttt{dom}_i$ of $\texttt{d}_2$ and also that none of the individual intervals in the exclusion list $\texttt{dom}_o$ of $\texttt{d}_1$ is a subset of any interval in $\texttt{dom}_o$ of $\texttt{d}_2$. This operation returns `True` if the first powerset is a subset of the second, but if it returns `False` it may or may not be powerset. We have not found this to be limiting in practice, as this criteria is sufficient for verification. We plan to improve the accuracy via better algorithms in future work. Intersection $\texttt{d}_1 \cap \texttt{d}_2$ produces a new powerset, whose inclusion list is made of pairwise intersecting intervals from $\texttt{dom}_i$ of $\texttt{d}_1$ and $\texttt{dom}_i$ of $\texttt{d}_2$ and the exclusion interval list is simply the union of all intervals in the individual exclusion lists $\texttt{dom}_o$ of $\texttt{d}_1$ and $\texttt{dom}_o$ of $\texttt{d}_2$. Size is the sum of the size of all intervals in the inclusion list minus the size of all intervals in the exclusion list.

3.5 Synthesis of Optimal Domains

We use synthesis in ANOSY to automatically generate ind. sets that satisfy the correctness types of Figure 3.4 for each downgrade query. Our synthesis technique proceeds in three steps: first, ANOSY extracts the sketch of the posterior computation (§ 3.5.2). Second, it translates this to SMT constraints with relevant optimization directives to synthesize the abstract domains (§ 3.5.3). Finally, the SMT synthesis is iterated to allow synthesis of powersets of any size (§ 3.5.4). To efficiently perform these synthesis steps using SMT, we used a very restrictive form of the query language (§ 3.5.1).

3.5.1 The query language

The queries analyzed by ANOSY are Haskell functions that take one input, of the secret type, and return a boolean: `query` :: $s \rightarrow$ `Bool` (as per Figure 3.4).

For algorithm and efficient synthesis and verification, all the queries we tried are restricted to linear arithmetic, bool-eans, and data types that have a direct, syntactic translation to SMT functions restricted to decidable logic fragments. Concretely, the queries can call other functions that belong to the same fragment, but recursive definitions of queries are rejected by ANOSY.

Supporting other query classes The query language can be easily extended to support non-boolean queries with finitely many outputs. This can be done by computing one ind. set per possible output. Further, our secrets currently and for simplicity are restricted to integer products, but they can be easily extended to other domains with decidable decision procedures (*e.g.,* datatypes). Extensions to undecidable secret types (*e.g.,* floating points, strings) has unclear implications and is deferred to future work.

3.5.2 Synthesis Sketch

We use syntax-directed synthesis by starting with a partial program i.e., sketch [98], for the ind. sets based on their type specifications in Figure 3.4. For example, the sketch for the under-approximate ind. sets would be:

```
under_indset = (□::𝒜 <{\x →   query x, true}>,
                □::𝒜 <{\x → ¬ query x, true}>)
```

Following the structure of the type we simply introduce *typed* holes of the form $\square :: \tau$ for each abstract domain, which for this case is (refined) $\mathcal{A}$.

3.5.3 SYNTH: SMT-based Synthesis of Intervals

We define the procedure SYNTH that given a typed hole of an abstract domain, the number of fields in the secret n, and the kind of approximation (over or under), it returns a solution, i.e., an abstract domain that satisfies the hole's type. As an example, consider the below solution to first typed hole of `under_indset`. All `l` and `u` are symbolic integers.

```
□  :: 𝒜_I <{\x → query x, \_ → True}>
□   = 𝒜_I dom pos neg
dom = [𝒜Int l_1 u_1, ..., 𝒜Int l_n u_n]
```

The above solution is using the $\mathcal{A}_I$ applied to the domain list `dom` and the `pos` and `neg` proof terms. The proof terms for our (non recursive) queries follow concrete patterns (as the example of § 3.4.3) and are generated from syntactic templates. The `dom` is a list of ranges $\mathcal{A}$`Int` that contains symbolic integers as lower ($\texttt{l}_i$) and upper ($\texttt{u}_i$) bounds, while the length n of the list is the number of fields of the secret data type.

To find concrete values for the symbolic integers $\texttt{l}_i$ and $\texttt{u}_i$, SYNTH mechanically generates SMT implications based on the type indexes. Since the positive index states that all elements `x` on the domain should satisfy `query x` and the negative index

states that all elements outside of the domain should satisfy `True`, the following SMT constraint is mechanically generated:

$\forall$ `x. (x`$\in$ `dom` $\Rightarrow$ `query x)` $\wedge$ `(x`$\notin$ `dom` $\Rightarrow$ `True)`

Such constraints (see § 3.2.3 for a concrete example) are sent to the SMT, by a direct, syntactic translation of the Haskell instance method $\in$ and the `query` definitions into Z3 functions (§ 3.5.1).

Solving such constraints gives us a value for `dom` if a solution exists. In practice, however, such solutions are often just a point, i.e., the abstract domain contains only one secret. Although this is a correct solution, it is not precise. To increase precision we add optimization directives to constraints depending on the type of our approximation. That is, for $i \in \{1 \dots n\}$ we add `maximize` $\mathtt{u}_i$ `-` $\mathtt{l}_i$ or `minimize` $\mathtt{u}_i$ `-` $\mathtt{l}_i$ for under-approximations and over-approximations respectively. These optimization constraints are handed to an SMT solver that supports optimization directives [14] and the produced model is an intended solution for `dom`. We used the Pareto optimizer of Z3 [14], such that no single optimization objective dominates the solution. For example, if two domains of sizes 400×1 and 20×20 are valid solutions, ANOSY will prefer the latter.

3.5.4 ITERSYNTH: Iterative Synthesis of PowerSets

Powerset abstract domains (§ 3.4.4) are synthesized by Algorithm 2 that iteratively increments the powersets with individual intervals to avoid scalability problems faced by Z3 when optimizing multiple intervals at once.

The algorithm takes as arguments the number of intervals `k` to be included in the powerset, the number of fields in the secret n, the refinement type of the powerset domain τ, and the kind of approximation `apx` (`under` or `over`). It first runs SYNTH (§ 3.5.3) to generate the first interval, with the top level type properly propagated to the hole. If this is for an under-approximation, more such intervals can be added

Algorithm 2 Iterative Synthesis of Powersets

```
1:  procedure IterSynth(k, n, τ, apx)
2:    dom_i ← [Synth (A_P [□] [] _ _)::τ n apx]
3:    dom_o ← []
4:    for i = 2 to k do
5:      if apx == under then
6:        dom_t ← Synth (A_P (dom_i ++ □) dom_o _ _)::τ n apx
7:        dom_i ← dom_i ++ [dom_t]
8:      else
9:        dom_t ← Synth (A_P dom_i (dom_o ++ □) _ _)::τ n apx
10:       dom_o ← dom_o ++ [dom_t]
11:     end if
12:   end for
13:   return (A_P dom_i dom_o _ _)
14: end procedure
```

to the powerset to boost the precision. Conversely, if the first synthesized interval is an over-approximation, then more intervals can be eliminated from the powerset to return a more precise over-approximation. At each iteration, the algorithm creates a new placeholder interval $\square$ and Synth solves it, incrementally building up the inclusion list `dom_i`, or the exclusion list `dom_o`. Finally, the powerset is returned after k iterations. This is Anosy's general synthesis algorithm since for $k = 1$ the returned powerset has a single interval.

As a final step, the returned powerset is lifted to the Haskell source and substituted in the sketch in § 3.5.2, which as a sanity check is validated by Liquid Haskell.

Discussion Traditional abstract interpretation based techniques will refine the domains, as the query is evaluated with small step semantics, leading to imprecision at each step. In contrast, Anosy is more precise (as we show in § 3.6), because the final abstract domain is synthesized in the final step after accumulating constraints. However, Z3 does not give precise solutions when there are too many maximize/minimize directives (more than 6 in our experience) and it does not handle non-linear objectives well. We leave exploration of better optimization algorithms to future work.

3.6 Evaluation

We empirically evaluated ANOSY's performance using two case studies. In the first one (§ 3.6.1), we analyze efficiency and precision of ANOSY when verifying and synthesizing ind. sets using a set of micro-benchmarks from prior work. In the second one (§ 3.6.2), we use the ANOSY monad to construct an application that performs multiple queries (similar to those of § 3.2) while enforcing a security policy on the attacker's knowledge. With this case study, we evaluate how losses of precision introduced by ANOSY's abstract domains affect the ability of answering multiple queries.

Experimental setup ANOSY is a GHC plugin built against GHC 8.10.1. All refinement types were verified with LiquidHaskell 0.8.10. Z3 4.8.10 was used to synthesize the bounds of the abstract domains. All experiments were performed on a Macbook Pro 2017 with 2.3 GHz Intel Core i5 and 8GB RAM.

3.6.1 Verification & Synthesis of ind. sets

In this case study, we analyze the ANOSY's performance with respect to the verification and synthesis of ind. sets.

#	Under-approximation				Over-approximation			
	Size	% diff.	Verif. time	Synth. time	Size	% diff.	Verif. time	Synth. time
B1	259 / 9620	0 / 27	2.78 ± 0.03	1.11 ± 0.01	259 / 13505	0 / 2	2.64 ± 0.03	1.07 ± 0.01
B2	2.21e+05 / 1.01e+07	78 / 58	3.62 ± 0.02	9.26 ± 0.04	2.02e+06 / 2.54e+07	100 / 5	3.17 ± 0.02	4.00 ± 0.12
B3	4 / 664	0 / 25	3.12 ± 0.06	0.90 ± 0.07	4 / 888	0 / 0	2.83 ± 0.03	0.90 ± 0.01
B4	3.53e+04 / 1.35e+05	100 / 100	3.66 ± 0.04	20.92 ± 0.11	9.22e+12 / 2.81e+13	67200 / 0	3.29 ± 0.08	10.87 ± 0.01
B5	360 / 5.04e+06	83 / 25	3.81 ± 0.04	1.38 ± 0.04	35460 / 6.72e+06	1542 / 0	3.47 ± 0.04	0.89 ± 0.01

(a) Interval abstract domain

#	Under-approximation				Over-approximation			
	Size	% diff.	Verif. time	Synth. time	Size	% diff.	Verif. time	Synth. time
B1	259 / 13246	0 / 0	4.51 ± 0.05	1.13 ± 0.02	259 / 13505	0 / 2	4.34 ± 0.03	1.08 ± 0.01
B2	6.78e+05 / 1.62e+07	33 / 33	5.32 ± 0.09	14.34 ± 0.11	1.80e+06 / 2.54e+07	78 / 5	5.17 ± 0.02	4.89 ± 0.09
B3	4 / 880	0 / 0	5.29 ± 0.09	1.07 ± 0.03	4 / 888	0 / 0	4.99 ± 0.03	1.03 ± 0.01
B4	3.88e+05 / 4.00e+05	100 / 100	5.78 ± 0.03	54.89 ± 0.23	9.22e+12 / 2.81e+13	67200 / 0	5.48 ± 0.08	30.57 ± 0.07
B5	720 / 6.70e+06	67 / 0	6.02 ± 0.07	13.26 ± 0.09	6300 / 6.72e+06	192 / 0	5.96 ± 0.04	15.25 ± 0.03

(b) Powerset of intervals with size 3

Figure 3.5: Ind. sets synthesis and verification of posteriors. Column *Size* reports the size of the synthesized ind. sets, where x is the size of the `True` set and y of the `False` set in x/y. *% diff* shows the percentage difference of the size from precise ind. set in Table 3.1 (lower value is better). *Verif. time* and *Synth. time* columns report (in seconds) the median and the semi-interquartile over 11 runs.

Table 3.1: Number of fields in the secret, and size of the precise ind. sets x/y for our benchmarks, where x and y denotes the number of secrets that evaluate to `True` and `False`, respectively.

#	Name	No. of fields	Size of ind. sets
B1	Birthday	2	259 / 13246
B2	Ship	3	1.01e+06 / 2.43e+07
B3	Photo	3	4 / 884
B4	Pizza	4	1.37e+10 / 2.81e+13
B5	Travel	4	2160 / 6.72e+06

Benchmark Programs Our benchmarks consist of 5 problems from [68], which represent a diverse set of queries (B3 and B4 come from a targeted advertisement case study from Facebook [23]). We selected these benchmarks to illustrate that ANOSY supports similar classes of queries as existing prior work and to compare performance and precision with available tools.

(B1) *Birthday* checks if a user's birthday, the secret, is within the next 7 days of a fixed day[2].

(B2) *Ship* calculates if a ship can aid an island based on the island's location and the ship's onboard capacity.

(B3) *Photo* checks if a user would be possibly interested in a wedding photography service by checking if they are female, engaged, and in a certain age range.

(B4) *Pizza* checks if a user might be interested in ads of a local pizza parlor, based on their birth year, the level of school attended, their address latitude and longitude (scaled by 10^6).

(B5) *Travel* tests for a user interest in travels by checking if the user speaks English, has completed a high level of education, lives in one of several countries, and is older than 21.

[2]We only use the deterministic version of the *Birthday* problem.

For each problem, we encode the query as a Haskell function with the appropriate refinement type [105] where the secret domain is represented as a Haskell datatype for which we use the same bounds as [68]. Table 3.1 reports the number of fields in the secret, and the size of the precise ind. sets for each benchmark as x/y, where x denotes the size of the precise ind. set for the `True` response from query and y is the size when the query responds `False`.

Experiment For each benchmark, we use ANOSY to (1) synthesize the under- and over-approximated ind. sets for both results `True` and `False` and (2) verify that the synthesized approximations match the refinement types from § 3.4. We run each benchmark 11 times to collect synthesis and verification times. We use a 10 second timeout for each Z3 call. The goal is to evaluate the precision of the synthesized ind. set and time taken for synthesis and verification to run.

Intervals Figure 3.5a reports the results of our experiments for both the under- and over-approximated ind. sets using the interval abstract domain. Specifically, the column *Size* reports the number of secrets in the approximated ind. set, the column *Verif. time* reports the time (in seconds) LiquidHaskell takes to verify the posteriors, and the column *Synth. time* reports the time (in seconds) taken for synthesizing the approximate ind. sets. The *% diff.* column lists the difference in size of the approximate ind. sets with the exact ones from Table 3.1. The lower the *% diff.* column value, the more precise is the synthesized ind. set, i.e., it is closer to the ground truth.

For all our benchmarks, LiquidHaskell quickly verifies the correctness of the posteriors, in less than 4 seconds on average. In some cases, like B1 and B3, ANOSY can synthesize the exact ind. set for the `True` result using a single interval (for both approximations). For the `False` set, however, the tool returns an approximated result because the precise ind. set is not representable using intervals.

In 7 out of 10 synthesis problems, ANOSY synthesizes the approximations in less than 5 seconds. The three outliers are the synthesis of under-approximations for B2 and the synthesis of both approximations for B4. B2 uses a relational query that creates a dependency between two secret fields, where the multi-objective maximization employed by Z3 runs longer. B4 uses very large bounds (in the orders of 10^8) which result in Z3 quickly finding a sub-optimal model but timing-out before finding an optimal solution.

Powersets of intervals Figure 3.5b reports the results of our experiments using the powersets domain with 3 intervals. A higher number gives more precision for representation of the ind. set at the cost of taking more time for synthesis, due to our iterative synthesis algorithm (§ 3.5.4).

For under-approximations, ANOSY successfully synthesizes both exact ind. sets for B1 using powersets, even though the `False` set was not representable using just a single interval. For B2 and B3, the powersets significantly improve precision, i.e., we synthesize larger under-approximations.

This can be seen by comparing the *% diff.* column between Figure 3.5a and 3.5b, where the latter reports lower percentage differences from ground truth. In fact, for B3, ANOSY can almost synthesize the entire ind. set for `False` with powersets of size 3 and it can synthesize the exact ind. set with powersets of size 4 (not shown in Figure 3.5b). For B4, powersets only marginally improve precision due to SMT optimization timing out. For over-approximations, we observe a similar increase in precision, in particular in B3 and B5 where the synthesized approximations are close to the exact values. B4 slows down drastically because synthesis of each interval takes almost 10 seconds due to SMT timeouts.

Discussion ANOSY synthesizes ind. sets, and a function to compute a posterior for any prior, incurring one-time cost for synthesis but making posterior computation

free at runtime. In contrast, prior tools like `Prob` [68] need to run an expensive static analysis each time when computing the posterior knowledge. While the synthesis takes 54.2x longer on average than running Prob each time, this cost is amortized over multiples runs of the program with ANOSY.

Moreover, ANOSY is more precise than `Prob`, as demonstrated by difference from ground truth in benchmarks like B3 (Figure 3.5b). A difference of 0 indicates that ANOSY synthesized an exact ind. set. In contrast `Prob`'s belief was 0.1429 (i.e., had some uncertainty; 0 is exact) for the same example in same conditions. ANOSY is more precise because it can automatically split regions into intervals (§ 3.5.4) whose union in the powerset gives a better accuracy. For instance, in Figure 3.5b, a powerset of size 3 is enough to synthesize the exact ind. set (*% diff.* is 0) for several benchmarks.

In our experience, iterative synthesis (§ 3.5.4) works better than existing techniques [9, 68] for queries (benchmarks B1, B3, and B5) that contain point-wise comparisons, i.e., the query checks if a secret x is one of several constant values $c_1, c_2, \ldots$, or in other words, formulas of the form $x = c_1 \vee x = c_2 \vee \ldots$. These queries split the indistinguishable sets into a union of disjoint sets, and the SMT solver efficiently identified the best possible solution for the abstract domain. However, benchmarks that do not use point-wise comparisons (like B2) perform equivalent to prior work [68] in our experience.

3.6.2 Secure Advertising System

In this case study, we go back to the advertisement example in § 3.2 which we implement using ANOSY to restrict the information leaked through `downgrade`. The goal of this case study is evaluating how the choice of abstract domains affects the number of declassification queries authorized by ANOSY.

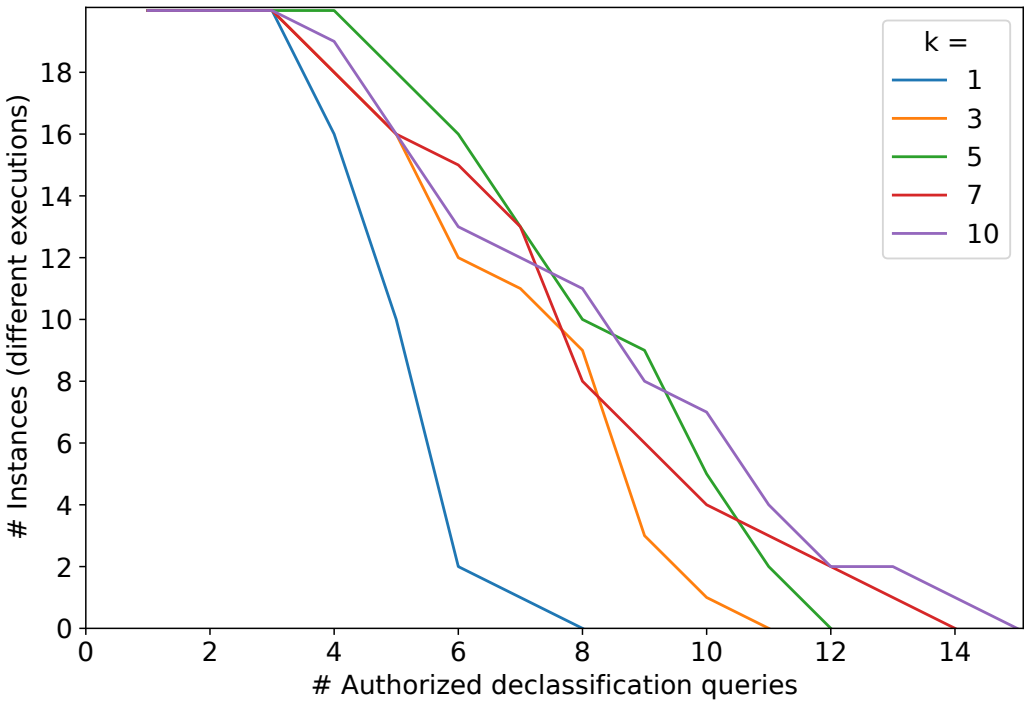


Figure 3.6: The lines show the number of execution instances (Y-axis) that were authorized for the i-th declassification query (X-axis). Each line corresponds to the under-approximated ind. set of powersets of size k.

Application We implemented the advertising query system from § 3.2 in Haskell using the `AnosyT` monad, with the `UserLoc` type as the secret. The system executes a sequence of 50 queries (one per restaurant branch): we use the `nearby` query from § 3.2 with the origin, denoting in this experiment the location of the restaurant, being a randomly generated point in the 400×400 space.

Security policy and enforcement Our program implements the security policy `qpolicy` from § 3.2, which restricts the restaurant chain from learning the user location below a set of 100 possible locations. To easily enforce the security policy, we wrapped the advertising query in the `downgrade` operation of `AnosyT` as in § 3.3.

Initially, our system starts with a prior knowledge equivalent to the entire secret domain 400×400 (i.e., the attacker does not have any information about the secret). As the system executes queries, the `AnosyT` monad tracks an under-approximation of

the attacker's posterior knowledge based on the query result and on the prior. If the posterior complies with the policy, then the monad outputs the query result and the system continues with the next query. If a policy violation is detected, the system terminates the execution.

Experiment For each experiment, we generate a new user location randomly, used as the secret, in the 400×400 space, and we run through the 50 queries for every restaurant location. For each execution, we measure after how many queries the system stops due to a policy violation. We repeat this experiment 20 times, to get the mean and standard deviation of query count and discuss them below.

Results Figure 3.6 reports the results of our experiments. The line for each k, i.e., the number of synthesized intervals in the powerset, depicts the number of experiment instances that are still running (Y-axis) after executing the i-th query (X-axis). For example, in the $k = 1$ powerset (equivalent to an interval), the system was able to answer the first 3 queries in all 20 instances without violating the policy, but only 2 instances were able to answer the 6th query.

As the size of powersets increases (from 3 to 10), the system can compute more precise under-approximations and, therefore, securely answer more queries, as can be seen in the figure. Specifically, for powerset of size $k = 3$, the system answers a maximum of 10 queries over 20 runs, with only 1 run reaching the 10th query. Similarly, the maximum number of queries answered increases to 14, due to increased precision by using powersets of size 10. Moreover, more than 10 instances answer more than 6 queries if the size of powersets goes above 3. This shows that ANOSY can be used to build a system, that can answer multiple queries sequentially with precision without violating the declassification policy.

Figure 3.6 shows a tradeoff between number of queries answered and the precision of the powersets. Higher sized powersets ($k = 7$ or 10) under-perform in the intermediate

declassifcations from 5 to 7 (on the X-axis) when compared to $k = 5$. The intersection of powersets made of k_1 and k_2 intervals produces a powerset of $k_1 k_2$ intervals, of which many intervals are small or empty (as individual powersets might have very little overlap). Hence a slightly more imprecise powerset $k = 5$ declassifies allows more instances of the query to run. However, over a longer sequence of queries a higher sized powerset performs better due to improved precision in tracking knowledge (as can be seen from $k = 10$ allowing 14 declassifcations).

3.7 Related Work

Information-flow control Language-based information-flow control (IFC) [92] provides principled foundations for reasoning about program security. Researchers have proposed many enforcement mechanisms for IFC like type systems [18, 7, 85, 65, 29, 91, 82], static analyses [56], and runtime monitors [44] to verify and enforce security properties like non-interference. The ind. sets and knowledge approximations computed by ANOSY can be used as a building block to enforce both non-interference as well as more complex security policies, as we discuss below.

Use of knowledge in IFC The notion of attacker knowledge has been originally introduced to reason about dynamic IFC policies, where the notion of "public" and "secret" information can vary during the computation [8, 44, 28]. The notion of *belief* consists of a knowledge, i.e., set of possible secret values, equipped with a probability distribution describing how likely each secret is. Existing approaches [100, 68, 43, 62] can enforce security policies involving probabilistic statements over an attacker's belief, *e.g.,* "an attacker cannot learn that a secret holds with probability higher than 0.7". We plan to deal with probability distributions in future work. However, ANOSY synthesizes a function that computes the posterior given a prior, eliminating the need to run the full static analysis for each query execution. This enables applications to

directly use knowledge based policies without expensive static analysis at runtime. Additionally ANOSY's posterior knowledge is correct-by-construction and mechanically verified using LiquidHaskell's refinement types, unlike existing tools [68] which rely on (often complex) pen-and-paper proofs.

Quantitative Information Flow approaches provide quantitative metrics, *e.g.,* Shannon entropy [94], Bayes vulnerability [95], and guessing entropy [69], that summarize the amount of leaked information. For this, several approaches [22, 9, 61] first compute a representation of a program's indistinguishability equivalence relation, whereas we represent the partition induced by the indistinguishability relation, where each ind. set is one of the relation's equivalence classes.

There are several approaches for approximating the indistinguishability relation in the literature. [22] provide techniques to approximate the indistinguishability relation for straight line programs. [9] automates the synthesis of such equivalence relations using program verification techniques, and [61] further improve the approach by combining it with sampling-based techniques. Similarly to [9], we automatically synthesize ind. sets from programs. In contrast to [22, 9, 61], the correctness of our ind. sets is also automatically and machine-checked.

Declassification Declassification is used in IFC systems to selectively allow leaks, and several extensions of non-interference account for it [8, 44, 28]; we refer the reader to [93] for a survey of declassification in IFC. Most systems treat declassification statements as *trusted*. Our work focuses on the *what* dimension of declassification, that is, our policies restrict *what* information can be declassified. In contrast, [21] enforce declassification policies that target other aspects of declassification, specifically, limiting in which context declassification is allowed and how data can be handled after declassification.

Program Synthesis ANOSY's synthesis technique follows sketch-based synthesis [98], where traditionally users provide a partial implementation with *holes* and some specifications based on which the synthesizer fills in the holes. Standard types have extensively served as a synthesis template often combined with tests, examples, or user-interaction [76, 49, 32, 67]. Refinement types provide stronger specifications, thus, as demonstrated by SYNQUID [80], do not require further tests or user information. In ANOSY, we use the refinement type synthesis idea of SYNQUID, but also mechanically generate the knowledge specific refinement types.

3.8 Conclusion & Further Applications

We presented ANOSY, a novel technique that uses the abstractions of refinement types to synthesize and statically machine-check correct approximations of knowledge and ind. sets. Using these approximations of knowledge, we defined a bounded downgrade function that can be staged on top of existing IFC systems to enforce declassification policies. We implemented ANOSY and demonstrated it runs across a variety of benchmarks from prior work and can securely answer multiple sequential queries without losing precision. We believe ANOSY represents a promising approach to embedding declassification policies in applications.

Though we only used ANOSY's precise, explicit representation of knowledge for declassification, such a representation is at the core of many information flow control tasks. Enforcing probabilistic policies requires combining knowledge, computed by ANOSY, with a probability distribution [68]. Moreover, dynamic security policies can be enforced by keeping track of attacker knowledge and comparing it with the current policy [44]. Finally, approximations of classical quantitative information flow measures, such as Shannon entropy [94], can be derived from the user's knowledge, i.e., by counting the number of concrete elements represented by the knowledge.

Chapter 4

ABSYNTHE: Abstract Interpretation-Guided Synthesis

4.1 Introduction

In recent years, there has been a significant interest in automatically synthesizing programs from high-level specifications, which often take the form of logical formulas [33], type signatures [80], or even input-output examples [37]. Program synthesis has seen significant success in domains such as spreadsheets [45], compilers [79] or even database access programs [49]. Much of the prior work, however, requires a complete and accurate embedding of the source language in the logic of the underlying solver the synthesis tool uses. These often range from symbolic execution [102], counter-example guided synthesis [97], or over-approximate semantics as predicates [58, 80, 32] (often requiring termination measures and additional predicates for verifcation). This is infeasible for many industrial-grade languages such as Ruby or Python. Other approaches are strongly coupled with the semantics of the source language with purpose-built solvers [90], but this necessarily ties the synthesis engine to the particular language model used.

In this chapter, we propose ABSYNTHE, an alternative approach based on user-defined abstract semantics that aims to be both lightweight and language agnostic. The abstract semantics are lightweight to design, simplifying away inconsequential language details, yet effective in guiding the search for programs. The synthesis engine is parameterized by the abstract semantics [25] and independent of the source language. In ABSYNTHE, users define a synthesis problem via concrete test cases and an abstract specification in some user-defined abstract domain. These abstract domains, and the semantics of the target language in terms of the abstract domains, are written by the user in a DSL. Moreover, the user can define multiple simple domains, each defining a partial semantics of the language, which they can combine together as a product domain automatically. ABSYNTHE uses these abstract specifications to automatically guide the search for the program using the abstract semantics. The key novelty of ABSYNTHE is that it separates the search procedure from the definition of abstract domains, allowing the search to be guided by any user-defined domain that fits the synthesis task. More specifically, the program search in ABSYNTHE begins with a hole tagged with an abstract value representing the method's expected return value. At each step, ABSYNTHE substitutes this hole with expressions, potentially containing more holes, until it builds a concrete expression without any holes. Each concrete expression generated is finally tested in the reference interpreter to check if it passes all test cases. A program that passes all tests is considered the solution. (§ 4.2 gives a complete example of ABSYNTHE's synthesis strategies).

We formalize ABSYNTHE for a core language $\mathcal{L}_f$ and define an abstract interpreter for $\mathcal{L}_f$ in terms of abstract transformer functions. Next, we describe a DSL $\mathcal{L}_{meta}$ used to define these abstract transformers. Notably, as ABSYNTHE synthesizes terms at each step, it creates holes tagged as abstract variables $\tilde{x}$, i.e., holes which will be assigned a fixed abstract values later. We give evaluation rules for these transformers written in $\mathcal{L}_{meta}$, that additionally narrows these abstract variables to sound range of

abstract values. For example, given a specification that requests Pandas programs that should evaluate to a data frame, a term `(`$\square : \tilde{x}_1$`).query(`$\square : \tilde{x}_2$`)` is a viable candidate that queries a data frame. However, semantics of $\mathcal{L}_{meta}$ help with constraining the bounds on $\tilde{x}_1$ and $\tilde{x}_2$ such that these holes are substituted by values of a `DataFrame` and `String` respectively. Finally, we present the synthesis rules used by ABSYNTHE to generate such terms. Specifically, we discuss how ABSYNTHE specializes term generation based on the properties of the domain, such as a finite domain enables enumeration through domain, or a domain that can be lifted to solvers can use solver-backed operations, or domains expressed as computations not supported by dedicated SMT solvers. (§ 4.3 discusses our formalism).

We implemented ABSYNTHE as a core library in Ruby, that provides the necessary supporting classes to implement user-specific abstract domains and abstract interpretation semantics. It further integrates automatic support for $\top$ and $\bot$ values and abstract variables , as well as `ProductDomain` to combine the individual domains point-wise. The ABSYNTHE implementation has interfaces to call a concrete interpreter with a generated program to check if a program satisifies the input/output examples. Finally, we also discuss some optimizations to scale ABSYNTHE for practical problems, such as caching small terms and guessing partial programs based on testing predicates on the input/output examples, and some limitations of the tool. (§ 4.4 discusses our implementation).

We evaluate ABSYNTHE as a general-purpose tool on a diverse set of synthesis problems while being at par on performance with state-of-the-art tools. We first use ABSYNTHE to solve the SyGuS strings benchmarks [5] using simple domains such as string prefix, string suffix, and string length to guide the search. Though ABSYNTHE operates with minimal semantic information about SyGuS programs, it still performs similar to enumerative search solvers such as EUPHONY [3], solving most benchmarks in less than 7 seconds. SMT solvers such as CVC4, or BLAZE that rely on precise

abstractions perform much faster than ABSYNTHE, but require large specification effort. We further evaluate the impact of our performance optimizations and verify that ABSYNTHE's synthesis cost adjusts with the expressiveness of the domain. More specifically, the string prefix and suffix domains written in Ruby generate a concrete candidate 0.41ms average, whereas string length domain being a solver-aided domain takes around 16.93ms per concrete candidate on average due to calls to Z3. Next, we use ABSYNTHE synthesize an unrelated benchmark suite, for which it is harder to write precise formal semantics—Python programs that use Pandas, a data frame manipulation library. We evaluate ABSYNTHE on the AUTOPANDAS benchmarks [13], a suite of Pandas data frame manipulating programs in Python. The AUTOPANDAS tool trains deep neural network models to synthesize Pandas programs. ABSYNTHE is at par with AUTOPANDAS, including a significant overlap in the benchmarks both tools can solve, despite using simple semantics such types and column labels of a data frame while running on a consumer Macbook Pro without specialized hardware requirements. (§ 4.5 discusses our evaluation).

In summary, we think ABSYNTHE represents an important step forward in the design of practical synthesis tools that provide lightweight formal guarantees while ensuring correctness from tests.

4.2 Overview

In this section, we demonstrate ABSYNTHE by using it to synthesize data frame manipulation programs in Python using the Pandas library [87]. In this example, we abstract data frames as sets of column names, and use a lightweight type system for Pandas API methods to effectively guide synthesis.

A *data frame* is a collection of data organized into rows and columns, similarly to a database table. Data frame manipulation is a key task in data wrangling, a preliminary

	id	valueA
0	255	1141
1	91	1130
2	347	830
⋮	⋮	⋮
8	225	638
9	257	616

(a) Data frame `arg0`

	id	valueB
0	255	1231
1	91	1170
2	5247	954
⋮	⋮	⋮
12	211	575
13	25	530

(b) Data frame `arg1`

	id	valueA	valueB
0	255	1141	1231
1	91	1130	1170
2	347	830	870
5	159	715	734
8	225	638	644

(c) Output data frame

```
1 'valueA != valueB'
```

(d) Query string `arg2`

```
1 def goal(arg0, arg1, arg2):
2   return arg0.merge(arg1, on=['id']).query(arg2)
```

(e) Synthesized program

Figure 4.1: Data Frame Manipulation Example [13]. The synthesis goal is to produce (c) given inputs (a), (b), and a query string (d). ABSYNTHE synthesizes the solution (e).

step for data science or scientific computing tasks. For example, Figure 4.1 shows a data frame manipulation synthesis task taken from the AUTOPANDAS benchmark suite [13]. The goal is to use the Python Pandas library [87] to produce the data frame in Figure 4.1c, given the two input data frames in Figure 4.1a and 4.1b and a query string (Figure 4.1d). In this case, the output joins the input rows with the same `id` but with different values in `valueA` and `valueB` columns. The Pandas library provides a wide range of methods that perform complex data frame manipulation. For example, calling `left.merge(right, on: ['col'])` joins the data frames `left` and `right` on column `col`. As another example, calling `df.query(str)` returns a new data frame with the rows of `df` that satisfy query string predicate `str` (as in Figure 4.1d).

To keep the synthesis task tractable, ABSYNTHE restricts its search to Python code consisting of input variables `arg0` through `arg2`; constants such as column names `'id'`, `'valueA'`, and `'valueB'` or row labels 0, 1, ..., 13 from the data frames; array literals and indexing; and dictionaries (for keyword arguments). Additionally, for this discussion we will limit ABSYNTHE to the `merge` and `query` methods just mentioned, even though our evaluation (§ 4.5.2) supports many more methods. Nonetheless,

even with this restricted search space, naïve enumeration of possible solutions times out after 20 minutes. In contrast, using ABSYNTHE, we can guide the search using abstract interpretation to find a solution in 0.47 seconds.

Abstract Domains and Semantics The first step in using ABSYNTHE is to identify appropriate abstract domains for the abstract interpretation and implement the abstract semantics. Typically, we develop the domain by looking at the input/output examples and thinking about the problem domain. In our running example, we observe that the data frames use columns `id`, `valueA`, and `valueB`, but each frame has a slightly different set of columns. This gives us the idea of introducing a domain `ColNames` that abstracts data frames to a set of column labels.

Abstract values, drawn from an abstract domain, represent a set of concrete values in the program. The abstract semantics define the evaluation rules of the program under values from this abstract domain. This approach has seen considerable success in practical static analysis tools such as ASTREÉ [26] and Sparrow [73]. Figure 4.2a shows, similar to these tools, the definition of the `ColNames` domain, which is a class whose instances are domain values. ABSYNTHE is implemented in Ruby, and ABSYNTHE domains subclass `AbstractDomain`, which provides foundational definitions such as $\top$ and $\bot$ (see § 4.4). A value in the `ColNames` domain stores the set of columns it represents in the instance variable `@cols`, which by line 2 can be read with an accessor method `cols`. All abstract domains *require* a partial ordering relation $\subseteq$ on the domain that returns true if and only if the first columns label set (`rhs.cols`) is a subset of the second set (`@cols`). Finally, the $\cup$ method returns a new abstract value containing the union of the column names of the two arguments. The $\cup$ method is optional, however we define this as it will be used in the abstract semantics.

After defining the abstract domain, next we need to define the abstract interpreter to give semantics to the target language in our abstract domain. Figure 4.2b defines

```ruby
class ColNames < AbstractDomain
  attr_reader :cols
  def initialize(cols)
    @cols = cols.to_set
  end

  def ⊆(rhs)
    rhs.cols.subset?(@cols)
  end

  def ∪(rhs)
    ColNames.new(@cols∪rhs.cols)
  end
end
```

(a) ColNames Domain

```ruby
class ColNameInterp < AbsInterp
  def self.interpret(env, prog)
    # details omitted for brevity
  end

  def self.pd_merge(left,right,opt)
    left ∪ right
  end

  def self.pd_query(df, pred)
    df
  end
end
```

(b) ColNames Abstract Semantics

```ruby
class PyType < AbstractDomain
  attr_reader :ty

  def initialize(ty)
    @ty = ty
  end

  def ⊆(rhs)
    @ty <= rhs.ty
  end
end
```

(c) PyType Domain

```ruby
class PyTypeInterp < AbsInterp
  def self.pd_merge(left,right,opt)
    DataFrame if left ⊆ DataFrame ∧
          right ⊆ DataFrame ∧
          opt ⊆ {on: Array<String>}
  end

  def self.pd_query(df, pred)
    DataFrame if df ⊆ DataFrame ∧
          pred ⊆ String
  end
end
```

(d) PyType Abstract Semantics

Figure 4.2: Abstract domain definition for column names domain (a) and types domain definition (c). Abstract semantics for the required methods are defined in (b) using ColNames domain and (d) using PyType domain.

the abstract interpreter for `ColNames` domain as `ColNameInterp` class. All abstract interpreters are defined as a subclass of `AbsInterp` class, provided by ABSYNTHE. It needs a definition of the `interpret` class method (the preceding `self.` denotes it is a class method), that given an environment `env`, and a term `prog` reduces it to a value of type `ColNames`. The `interpret` is a standard recursive interpreter, so we omit the definition of `interpret` for brevity. Then we define the `pd_merge` and `pd_query` class methods that define the operations for the Pandas `merge` and `query` methods on values from `ColNames` domain. A call to `left.merge(right, opt)` in the source term under abstract interpretation is computed via a call `pd_merge(abs(left), abs(right), abs(opt))`, where `abs()` indicates the abstract values of the arguments. In the column name abstraction, we only need to compute the column names of the resulting data frame, which is just the union of the column names of the input data frame (line 7). Notice the `opt` argument can be ignored, as it impacts how the data frames are merged in the concrete domain, but the set column names of the final data frame is unaffected. Similarly, a call to `df.query(pred)` is abstractly evaluated via a call to `pd_query(abs(df), abs(pred))`. Since the data frame returned by `query` has the same columns as its input data frame, `pd_query` simple returns the abstract data frame `df` (line 11).

ABSYNTHE can also combine multiple domains together pointwise. We observe that the Pandas API methods expect values of a specific type. Hence, we also introduce a `PyType` abstract domain as a lightweight type system for Python. Figure 4.2c defines the abstract domain, which stores a type in the `@ty` field as a type from RDL [36], a Ruby type system. We build on RDL for representing Python types because it comes with built-in representations for nominal types, generic types, etc. and a subtyping relation between them. The $\subseteq$ method for `PyType` simply calls the subtyping method $\leq$ of RDL types. The subtyping method $\leq$ is a special-case of the partial ordering relation $\subseteq$.

Figure 4.2d defines gives the abstract semantics for `merge` and `query` in the `PyType` domain. The method `pd_merge` checks that the types of `left` and `right` are subtypes of `DataFrame`, i.e., the type that represents Pandas data frames as shown in Figure 4.1, and that `opt` is a dictionary with a key `on` that admits an array of strings. If this check is satisfied, the return type is `DataFrame`. Otherwise, `pd_merge` returns `nil`, which ABSYNTHE interprets as $\top$, i.e., any value is possible. Note, in a type checker, if the arguments do not match the expected types a type error occurs. Here, in contrast, we are computing what would be a valid abstraction, and since we don't have a specific type we can assume $\top$, i.e., anything can happen. Later, during synthesis the search procedure will appropriately do the pruning by *type-checking* when it is provided a user specification. `pd_query` also checks if the receiver is a subtype of `DataFrame` and the query string is a `String`. If so, it returns `DataFrame`, otherwise it returns `nil`.

These domains are combined together using a `ProductDomain` class, provided by ABSYNTHE. Here we write $\times$ to pair elements from the `ColNames` domain and the `PyType` domain. For example, {`'id', 'valueA'`} $\times$ `DataFrame` denotes all data frames that have the columns `'id'` and `'valueA'`. The `ProductDomain` also comes with a `ProductInterp` that evaluates product domain values with respective individual semantics and combines these into a final product abstract value.

Synthesizing Solutions An ABSYNTHE synthesis problem is specified by giving input/output examples for the synthesized function. Synthesis begins by abstractly interpreting the input/output examples to compute an *abstract signature* for the function. We have automated this for the AUTOPANDAS benchmark suite. The upper-right corner of Figure 4.3 gives the abstract signature for out example. In particular, the first argument is a `DataFrame` with columns `'id'` and `'valueA'`; the second argument is a `DataFrame` with columns `'id'` and `'valueB'`; and the third argument is a `String` and has no columns. The synthesized function should return a `DataFrame`

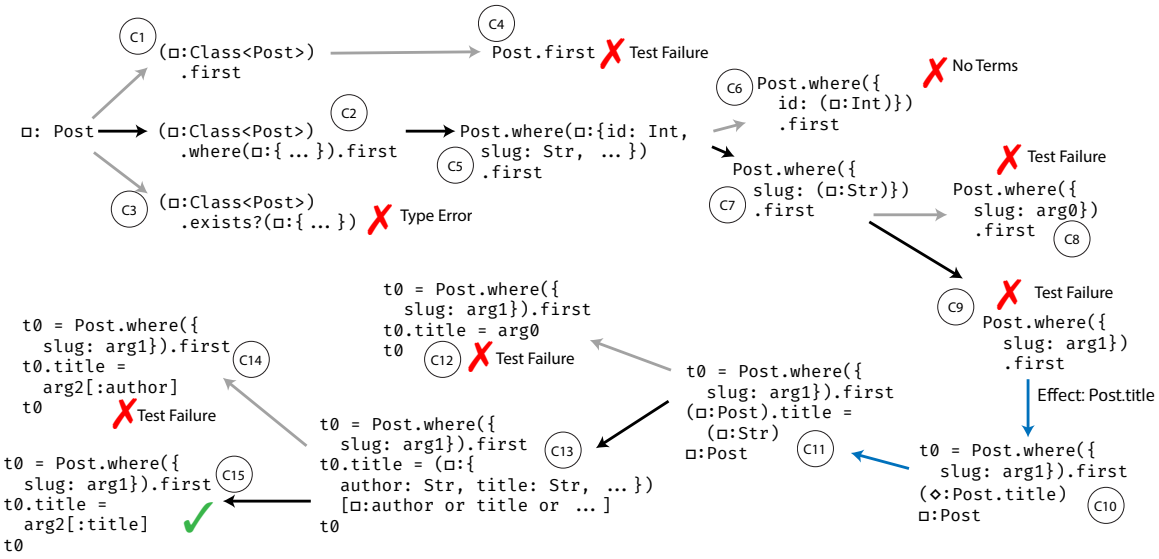


Figure 4.3: Steps in the synthesis of solution to the problem in Figure 4.1. Some choices available to the synthesis algorithm has been omitted for simplicity.

that has columns `'id'`, `'valueA'`, and `'valueB'`. Additionally, ABSYNTHE also uses a set of constants that can be used during the synthesis process. It constructs this from the rows and columns of the dataframes in the input/output example: {`'id'`, `'valueA'`, `'valueB'`, $0, 1, \ldots, 13$}.

ABSYNTHE iteratively produces candidate function bodies that may contain *holes* $\square : a$, where each hole is labeled with the abstract value a its solution must abstractly evaluate to. Synthesis begins (left side of figure) with candidate `C0`, which is a hole labeled with the abstract return value of the function. At each step, ABSYNTHE replaces a hole with an expression that satisfies its labels. For example, candidate `C1` is not actually generated because its concrete value `'valueA'` is not of type `DataFrame`. The process continues until the program has been full concretized, at which point is it tested in the Python interpreter against the input/output examples. Synthesis terminates when it finds a candidate that matches the input/output examples. For our running example, Figure 4.1e shows the solution synthesized by ABSYNTHE.

The rest of the figure illustrates the search process. The candidate `C2` does not satify the abstract specification on columns, so it is also never generated. The candidate

C3 instead expands the hole to a call to `query`, which itself has holes for the receiver and argument. Note we omit the abstraction labels here because ABSYNTHE has not fixed the abstract value for that hole yet. ABSYNTHE treats these as abstract variables that can be used during abstract interpretation, but will be eventually substituted with a fixed abstract value as the search proceeds (discussed in § 4.3).

After a single set of expansion of holes, ABSYNTHE runs the abstract interpreter on all candidates (including the partial programs). Running C3 through the abstract interpreter calls the `pd_query` function from `PyTypeInterp` (Figure 4.2d). From the evaluation of the `pd_query` ABSYNTHE can infer the first hole has to be a subtype of `DataFrame` and the second hole should be a subtype of `String`. Thus candidates like C4 will not be generated as it is ill-typed (`arg0` is a `DataFrame`).

We use filling the remaining hole in C5 to illustrate another feature of ABSYNTHE, enumerating finite abstract domains. ABSYNTHE has the upper bound of this hole at `DataFrame`, it will substitute all possible values from `PyType` that are subtypes of `DataFrame`. Since there is only one type i.e., `DataFrame`, it synthesizes expressions of that type at the hole. For the next candidate C6, again by running abstract interpreter bounds for □ are determined. The `_` in the □ signifies that `ColNames` domain still is an abstract variable, while the types have been concretized. ABSYNTHE can determine bounds for variables only if the abstract transformers have conditionals (discussed in § 4.3.1), not present in `pd_merge` of `ColNameInterp`. Running the abstract interpreter eliminates candidate C7 as the partial program will not satisfy the synthesis goal. Eliminating partial programs removes a family of concrete programs, narrowing the search space further. ABSYNTHE next generates candidates C8 and C9. C8, however, is eliminated because the `ColNames` domain interpreter computes the final data frame will have columns {`'id'`, `'valueA'`}. Eventually, the keyword argument to the merge method is filled with an array. Some ways of filling that argument fail the test cases (C10), but C11 passes all tests and is accepted as the solution (after being wrapped in

a Python method defintion), also shown in Figure 4.1e.

4.3 Formalism

In this section we formalize ABSYNTHE in a core language $\mathcal{L}_f$. Figure 4.4a shows the $\mathcal{L}_f$ syntax. Expressions in $\mathcal{L}_f$ have values v, drawn from a set of *concrete values* $\mathbb{V}$; variables x; holes $\square : a$ tagged with an abstraction a; and function application $f(e, \ldots, e)$. Note that these are external functions f, e.g., to call out to libraries. Programs in $\mathcal{L}_f$ consist of a single function definition `def` $m(x) = e$ of a function m that takes an argument x and returns the result of evaluating e.

Abstractions a include abstract values $\tilde{v}$ drawn from an abstract domain $\mathbb{A}$. We assume this domain forms a complete lattice with greatest element $\top$, least element $\bot$, and partial ordering $a_1 \subseteq a_2$. Abstractions also include *abstract variables* $\tilde{x}$, which ABSYNTHE uses to label holes whose abstractions cannot immediately be determined. For example, if ABSYNTHE synthesizes an application of a function f, it labels f's arguments with abstract variables. During synthesis, ABSYNTHE maintains bounds on such variables to narrow down the search space (see below). We refer to abstract values from § 4.2 as *abstractions* in this section to avoid the ambiguity between abstract variables and values. Concrete values are lifted to abstract values using the abstraction function α, mapping concrete values to abstract values, i.e., α maps $\mathbb{V}$ to $\mathbb{A}$. Likewise, abstract values map to a set of concrete values using the concretization function γ, i.e., γ maps $\mathbb{A}$ to the $\wp(\mathbb{V})$. We write $v \in \tilde{v}$ as a shorthand for checking that v is in the concretization of $\tilde{v}$. We assume that for each function f, we have a corresponding abstract transfer function $f^{\#}$ that soundly captures its semantics. Finally, during synthesis, ABSYNTHE maintains two variable environments: Γ, binding variables x to their abstractions, and Δ, binding abstract variables $\tilde{x}$ to their bounds. Abstract variable bounds are written as a tuple of the lower and upper bound respectively

$$
\begin{array}{llcl}
\textit{Expressions} & e & ::= & v \mid x \mid \square : a \mid f(e, \ldots, e) \\
\textit{Programs} & P & ::= & \texttt{def}\ m(x) = e \\
\textit{Concrete Values} & v & \in & \mathbb{V} \\
\textit{Abstractions} & a & ::= & \tilde{v} \mid \tilde{x} \\
\textit{Abstract Values} & \tilde{v} & \in & \mathbb{A} \\
 & & & \\
\textit{Abstraction Function} & \alpha & : & \mathbb{V} \to \mathbb{A} \\
\textit{Concretization Function} & \gamma & : & \mathbb{A} \to \wp(\mathbb{V}) \\
\textit{Inclusion} & v \in \tilde{v} & \text{if} & v \in \gamma(\tilde{v}) \\
\textit{Abstract Transfer Function} & f^{\#} & : & (\mathbb{A}, \ldots, \mathbb{A}) \to \mathbb{A} \\
 & & & \\
\textit{Abstract Environment} & \Gamma & ::= & \emptyset \mid x : a,\ \Gamma \\
\textit{Bounds Environment} & \Delta & ::= & \emptyset \mid \tilde{x} : (a, a),\ \Delta
\end{array}
$$

(a) Syntax and relations of $\mathcal{L}_f$.

$$\boxed{\Gamma \vdash e \Downarrow a}$$

$$\frac{\eta(v) = a}{\Gamma \vdash v \Downarrow a}\ \textsc{E-Val} \qquad \frac{}{\Gamma \vdash x \Downarrow \Gamma[x]}\ \textsc{E-Var} \qquad \frac{}{\Gamma \vdash \square : a \Downarrow a}\ \textsc{E-Hole}$$

$$\frac{\Gamma \vdash e_1 \Downarrow a_1 \quad \ldots \quad \Gamma \vdash e_n \Downarrow a_n}{\Gamma \vdash f(e_1, \ldots, e_n) \Downarrow f^{\#}(a_1, \ldots, a_n)}\ \textsc{E-Fun}$$

(b) Abstract semantics for $\mathcal{L}_f$.

Figure 4.4: Syntax, relations, and abstract semantics of $\mathcal{L}_f$.

(details in § 4.3.1).

Abstract Semantics Next, we define semantics to abstractly interpret candidate programs in our domain. Figure 4.4b presents the relation $\Gamma \vdash e \Downarrow a$ that, given an abstract environment Γ, evaluates an expression e to an abstract value a. E-Val lifts a concrete value to the abstract domain by applying the abstraction function. E-Var lifts a variable to an abstract value by substituting the value from the environment Γ. E-Hole abstractly evaluates a hole to its label. Finally, E-Fun recursively evaluations a function application's arguments and then applies the abstract transfer function $f^{\#}$.

Synthesis Problem We can now formally specify the synthesis problem: Given an abstract domain $\mathbb{A}$, a set of abstract transformers $f^{\#}$, and an abstract specification of the function's input and output $a_1 \to a_2$, synthesize a set of programs P such that NoHole(P), i.e., P has no holes in it, and $x : a_1, \emptyset \vdash P \Downarrow a_2$, i.e., P abstractly evaluates to a_2 given that x has abstract value a_1. Then, the final solution is chosen as a synthesized candidate P that passes all input/output examples.

4.3.1 Abstract Transformer Function DSL

Figure 4.5a shows $\mathcal{L}_{meta}$, the DSL to define abstract transformer functions $f^{\#}$ for Absynthe. The primary purpose of the DSL is to let users define $f^{\#}$ that can handle both abstract values $a$and variables $\tilde{x}$ correctly. It is expressive enough to write the abstract transformer function for domains in § 4.2. Expressions $\hat{e}$ in $\mathcal{L}_{meta}$ can be either such abstractions a, variables y, function application $g(\hat{e})$ and if-then-else statements. We consider g as uninterpreted abstract functions. The conditionals b for if statements include `top?` that tests if an expression is $\top$, `bot?` that tests if an expression is $\bot$, `var?` that tests if an expression is an abstract variable $\tilde{x}$, and `val?` that tests if an expression is a abstract value a. Additionally, expressions $\hat{e}$ can test for ordering using $\subseteq$ or can call an abstract function $g(\hat{e})$. The else branch of these conditionals evaluate to $\top$, i.e., it evaluates to the largest possible abstraction $\top$ if a test of ordering fails. This is done to soundly over-approximate program behavior, while sacrificing precision. The abstract transformer is defined as a function $f^{\#}$ that takes the input abstract value as argument y and computes the output abstraction by evaluating the expression $\hat{e}$.

Figure 4.5b shows selected big-step evaluation rules for the abstract transformer functions written in $\mathcal{L}_{meta}$. Under an abstract environment Γ and a bounds environment Δ, expression $\hat{e}$ evaluates to a new bounds environment and a value v. In general these rules reflect standard big step semantics, except for the $\subseteq$ operation, where the

$$
\begin{array}{llll}
\textit{Expressions} & \hat{e} & ::= & a \mid y \mid g(\hat{e}) \mid \texttt{if}\ b\ \texttt{then}\ \hat{e}\ \texttt{else}\ \hat{e} \\
 & & \mid & \texttt{if}\ \hat{e} \subseteq \hat{e}\ \texttt{then}\ \hat{e}\ \texttt{else}\ \top \mid \texttt{if}\ g(\hat{e})\ \texttt{then}\ \hat{e}\ \texttt{else}\ \top \\
\textit{Conditionals} & b & ::= & \texttt{top?}\ \hat{e} \mid \texttt{bot?}\ \hat{e} \mid \texttt{var?}\ \hat{e} \mid \texttt{val?}\ \hat{e} \\
\textit{Transfer Functions} & \hat{P} & ::= & \texttt{def}\ f^{\#}(y) = \hat{e}
\end{array}
$$

(a) Syntax of $\mathcal{L}_{meta}$.

$$\boxed{\Gamma \vdash \langle \Delta, \hat{e} \rangle \Downarrow \langle \Delta, a \rangle}$$

$$\frac{\Gamma[y] = a}{\Gamma \vdash \langle \Delta, y \rangle \Downarrow \langle \Delta, a \rangle}\ \text{A-VAR} \qquad \frac{\Gamma \vdash \langle \Delta_1, \hat{e} \rangle \Downarrow \langle \Delta_2, a \rangle}{\Gamma \vdash \langle \Delta_1, g(\hat{e}) \rangle \Downarrow \langle \Delta_2, g(a) \rangle}\ \text{A-FUNC}$$

$$\frac{\begin{array}{c}\Gamma \vdash \langle \Delta_1, b \rangle \Downarrow \langle \Delta_2, \texttt{true} \rangle \\ \Gamma \vdash \langle \Delta_2, \hat{e}_1 \rangle \Downarrow \langle \Delta_3, v \rangle\end{array}}{\Gamma \vdash \langle \Delta_1, \texttt{if}\ b\ \texttt{then}\ \hat{e}_1\ \texttt{else}\ \hat{e}_2 \rangle \Downarrow \langle \Delta_3, v \rangle}\ \text{A-IFT}$$

$$\frac{\Gamma \vdash \langle \Delta, \hat{e} \rangle \Downarrow \langle \Delta, \top \rangle}{\Gamma \vdash \langle \Delta, \texttt{top?} \rangle \Downarrow \langle \Delta, \texttt{true} \rangle}\ \text{A-TOPT}$$

$$\frac{\begin{array}{c}\Gamma \vdash \langle \Delta_1, \hat{e}_1 \rangle \Downarrow \langle \Delta_2, \tilde{x} \rangle \\ \Gamma \vdash \langle \Delta_2, \hat{e}_2 \rangle \Downarrow \langle \Delta_3, \tilde{v} \rangle \\ \Delta_3[\tilde{x}] = (a_2, a_3) \quad a_2 \subseteq \tilde{v} \subseteq a_3 \\ \Delta_4 = \Delta_3[\tilde{x} \mapsto (a_2, \tilde{v})]\end{array}}{\Gamma \vdash \langle \Delta_1, \hat{e}_1 \subseteq \hat{e}_2 \rangle \Downarrow \langle \Delta_4, \texttt{true} \rangle}\ \text{A-VC}$$

$$\frac{\begin{array}{c}\Gamma \vdash \langle \Delta_1, \hat{e}_1 \rangle \Downarrow \langle \Delta_2, \tilde{x}_1 \rangle \\ \Gamma \vdash \langle \Delta_2, \hat{e}_2 \rangle \Downarrow \langle \Delta_3, \tilde{x}_2 \rangle\end{array}}{\Gamma \vdash \langle \Delta_1, \hat{e}_1 \subseteq \hat{e}_2 \rangle \Downarrow \langle T(\Delta_3, \tilde{x}_1, \tilde{x}_2), \texttt{true} \rangle}\ \text{A-VS}$$

$$T(\Delta, \tilde{x}_1, \tilde{x}_2) = \begin{cases} \Delta[\tilde{x}_1 \mapsto (a_3, a_4)] & if\, a_1 \subseteq a_3, a_4 \subseteq a_2 \\ \Delta[\tilde{x}_1 \mapsto (a_3, a_2), \tilde{x}_2 \mapsto (a_3, a_2)] & if\, a_3 \subseteq a_2 \subseteq a_4 \\ \Delta & if\, a_1 \not\subseteq a_4 \\ \quad where\ \Delta[\tilde{x}_1] = (a_1, a_2), \Delta[\tilde{x}_2] = (a_3, a_4) & \end{cases}$$

(b) Selected $\mathcal{L}_{meta}$ evaluation rules.

Figure 4.5: Syntax and evaluation rules of $\mathcal{L}_{meta}$.

bounds get constrained because of the comparison. The rule A-IfT evaluates the branch condition b and evaluates $\hat{e}_1$ if it is `true`. A similar rule (omitted here) can be written if the conditional evaluates to `false`. A-TopT checks if the expression $\hat{e}$ evaluates to $\top$. We omit evaluation rules for the `false` case and other branching predicates such as `bot?`, `var?`, and `val?` which are similar to `top?`.

The rules for evaluating $e \subseteq e$ are most interesting, as these test for the $\subseteq$ relation while constraining abstract variables $\tilde{x}$ to the range under which the relation $e \subseteq e$ holds. In general, the abstract variable narrowing reduces the range of $\tilde{x}$ to a sound range for that evaluation through $f^{\#}$. In effect it is finding satisfiable range for $\tilde{x}$ for that branch. A-VarConst tests for the $\subseteq$ relation when $\hat{e}_1$ evaluates to a variable $\tilde{x}$ and $\hat{e}_2$ evaluates to a values $\tilde{v}$. In such a case, if $\tilde{v}$ is within the range of the variable $\tilde{x}$ the term evaluates to `true`, while updating the upper bound of $\tilde{x}$ to $\tilde{v}$. This narrows the abstract variables, while still being sound under which the partial order relation $\subseteq$ holds true. A similar symmetrical rule exists (omitted here) where the left hand evaluates to $\tilde{v}$ and right hand evaluates to $\tilde{x}$. Finally, A-VarSub gives the rules for comparing two abstract variables $\tilde{x}_1$ and $\tilde{x}_2$. It uses a metafunction T to describe the cases where $\tilde{x}_1$ is contained in $\tilde{x}_2$, or has some overlap, or $\tilde{x}_1$ is less than $\tilde{x}_2$.

4.3.2 Abstraction-Guided Synthesis

To perform abstraction-guided synthesis, Absynthe recursively replaces holes by suitable expressions and then tests fully concretized candidates. Figure 4.6 shows the rules for hole replacement. These rules prove judgments of the form $\Delta, \Gamma \vdash e_1 \rightsquigarrow e_2 : a$, meaning in bounds environment Δ and abstract environment Γ, expression e_1 takes a step by replacing a hole in e_1 to yield a new expression e_2. In particular, S-Val replaces $\square : a$ with a value v from the concrete set that $a$abstracts. Similarly, S-Var replaces a hole with a variable that is compatible with the hole's label.

The next few rules are used to generate function applications, or more generally,

$$\boxed{\Delta, \Gamma \vdash e \rightsquigarrow e : a}$$

$$\frac{v \in \gamma(a') \quad a' \subseteq a}{\Delta, \Gamma \vdash \square : a \rightsquigarrow v : \alpha(v)}\ \text{S-Val} \qquad \frac{\Gamma[x] = a' \quad a' \subseteq a}{\Delta, \Gamma \vdash \square : a \rightsquigarrow x : a'}\ \text{S-Var}$$

$$\frac{\begin{array}{c} f^{\#}(\tilde{x}_1, \ldots, \tilde{x}_n) = a' \quad \Delta[\tilde{x}_i] = (a_{i,1}, a_{i,2}) \\ a_{i,1} \subseteq a_i \quad a_i \subseteq a_{i,2} \quad a' \subseteq a \end{array}}{\Delta, \Gamma \vdash \square : a \rightsquigarrow f(\square : a_1, \ldots, \square : a_n) : a'}\ \text{S-Finite}$$

$$\frac{\begin{array}{c} \Delta, \Gamma \vdash \square : \tilde{x}_1 \rightsquigarrow e_1 : \tilde{x}_1 \quad \ldots \quad \Delta, \Gamma \vdash \square : \tilde{x}_{n-1} \rightsquigarrow e_{n-1} : \tilde{x}_{n-1} \\ \Gamma \vdash e_1 \Downarrow a_1 \quad \ldots \quad \Gamma \vdash e_{n-1} \Downarrow a_{n-1} \\ f^{\#}(a_1, \ldots, a_n) = a' \quad a' \subseteq a \end{array}}{\Delta, \Gamma \vdash \square : a \rightsquigarrow f(e_1 : a_1, \ldots, \square_n : a_n) : a'}\ \text{S-Solve}$$

$$\frac{\tilde{x}_i \text{ and } \tilde{x}' \text{ is fresh}}{\Delta, \Gamma \vdash \square : a \rightsquigarrow f(\square : \tilde{x}_1, \ldots, \square : \tilde{x}_n) : \tilde{x}'}\ \text{S-Enumer}$$

Figure 4.6: Hole replacement rules for $\mathcal{L}_f$.

any term that may have more holes. First, S-Finite generates function application when the domain from which $a$is drawn is finite, *e.g.,* a simple type system that without polymorphic types or first class lambdas, or an effect system as used in Guria, Foster, and Van Horn [49]. This rule can produce multiple candidates with each hole tagged with distinct abstract values from the domain. Second, for abstract domains with infinite values that can be represented in a background theory solver, Absynthe applies the S-Solve. If the function application requires n arguments, only $n-1$ arguments are concretized to a term. This gives the constraint $f^{\#}(a_1, \ldots, a_n) = a'$ with only one unknown, a_n, that can solved for and assigned to the hole. For $f^{\#}$ to be lifted to a SMT solver $f^{\#}$ should also have an interpretation a background theory supported by the solver. This is useful for representing predicate abstractions or numeric domains such as intervals or string lengths (used in SyGuS evaluation in § 4.5.1). Finally, S-Enumer replaces a hole with a function application with fresh abstract variables $\tilde{x}_i$ for the arguments and return. Notice there is no guarantee f will produce a value of the appropriate abstraction. This is because, while we assume

we have an abstract transfer function $f^{\#}$, we do not know what abstraction it will compute without concretizing the arguments. However, unsound partial programs will be eliminated by the abstract interpreter as discussed below. Given only forward evaluation semantics and no other information about the domains, this is best way to construct partial program candidates. ABSYNTHE can switch between bottom-up synthesis (S-ENUMER) and top-down goal-directed synthesis (rest of the S- rules) depending on which rule is applied. While these rules are non-deterministic, the ABSYNTHE implementation (§ 4.4) chooses and applies these rules for the correct domain in a fixed order to yield solutions.

Algorithm 3 Synthesis of programs that passes a spec s

1: **procedure** GENERATE($a_1 \to a_2$, maxSize)
2: $\Gamma \leftarrow [x \mapsto a_1]$
3: $e_0 \leftarrow \square : a_2$
4: workList $\leftarrow [e_0]$
5: **while** workList is not empty **do**
6: $e_{curr} \leftarrow$ pop(workList)
7: $\omega_{enumer} \leftarrow \{e_t \mid \Gamma \vdash e_{curr} \rightsquigarrow e_t : a\}$
8: $\omega_{valid} \leftarrow \{e_t \in \omega_{enumer} \mid \Gamma \vdash e_t \Downarrow a \wedge a \subseteq a_2\}$
9: $\omega_{eval} \leftarrow \{e_t \in \omega_{valid} \mid \text{NOHOLE}(e_t)\}$
10: $\omega_{rem} \leftarrow \omega_{valid} - \omega_{eval}$
11: **for all** $e_t \in \omega_{eval}$ **do**
12: **return** e_t if TESTPROGRAM(e_t)
13: **end for**
14: $\omega_{rem} \leftarrow \{e_t \in \omega_{rem} \mid \texttt{size}(e_t) \leq \text{maxSize}\}$
15: workList $\leftarrow$ reorder(workList $+ \omega_r$)
16: **end while**
17: **return** Error: No solution found
18: **end procedure**

Synthesis Algorithm Algorithm 3 performs abstraction-guided synthesis. The algorithm uses a work list and combines synthesis rules for candidate generation with search space pruning based on abstract interpretation, in addition to testing in a concrete interpreter. The ordering of programs in the worklist determines the order in which program candidates are explored (discussed in § 4.4). The synthesis algorithm

starts off with an empty candidate e_0 as a base expression in the work list. At every iteration it pops one item from the work list and applies synthesis rules (Figure 4.6) in a non-deterministic order to produce multiple candidates ω_{enumer}. Each candidate is abstractly interpreted, and then checked to see if the computed abstraction satisfies the goal abstraction. If it is satisfied it is added to the set of valid candidates ω_{valid} (line 8). As partial programs with holes represent a class of programs, abstractly interpreting these eliminate a class of programs if they are not included in the goal a_2. Thus, the algorithm iterates through partial programs which are *sound* with respect to the abstract specification. Any unsound programs generated by S-ENUMER are pruned here.

Finally, all concrete programs ω_{eval} are tested in the interpreter to check if a program satisfies all test cases, in which case it is returned as the solution. The remaining programs ω_{rem} contain holes, so these can be expanded further by the application of synthesis rules. Only programs below the maximum size of the search space are put back into the work list, and the order of the work list is always based on some domain-specific heuristics (§ 4.4 discusses our program ordering).

4.4 Implementation

ABSYNTHE is implemented in approximately 3000 lines of Ruby excluding dependencies. It is architected as a core library whose interfaces are used to build a synthesis tool for a problem domain. Additionally, to support solver-backed domains, we developed a library (~460 lines) to lazily convert symbolic expressions to Z3 constraints and solve those in an external process. ABSYNTHE uses a term enumerator that, at each step, visits holes in a term and substitutes it with values or subterms containing more holes applying the rules shown in § 4.3.2. ABSYNTHE requires users to define a translation from the ASTs to the source program and a method that tests a candidate to return

if the test passed or not. Users may provide a set of constants for the language which are used as values to be used in the concretization function. In practice, this is useful when the language has infinite set of terminals (like Python), and selecting values from the set of constants makes the term generation tractable. For AUTOPANDAS benchmarks, we infer such constants from the data frame row and column labels (§ 4.2).

ABSYNTHE explores program candidates in order of their size, preferring smaller programs first (line 15 of Algorithm 3). We plan to explore other program exploration order in future work. The synthesis rules presented in § 4.3.2 are non-deterministic, however, our implementation fixes an order of application such rules. It prefers to synthesize constants and variables followed by function applications, hashes, arrays, etc. Moreover, based on the definition of abstract domains (discussed below), it can automatically choose to apply the S-FINITE or S-SOLVER rules. If none of these specialized rules apply, it uses S-ENUMER rule to synthesize subterms.

Abstract Domains To guide the search, users need to implement an abstract domain. ABSYNTHE provides a base class—`AbstractDomain` from which a programmer can inherit their own abstract domains implementation, like Figure 4.2. The base classes come with machinery that gives built-in implementation of $\top$, $\bot$, abstract variables $\tilde{x}$, and supporting code for partial ordering between these abstract values. The user has to define how to construct abstract values for that domain (the `initialize` method in § 4.2), the partial ordering relation $\subseteq$ between two abstract *values*. The abstract variable narrowing (§ 4.3.1) is implemented as the $\subseteq$ method in the `AbstractDomain` base class. Solver-aided domains (such as string length in § 4.5.1) construct solver terms when initializing an abstract value, or apply functions that compute abstract values (including $\cup$ and $\cap$). These terms are checked for satisfiability of $a_1 \subseteq a_2$ in the solver when the $\subseteq$ method is invoked, and any solved abstract

variables are assigned to its holes. If the solver proves the solver term unsatisfiable, the candidate is eliminated. The rule S-FINITE is applied for domains with finite abstract values and S-SOLVE is used for domains whose values can be inferred using an SMT solver yielding top-down goal-directed synthesis. In case these cannot be applied, ABSYNTHE falls back to using the S-ENUMER rule that is equivalent to bottom-up term enumeration. We plan to explore a more ergonomic API for the ABSYNTHE framework in future work.

ABSYNTHE also provides a `ProductDomain` class to automatically derive product domains by combining any user-defined domains as needed. The $\subseteq$ method on `ProductDomain` returns the conjunction of respective $\subseteq$ on the individual domains it is composed of.

Abstract Interpreters Each abstract domain needs a definition of abstract semantics, inherited from the `AbstractInterp` class provided by ABSYNTHE (as shown in § 4.2). All subclasses override the `interpret` method that takes as argument the abstract environment and the AST of the term that is being evaluated. In practice, it is implemented as evaluating subterms recursively, and then applying the abstract transformer function written in a subset of Ruby (similar to $\mathcal{L}_{meta}$ in § 4.3.1) to evaluate the program in the abstract domain. A sound interpreter for `ProductDomain` is derived automatically, by composing the interpreters of its base domains. More specifically, it evaluates the term under individual base domains and then combines the results pointwise into a product.

Concrete Tests Any synthesized term without holes that satisfies the abstract specification is tested by ABSYNTHE in a reference interpreter against concrete test cases. ABSYNTHE expects the programmer to define a `test_prog` method that calls the reference interpreter with the synthesized source program (as a string in the source language), and returns a boolean to indicate if the tests passed. The reference

interpreter runs the test case, which in many cases boils down to checking the program against the provided input/output examples. If the program passes all test cases, it is considered the correct solution. If the program fails a test, it is discarded.

Optimizations In practice, ABSYNTHE uses a min-heap to store a work list of candidates ordered by their size. This eliminates the reorder step (Algorithm 3 line 15), saving an average cost of $\mathcal{O}(n \log n)$ at each synthesis loop iteration. Additionally, we found certain common subterms occur frequently in the same program, *e.g.*, computing the index of the first space in a string in a SyGuS program. ABSYNTHE caches small terms (containing up to one function application) that do not have any holes to save the cost of synthesizing these small fragments. Whenever, a hole with compatible abstract value is found, these fragments are substituted directly without doing the repetitive work of synthesizing the function application from scratch again (similar to subterm reuse in DRYADSYNTH [52]). Finally, ABSYNTHE tests a set of predicates against given input/output examples, to guess a partial program instead of starting from just a $\square$ term. For example, ABSYNTHE has a predicate that checks if the output is contained in the input, then the output is a substring of the input. For the SyGuS language, if the predicate `(str.contains output input)` tests true, then the partial program is inferred to be `(str.substr input` $\square$ $\square$`)`. This reduces the problem complexity by cutting down the search space. Another predicate `(str.suffixof output input)` tests if the input ends with output, then it infers the partial program `(str.substr input` $\square$ `(str.len input))`, i.e., the program is possibly a substring of the input from some index to the end. We evaluate the performance impact of the latter two optimizations in § 4.5 (No Template column in Table 4.1).

Limitations While ABSYNTHE is a versatile tool to define custom abstract domains and combine it with testing in a reference interpreter, the approach does have some limitations. First, ABSYNTHE only works with forward evaluation rules over the

abstract domain, in contrast to FlashMeta [83] that requires "inverse semantics", i.e., rules that given a target abstraction computes the arguments to the abstract transformer. While specifying only the forward semantics eases the specification burden for users, it require more compute time to synthesize subterms such as arguments to functions. Second, while we found product domain useful to combine separate domains, these domains remain independent through synthesis, unlike predicates where all defined semantics can be considered at the same time. We plan to explore methods to make product domains more expressive in future work. Third, problems where one can define full formal semantics are a better fit for solver-aided synthesis tools such as Rosette [102] or SemGuS [58]. We share performance benchmarks on SyGuS strings (which have good solver-aided tools) to give some evidence for this in our evaluation (§ 4.5.1). Notably, solver-aided tools can jointly reason about subterms. In contrast, when using solver-aided domains, Absynthe concretizes some of the subterms which requires enumeration through larger number of terms. Finally, Absynthe falls back on term enumeration when abstract domains do not provide any more guidance, often leading to combinatorial explosion for larger terms.

4.5 Evaluation

We evaluate Absynthe by targeting it in variety of domains, to verify it can synthesize different workloads. The primary motivation is to evaluate the general applicability of abstract interpretation-guided synthesis to diverse problems rather than being a state-of-the-art tool at a single synthesis benchmark suite. The questions we aim to answer in our evaluation are:

- How well does Absynthe work for problems traditionally targeted using solver-based strategies using the SyGuS strings benchmark [5] (§ 4.5.1)? We also discuss the performance impact of optimizations and program exploration behavior in

ABSYNTHE.

- Can ABSYNTHE be adapted to an unrelated problem (not handled by any tools that solve SyGuS benchmarks) where it is difficult to write precise formal semantics? We test this by using ABSYNTHE to synthesize Python programs that use the Pandas library from the AUTOPANDAS [13] benchmark suite (§ 4.5.2).

4.5.1 SyGuS Strings

Benchmarks To test that ABSYNTHE is a viable approach to synthesize programs that has been well explored in prior work, we target it on the SyGuS strings benchmark suite [5]. We believe strings form a good baseline to compare ABSYNTHE with other synthesis approaches that rely on enumerative search [3], SMT solvers [89], and abstract methods directed by solvers [110] (discussed in details in § 4.6). In contrast, ABSYNTHE uses only abstract domains with their forward transformers to guide the search. We *do not expect* ABSYNTHE to out-perform the past tools, rather to evaluate if it can solve most of the benchmarks at a lower cost of defining lightweight abstract domains and partial semantics upfront.

SyGuS strings has 22 benchmarks with 4 variants of each—standard (baseline set of input/output examples), small (fewer examples than standard), long (more examples than standard), long-repeat (more examples than long with repeated examples). As our approach is dependent only on the abstract specification and testing, not on the number of examples, we show detailed results for the standard version of these benchmarks. These results generalize to all variants of each benchmark. As we aim to evaluate how abstraction guided search performs, we exclude any programs containing branches. Previous work like RBSYN [49] and EUSOLVER [3] have used test cases that cover different paths through a program to do more efficient synthesis of branching programs. These can be adapted to a system like ABSYNTHE with minor effort.

ABSYNTHE parses the SyGuS specification files directly to prepare the synthesis

goal and load the target language. As SyGuS does not come with an official concrete interpreter for programs, we provide one written in Ruby that is compliant with the SyGuS specifications [86]. ABSYNTHE uses this interpreter as a black-box and does not receive any additional feedback other than the generated SyGuS programs satisfied the input-output examples or not.

Abstract Domains We defined the following abstract domains and their semantics to run the benchmark suite:

1. **String Length.** A solver-aided domain to lift strings to their lengths, while lifting integers and booleans without transformation. This means the concretization of the abstract value 5 can be the number 5 and the set of all strings of length 5, whereas the boolean abstract value `true` or `false` represents identical concrete values.

2. **String Prefix.** A domain to represent the set of strings that begin with a common prefix. For example, an abstract value with string "fo" is wider than an abstract value with string "foo", as the former denotes all strings starting with "fo" and the latter includes a subset of that, i.e., strings starting with "foo". The $\subseteq$ operation checks if the prefix of one string starts with the prefix of the other.

3. **String Suffix.** A domain to represent the set of strings that end with a common suffix, similar to string prefix domain. The $\subseteq$ operation checks if the suffix of one string ends with the suffix of the other.

These domains were created by looking at the input/output examples in the synthesis specs, and encoding the simplest partial semantics that guides the reasoning. For example, a few problems have programs that start with or end with a string constant. This is how we designed the string prefix and suffix domains respectively. On the other hand, many problems produce strings of fixed lengths or the length of the

output string is a function of the length of the input string. The string string length domain expresses semantics constraints of this kind. As the string length domain is solver-aided, it can handle symbolic constraints from abstract variables like the string length of a substring `str.substr` operation is `j - i` where i and j are the start and end index respectively. Although the string length domain does not preserve type information, SyGuS being a typed language (type-soundness enforced by the grammar) all programs in the language are type-correct by construction. Consequently, we did not need to write a type system as an abstract domain.

Finally, we give abstract specifications in the selected abstract domains where required. Specifically, we run each benchmark without an abstract annotation, i.e., equivalent to $\top \to \top$ specification which results in naive enumeration combined with abstract interpretation. If a benchmark times out, then we add an abstract annotation, such as $\top \to$ "Dr. " for the dr-name example (Table 4.1). This specification means, ABSYNTHE should find a function that given any input string ($\top$), it computes strings starting with "Dr. " only.

Results

Table 4.1: Results of running ABSYNTHE on SyGuS strings benchmarks. *# Ex* lists the number of I/O examples; *Time* lists the median and semi-interquartile range for 11 runs; *Size* and *Ht* reports the number of AST nodes and the height of the program AST respectively; *# Tested* is the number of programs run in the concrete interpreter before a solution was found; *Domains* lists the domains used to specify the abstract spec; and *# Elim* lists the number of partial programs eliminated by the abstract interpreter during search. *No cache* and *No Temp* measure the performance of ABSYNTHE when small expression cache and template inference (§ 4.4) are disabled respectively.

Benchmark	# Ex	Time (sec)	Size	Ht	# Tested	Domains	# Elim	No Cache	No Temp
bikes	6	1.70 ± 0.02	7	4	4808	$\top$	0	2.55	35.05
dr-name	4	1.54 ± 0.02	11	4	4797	Prefix	46610	139.53	2.92
firstname	4	0.03 ± 0.00	7	3	4	$\top$	0	0.63	0.18
initials	-	-	-	-	-	-	-	-	-
lastname	4	0.02 ± 0.00	10	4	15	$\top$	0	0.81	18.72
name-combine	6	0.21 ± 0.00	5	3	566	$\top$	0	0.24	0.22
name-combine-2	4	6.01 ± 0.06	9	4	9723	Suffix	48516	6.65	8.28
name-combine-3	6	47.86 ± 0.23	9	5	117370	Suffix	124573	68.29	43.63
name-combine-4	-	-	-	-	-	-	-	-	-
phone	6	0.03 ± 0.00	4	2	3	$\top$	0	0.03	0.12
phone-1	6	0.16 ± 0.00	6	3	1189	$\top$	0	0.20	7.32
phone-2	6	0.05 ± 0.01	7	3	41	$\top$	0	0.04	63.82
phone-3	-	-	-	-	-	-	-	-	-
phone-4	6	0.05 ± 0.01	4	2	1577	$\top$	0	0.05	0.14
phone-5	7	0.03 ± 0.00	7	3	18	$\top$	0	2.16	0.20
phone-6	7	100.54 ± 0.51	14	4	5937	Length	12234	-	27.79
phone-7	7	103.92 ± 0.37	14	4	54051	Length	12639	-	-
phone-8	7	0.72 ± 0.00	10	4	217	Length	31	1.37	-
phone-9	-	-	-	-	-	-	-	-	-
phone-10	-	-	-	-	-	-	-	-	-
reverse-name	6	0.35 ± 0.00	5	3	593	$\top$	0	0.41	0.42
univ-1	6	6.69 ± 0.07	7	3	19683	$\top$	0	8.08	7.73

Table 4.1 shows the results of running the SyGuS strings benchmarks through ABSYNTHE with the discussed domains. The numbers are reported as a median of 11 runs on a 2016 Macbook Pro with a 2.7GHz Intel Core i7 processor and 16 GB RAM. All experiments had a timeout of 600 seconds. In Table 4.1, *Benchmark* column is the name of the problem, # *Ex* shows the number of input/output examples. *Time* shows the median running time of the benchmark along with the semi-interquartile range over 11 runs. The *Size* and *Ht* columns give the size of the synthesized program as the count of the AST nodes in the SyGuS language and the height of the synthesized program AST respectively. The # *Tested* column lists the number of programs that were tested in the concrete interpreter before a solution was found. An abstraction that works well reduces this number compared to a worse abstraction or naïve enumeration. *Domains* column lists the domains used for synthesizing the program. These domains were provided as a specification in the abstract domain. $\top$ denotes that an abstract specification was provided as a product of $\top$ values in all individual domains for input and output, resulting in just term enumeration. The rows which mention the domain was provided abstract specs only from that domain, resulting in guidance from the provided specification. The # *Elim* lists the number of partial programs (denotes a family of concrete programs) that were eliminated by running the abstract interpreter with the provided specification during the search. For the problems which used the $\top$ domain, the abstract interpreter did not eliminate any partial programs, as specification admits all programs. Any row with – denotes time out of the benchmark under these abstract specifications.

Most benchmarks are solved within ∼7 seconds, with exceptions being *name-combine-3*, *phone-6*, and *phone-7* which take longer. In general a larger program takes much longer to synthesize, due to combinatorial increase in the number of terms being searched through as the AST size increases. For example, larger programs with same AST height take longer to synthesize due to higher number of function

arguments. The number of examples do not impact the time for synthesis as most time is spent in abstract interpretation and term generation. Testing a candidate on the examples take minimal time. ABSYNTHE performs reasonably well, solving around the same number of benchmarks as EUSOLVER [5]. We selected EUSOLVER as it is based on an enumerative search method like ABSYNTHE. The timeout of 600 seconds only applies to our ABSYNTHE evaluation, whereas EUSOLVER was evaluated with a timeout of 3600 seconds. ABSYNTHE solves around 77% of the benchmarks despite being a tool written a Ruby, one of the more slower languages. We suspect additional performance gains can be had by writing the tool in performant language that compiles to native code. We plan to explore this in future work. Additionally, ABSYNTHE does not have the problem of overfitting because the search algorithm does not use the input/output examples. It merely uses it as a test case, and since they do not influence term enumeration they do not cause overfitting with respect to the examples.

Domain-specific synthesis costs Another key advantage of the ABSYNTHE approach is only pay for what you use. The time of synthesis is dependent on the semantics of the abstract domain. String prefix and suffix are implemented in pure Ruby and does not incur much cost for invoking the solver, so these still guide the search without much cost. However, the string length domain being a solver-backed domain, requires a call to Z3 for every $\subseteq$ check. So it give more precise pruning, while taking a longer time for synthesis. Comparing the average time to generate all the concrete programs explored gives evidence for this. For example, consider *phone-6* which explores 5937 candidates in 100.54 seconds (16.93ms average) with the string length domain, whereas *name-combine-3* explores 117370 candidates in 47.86 seconds (0.41ms average) with the string suffix domain. Depending on how expensive a domain is, one can combine the domains to fit in a variety of synthesis time budgets.

Impact of performance optimizations We explore the impact of performance optimizations discussed in § 4.4. First, the performance of ABSYNTHE on these benchmarks when the small expressions cache is disabled is reported in the *No cache* column. It is slower than the baseline across all benchmarks. Notably, *phone-6* and *phone-7* reuse function application subterms. So without caching small expressions, these two benchmarks do repetitive work synthesizing the same expressions in different call sites, resulting in a timeout. Second, the *No Temp* column reports the performance numbers of ABSYNTHE when it is run on these benchmarks with the template inference by testing predicates is disabled. It is slower on most benchmarks than the baseline, and even causing timeouts on some (*phone-7* and *phone-8*). The exceptions are *phone-6* and *name-combine-3*, where the no templates version is faster than the baseline. Recall, that the inferred templates have holes, that have are tagged with a fresh abstract variable $\tilde{x}$ resulting in enumeration of more terms. In contrast, the candidate generation rules (S-) applied during the program search that may potentially synthesize holes with more precise abstractions resulting in less terms being enumerated. We plan to explore mechanisms to infer template holes with more precise abstractions in future work.

4.5.2 AutoPandas

Benchmarks We want to test if the approach used by ABSYNTHE, of guiding the search with lightweight abstract semantics combined with testing to ensure correctness, is general enough to be useful for another domain. For this purpose we use the AUTOPANDAS [13] benchmark suite from its artifact [1] as a case study. The benchmarks are sourced from StackOverflow questions containing the `dataframe` tag. Each benchmark contain the input data frames, additional arguments, the expected data frame output, the list of Pandas API methods to be used in the program, and

[1] GitHub: https://github.com/rbavishi/autopandas

the number of method calls in the final program.

Bavishi et al. [13] define *smart operators* to generate candidates and train neural models from a graph-based encoding on synthetic data to rank generated candidates. For a baseline, they consider an enumerative search synthesis engine that naïvely enumerates all possible programs using the methods specified in the benchmark. This narrows down the search space to a permutation of 1, 2, or 3 method calls specified upfront, instead of search over all supported Pandas API. In contrast, ABSYNTHE works like enumerative search, but large classes of programs are eliminated by abstract interpretation of partial programs, or terms are constructed guided by the abstract semantics. Unlike SyGuS, all benchmarks in AUTOPANDAS have only one input and output example. The synthesis goal is a multi-argument Python method that given the specified input produces the desired output.

The evaluation of AUTOPANDAS benchmarks uses the same ABSYNTHE core as the SyGuS evaluation. We wrote a test harness in Python that loads the AUTOPANDAS benchmarks written in Python and communicates with ABSYNTHE core running as a child process. The ABSYNTHE core is responsible for doing the enumerative search, while eliminating programs using abstract interpretation. Any concrete program generated by ABSYNTHE is tested in the host Python interpreter. These operations are performed as inter-process communication over Unix pipes between the host Python harness process and the child ABSYNTHE Ruby process. This allows the testing of generated programs in the host Python process, saving the overhead of launching a new Python process and importing Pandas packages (about 1-3 seconds) for every candidate. If the input/output examples are satisfied the synthesis problem is solved, else control is returned back to ABSYNTHE which searches and sends the next candidate for testing.

Abstract Domains The abstract domains used for AUTOPANDAS benchmarks are:

1. **Types.** A domain to represent the data type of the computed values (Figure 4.2c).

2. **Columns.** A domain to represent dataframes as a set of their column labels (Figure 4.2a).

Our Python harness infers the data types and the column labels from the input/output examples and the ABSYNTHE core constructs the abstract domain values from `PyType` and `ColNames` domains respectively. These individual domains are combined pointwise using the product domain `PyType` $\times$ `ColNames`, and ABSYNTHE soundly applies the individual abstract semantics to compute values in the same product domain. The types domain in ABSYNTHE is a wrapper around types from RDL [36], a type system for Ruby. ABSYNTHE uses RDL as a library to build the `PyType` class (the `ty` field holds an RDL type as shown in Figure 4.2c). This allows us to reuse prior work that defines nominal types, generic types, finite hash types, singleton types, and their subtyping relations. We define the semantics for these RDL types for the Python language in an abstract interpreter `PyTypeInterp` to handle features such as standard method arguments, optional keyword arguments, and singleton types as arguments (like `int`). We define the concretization function μ over these types, for example, nominal types can be concretized by all constants of the correct type from the set of constants or the singleton types are concretized to the singleton value itself. The semantics of the type domains are defined in terms of the `PyType` wrapper that calls into the relevant RDL methods. The example implementation of these domains in § 4.2 is a simplified version of these domains.

In practice, the AUTOPANDAS benchmarks have input/output examples that are not just data frames, but also integers, Python lambdas, and method references (such as `nunique` from the Pandas library). ABSYNTHE is soundly able to abstract these into the relevant domains. For types, integers become `Integer` and lambdas are inferred as a type `Lambda`. When these values are lifted to the columns domain, they

are represented as $\perp$ as these are not data frames, thus there is no way to soundly represent their column labels. Additionally, ABSYNTHE infers a set of constants from the input/output examples as well. It adds any string or numeric row and column labels of the data frames, in addition to any string or numeric standalone values passed as arguments. This set is used to synthesize the constants during the application of the S-VAL rules.

Results Table 4.2 shows the results of running the AUTOPANDAS benchmarks through ABSYNTHE. The numbers are collected on a 2016 Macbook Pro with a 2.7GHz Intel Core i7 processor and 16 GB RAM, with a timeout of 20 minutes (consistent with the timeout of Bavishi et al. [13]). The *Name* column shows the name of the benchmark, i.e., the StackOverflow question ID from which the problem is taken. The *Depth* column shows the length of sequence of method call chain in the final solution. The AUTOPANDAS benchmarks are tuned to synthesize programs with a chain of method calls, where the bulk of the time spend is in synthesizing arguments to these method calls. This is characteristic of the Pandas API which accepts many arguments, often optional keyword arguments. The *Time* column shows the median of 11 runs along with the semi-interquartile range, where – denotes that a benchmark timed out. The *Size* lists the synthesized program size as number of AST nodes. Note that, this number is affected by both the depth of the synthesized program (the number of method calls) and the number of arguments to those methods. *# Tested* lists the number of concrete programs generated by ABSYNTHE that were tested in the Python interpreter. Finally, *AP Neural* and *AP Baseline* shares the benchmarks solved by the AUTOPANDAS neural model and naïve enumeration to aid in comparison with ABSYNTHE. Two benchmarks, *SO_12860421* and *SO_49567723*, are marked with a * as these were found in the AUTOPANDAS artifact were not reported in the paper.

Table 4.2: Results of running AutoPandas benchmarks through Absynthe. The *Depth* column shares the longest chain of method calls in the synthesized solution; *Time* lists the median and semi-interquartile range of 11 runs for time taken to synthesize a program; *Size* lists the number of AST nodes in the synthesized solution; *# Tested* reports the number of concrete Python programs tested; *AP Neural* and *AP Baseline* shares the benchmarks that AutoPandas neural model and naïve enumeration could synthesize. The benchmarks denoted with a * were a part of the artifact, but not reported in the paper [13]. Benchmarks highlighted in blue and yellow shows the benchmarks only synthesized by Absynthe and AutoPandas respectively.

Name	Depth	Time (sec)	Size	# Tested	AP Neural	AP Baseline
SO_11881165	1	0.20 ± 0.00	6	40	✓	✓
SO_11941492	1	13.84 ± 0.04	5	2507	✓	✓
SO_13647222	1	-			✓	✓
SO_18172851	1	0.42 ± 0.00	3	70		
SO_49583055	1	3.77 ± 0.01	6	272		
SO_49592930	1	0.22 ± 0.00	3	21	✓	✓
SO_49572546	1	1.50 ± 0.01	3	548	✓	✓
SO_12860421*	1	686.50 ± 1.68	11	1537521		
SO_13261175	1	283.12 ± 0.39	11	237755	✓	
SO_13793321	1	5.70 ± 0.04	6	413	✓	✓
SO_14085517	1	216.14 ± 0.38	7	12844	✓	✓
SO_11418192	2	0.10 ± 0.00	5	11	✓	✓
SO_49567723	2	-			✓	
SO_49987108*	2	-				
SO_13261691	2	65.17 ± 0.17	3	22322	✓	✓
SO_13659881	2	0.21 ± 0.00	6	45	✓	✓
SO_13807758	2	54.92 ± 0.26	6	3144	✓	✓
SO_34365578	2	-				
SO_10982266	3	-				
SO_11811392	3	6.88 ± 0.03	4	921		
SO_49581206	3	-				
SO_12065885	3	0.24 ± 0.00	6	286	✓	✓
SO_13576164	3	-			✓	
SO_14023037	3	-				
SO_53762029	3	545.62 ± 0.91	9	229233	✓	✓
SO_21982987	3	-			✓	✓
SO_39656670	3	-				
SO_23321300	3	-				

ABSYNTHE solves 17 programs, the same number of programs as AUTOPANDAS neural model. However, the set of synthesized programs by both tools are different with a significant overlap. Benchmarks listed in Table 4.2 without any highlight shows the benchmarks that were synthesized by both tools. Benchmarks highlighted in blue were synthesized only by ABSYNTHE but not by AUTOPANDAS. Likewise, benchmarks highlighted in yellow are the benchmarks synthesized only by AUTOPANDAS but not by ABSYNTHE. The time taken to synthesize the programs is largely dictated by how the abstract semantics prunes the space of programs, hence it is proportional to the number of concrete programs generated and tested. The fact that, for the same program size, the number of AST nodes in the method arguments (the difference between size and depth) is indicative of solving time shows that synthesizing arguments is indeed the bottleneck of this benchmark suite. For example, like *SO_11811392* and *SO_12065885* the type system quickly narrows down the search space, and the solution uses API methods that have 0 or 1 arguments only, making the arguments synthesis quick.

Discussion ABSYNTHE solves a harder synthesis problem because it does not use the list of methods to be used as provided in the specification. Instead, ABSYNTHE uses the complete set of 30 supported Pandas API for every benchmark. Approximately, this gives us a choice of permutations of size 1, 2, or 3 (depending on the depth of the final solution) from 30 methods, without considering arguments from those methods. In contrast, the baseline enumerative search *AP Baseline* comparison limits the search to only the Pandas API methods that will be used in the final solution. Typically this limits the search space to 1, 2, or 3 methods as given in the specification. In other words, under naïve enumeration, ABSYNTHE explores a strictly larger set of programs than AUTOPANDAS baseline.

In the benchmarks where ABSYNTHE failed to synthesize a solution, it falls back

to term enumeration as the abstract domain was not precise. More specifically in the benchmarks with depth 3, ABSYNTHE could do better by jointly reasoning about values in relational abstractions between multiple arguments of the same method. We plan to explore support for relational abstractions in future work. The neural model trained by Bavishi et al. [13] is good at guessing the sequences that are potentially likely to solve the synthesis task. It, however, does not take into account semantics of the program, thus eliminating impossible programs from being considered. This shows up in *SO_18172851* and *SO_49583055* where both enumerative search and neural models failed, but ABSYNTHE succeeds. Moreover, any updates to the neural model would need to be addressed with a new encoding or a retraining of the model on new data, a potentially resource consuming process. However, exploring the synergy of guidance from abstract interpretation combined with neural models similar to Anderson et al. [6] to rank *sound* program candidate choices is an interesting future work.

4.6 Related Work

General Purpose Synthesis Tools SEMGUS [58] has the same motivation as ABSYNTHE to develop a general-purpose abstraction guided synthesis framework. However, SEMGUS requires the programmer to provide semantics in a relational format as constrained horn clauses (CHCs). While CHCs are expressive and have dedicated solvers [60], correctly defining semantics as a relations is prohibitively time-consuming and error-prone. Moreover, SEMGUS performs well in proving unrealizability of synthesis problems, but it has limited success in synthesizing solutions. In contrast, ABSYNTHE is a dedicated synthesizer that is geared towards synthesizing programs based on executable abstract semantics. ABSYNTHE can be thought of as an unrealizability prover if coarse-grained semantics, the focus of ABSYNTHE, is

sufficient to prove unrealizable. SemGuS also supports under-approximate semantics, which is an interesting future work in the context of Absynthe. Rosette [102] and Sketch [97] are solver-aided languages that use bounded verification using a SMT solver to synthesize programs written in a DSL. In contrast, Absynthe relies on abstract interpretation to guide search, so it can reason about unbounded program properties. There has been parallel work in synthesis using Christiansen grammars [75] that allows one to encode some program semantics as context-dependent properties directly in the syntax grammar. However, an abstract interpreter-based approach gives Absynthe more semantic reasoning capabilities (like polymorphism).

Domain-specific synthesis SyGuS [4] being a standard synthesis problem specification format, has seen a variety of solver approaches. CVC4 [89] is a general-purpose SMT solver that has support for synthesizing programs in the SyGuS format. CVC4 has complete support for theory of strings and linear integer arithmetic, so it performs better than Absynthe (which is guided by simple abstract domains) for SyGuS. However, Absynthe's strength is generalizability to other kinds of synthesis problems as demonstrated in synthesis of AutoPandas benchmarks (§ 4.5.2). DryadSynth [52] explores a reconciling deductive and enumerative synthesis in SyGuS problems limited to the conditional linear integer arithmetic background theory. Some of their findings has been adopted by Absynthe (§ 4.4). EuSolver [3] is an enumerative solver that takes a divide-and-conquer approach. It synthesizes individual programs that are correct on a subset of examples, and predicates that distinguishes the program and combines these into a single final solution. Absynthe is close to EuSolver, as it is also based on enumerative search, but it is also guided by abstract semantics as well. We plan to support synthesizing conditionals in future work.

Past work solves synthesis problems using domain specific abstractions such as types and examples [76, 37], over-approximate semantics on table operations [32],

refinement types [80], secure declassification [50], abstract domain to verify atomic sections of a program [107], and SQL equivalence relations [108]. These abstraction can be designed as a domain and an abstract evaluation semantics can be provided to ABSYNTHE for synthesizing such programs. However, ABSYNTHE being a general purpose synthesis tool, will not have domain specific optimizations. We plan to explore ABSYNTHE as platform deploying domain specific synthesis in future work.

Abstraction-guided Synthesis SIMPL [96] combines enumerative search with static analysis based pruning, which is similar to ABSYNTHE. However, the program search in ABSYNTHE can be parameterized by a user provided abstract interpreter allowing the user to write specifications and semantics in a domain fit for the task-at-hand. Additionally, ABSYNTHE can infer abstract values for the holes in partial programs, thus guiding the search using the abstract semantics (Figure 4.6). BLAZE [110] is very similar to ABSYNTHE as it uses abstract semantics to guide the search. It adapts *counterexample guided abstraction refinement* to synthesis problems by refining the abstraction when a test fails, and constructing a proof of incorrectness in the process. However, it starts with a universe of predicates that is used for abstraction refinement, which is a requirement ABSYNTHE doesn't place on users. FlashMeta [83] is similar, but requires the definition of "inverse" semantics for operators using *witness functions*. ABSYNTHE, however, requires only the definition of forward abstract semantics and attempts to derive the inverse semantics automatically where possible.

Learning-based approaches There has been a recent rise of learning based approaches to make program synthesis more tractable. AUTOPANDAS [13] is an example of applying neural models to rank candidate choices constructed by other program generation methods (*smart operators* in AUTOPANDAS' case). DEEPCODER [11] trains a deep neural network to predict properties of programs based on input/output examples. These properties are used to augment the search by an enumerative search

or SMT solvers. ABSYNTHE is complementary to these approaches and does not use machine learning. In future, we plan to explore extensions to ABSYNTHE that reorders the program search order using a model learned on program text *and* abstract semantics. EUPHONY [63], on the other hand, uses an approach inspired by transfer learning to learn a *probabilistic higher order grammar*, and uses that in enumerative search to synthesize solutions. PROBE [12] learn a probabilistic grammar *just-in-time* during synthesis. Their key insight is that many SyGuS programs that pass a few examples have parts of the syntax that has higher likelihood to be present in the final solution. In contrast, ABSYNTHE is complementary to the approach of learning probabilistic grammars; abstract domains can prune the space of programs, while the grammar can assign higher weights to the terms that should be enumerated earlier. We leave exploring the synergy between these approaches to future work.

4.7 Conclusion

We presented ABSYNTHE, a tool that combines abstract interpretation and testing to synthesize programs. It accepts user-defined lightweight abstract domains and partial semantics for the language as an input, and enables guided search over the space of programs in the language. We evaluated ABSYNTHE on SyGuS strings benchmarks and found ABSYNTHE can solve 77% of the benchmarks, most within 7 seconds. Moreover, ABSYNTHE supports a pay-as-you-go model, where the user only pays for the abstract domain they are using for synthesis. Finally, to evaluate the generality of ABSYNTHE to other domains, we use it to synthesize Pandas data frame manipulation programs in Python from the AUTOPANDAS benchmark suite. ABSYNTHE performs at par with AUTOPANDAS and synthesizes programs with low specification burden, but no neural network training costs. We believe ABSYNTHE demonstrates a promising design choice for design of synthesis tools that leverage testing for correctness along

with lightweight abstractions with partial semantics for search guidance.

Chapter 5

Conclusion and Future Work

This dissertation describes a synthesis framework that is both general and lightweight. The two key ideas underpinning the framework are: first, an abstraction-guided search component that can be inferred directly from the abstract domain and interpreter definitions. Second, a testing component that checks for correctness and additionally infers hints in the form of effects to refine the program candidates towards likely solutions. We demonstrate how simple types and effect labels inferred from failing tests can synthesize side-effect causing programs efficiently in RbSyn (Chapter 2). In contrast, Anosy (Chapter 3) demonstrates the use of a precise abstraction–refinement types to synthesize privacy preserving wrappers for queries on secret data. The use of refinement types allows Anosy to generate verified bounds that are crucial for security, without writing test cases. Finally, Absynthe (Chapter 4) abstracts over these tools to develop a search algorithm that is guided by any user-provided abstract domains and its semantics. This represents an important step forward in the design of practical synthesis tools that provide lightweight formal guarantees by developing static program analyses while ensuring correctness from tests. Together, these contributions show that techniques from abstract interpretation can be combined with testing to empower programmers with practical synthesis tools that generate programs in their languages

or libraries of choice along with correctness guarantees.

5.1 Future Work

This dissertation lays the groundwork for a some interesting avenues of future work:

Abstraction guided synthesis While ABSYNTHE shows that any abstract domain can be used to guide the search to synthesize the programs, there are future research questions that will allow synthesizers to be more efficient. First, can we use ABSYNTHE as a platform to deploy challenging abstract domains like probabilistic domains or borrowing/ownership models? This questions boils down to can we have algorithms to efficiently infer abstract value tags for each hole when generating terms. If ABSYNTHE scales to such expressive domains, we can synthesize programs that satisfy probabilistic properties like fairness, quantifying information flow over distributions, or programs that are resource safe, i.e., do not allocate and free memory, file and network handles correctly. Second, can ABSYNTHE do joint reasoning over all holes in a term? For example, consider a function that that has two arguments. Currently, ABSYNTHE has to concretize one abstract value to infer the abstraction for the other hole. In contrast, solver-aided tools like Synquid [80] can jointly reason over all holes in a term and infer weakest preconditions as abstractions for such subgoals. It would be worth exploring if algorithms from Synquid, that uses unsat cores can be used in the setting ABSYNTHE to doing joint inference of abstract values. Third, a synthesizer guided by an abstract interpreter, like ABSYNTHE, is complete with respect to the programs that can be automatically checked by the abstract interpreter. This limitation implies there may always be sounds programs that will never be considered because the abstract interpreter rejected it due to imprecision. A common example of such imprecision occurs when a type-checker rejects a valid program, and requires the programmer to provide a type cast to type-check. However, in our setting it would be worth

investigating the points at which a synthesizer lost precision and design a human-feedback loop (akin to a type cast) to keep going despite the imprecision. Moreover, in our setting the synthesis algorithm has access to concrete test cases which can provide more information that just using the abstract specification.

Inference from testing RbSyn infers effect annotations for a candidate programs and refines the program to match the inferred effect. While coarse-grained effects were effective for the domain of database accessing web apps, other abstractions might be a better fit for other domains. In general, the research question is can potential reasons for test failure be automatically inferred as an abstract specification and used to guide the search. As this inference is guided from test failures these are under-approximate. In contrast, the abstract interpreters and specifications using abstract domains provided by the user to the synthesizer as over-approximate. A key question is can we use principles of incorrectness logic [72] to prune the search space? I hypothesize just like over-approximations eliminate unsound programs, under-approximations can give hints to the likely program but not eliminate a family of terms. Such hints may be used to prioritize the order in which a synthesizer searches through the program space, making it likely to reach a refined program from test failure earlier in the search. I plan to explore this in future work.

Learned models Deep learning models for code generation has moved into regular use in recent years. For example, systems like GitHub Copilot [38] use natural language as specifications along with existing code in the editor buffer to suggest code completions. However, systems like Copilot provide no guarantees of correctness, requiring a manual audit of the generated code and often barring it's use in safety critical domains. There are opportunities in combining machine learning models with reasoning based approaches explored in this dissertation.

This dissertation utilized two key heuristics to explore program in the search

space. First, the size of candidate programs, i.e., smaller programs are explored first. Second, the number of times a candidate has added an effect hole in RbSyn, which is a proxy for program refinement using guidance from test failure. Deep learning can be used to learn heuristics to explore likely programs based on how the partial candidates programs from intermediate search steps are likely to satisfy the provided specifications. Another potential future work is to leverage large-language models like GPT-4 [74] to generate programs in one-shot, rather than gradually generating programs in increasing order of size. As these generated programs will not be correct, this is an opportunity to design effect error localization techniques to detect where the candidate is not aligning with the specification and apply program repair techniques.

Appendix A

RbSyn: Complete Evaluation and Synthesis Rules

A.1 Evaluation Rules

We extend λ_{syn} to include errors $\mathcal{E}$. Errors can originate from the evaluation of an assertion `assert` e and encapsulates the read effect ϵ_r and write effect ϵ_w inferred from e. Results $\mathcal{R}$ of an evaluation can either be a value or an error. Collecting effects while evaluating the postcondition of tests require special evaluation rules. Figure A.2 shows selected rules of the small step operational semantics for only postconditions (rest omitted as they are standard rules). The rules prove judgments of the form $[\![E, c, \langle \epsilon_r, \epsilon_w \rangle, Q]\!] \hookrightarrow_{CT} [\![E', c', \langle \epsilon'_r, \epsilon'_w \rangle, Q']\!]$ that reduce configurations that contain a dynamic environment E, counter of passed assertions c, the pair of read and write effects $\langle \epsilon_r, \epsilon_w \rangle$ collected during evaluation, and postcondition Q under evaluation. Rule

$$\begin{array}{llll} \textit{Errors} & \mathcal{E} & ::= & \texttt{err}(\epsilon_r, \epsilon_w) \\ \textit{Results} & \mathcal{R} & ::= & v \mid \mathcal{E} \end{array}$$

Figure A.1: Extended λ_{syn}.

$$\boxed{[\![E, c, \langle \epsilon_r, \epsilon_w \rangle, Q]\!] \hookrightarrow_{CT} [\![E', c', \langle \epsilon'_r, \epsilon'_w \rangle, Q']\!]}$$

$$\frac{v \in \{\texttt{true}, [A]\}}{[\![E, c, \langle \epsilon_r, \epsilon_w \rangle, \texttt{assert } v]\!] \hookrightarrow_{CT} [\![E, c+1, \langle \epsilon_r, \epsilon_w \rangle, v]\!]} \text{ E-AssertPass}$$

$$\frac{v \in \{\texttt{false}, \texttt{nil}\}}{[\![E, c, \langle \epsilon_r, \epsilon_w \rangle, \texttt{assert } v]\!] \hookrightarrow_{CT} [\![E, c, \langle \epsilon_r, \epsilon_w \rangle, \texttt{err}(\epsilon_r, \epsilon_w)]\!]} \text{ E-AssertFail}$$

$$\frac{[\![E, c, \langle \epsilon_r, \epsilon_w \rangle, e]\!] \hookrightarrow_{CT} [\![E', c, \langle \epsilon'_r, \epsilon'_w \rangle, e']\!]}{[\![E, c, \langle \epsilon_r, \epsilon_w \rangle, \texttt{assert } e]\!] \hookrightarrow_{CT} [\![E', c, \langle \epsilon'_r, \epsilon'_w \rangle, \texttt{assert } e']\!]} \text{ E-AssertStep}$$

$$\frac{\begin{array}{ccc}\text{type\_of}(v_r, v_a) = (A_r, A_a) & m : \tau_a \xrightarrow{\langle \epsilon_r, \epsilon_w \rangle} \tau \in CT(A) \\ A_r \leq A \qquad A_a \leq \tau_a & \text{call}(A.m, v_r, v_a) = v\end{array}}{[\![E, c, \langle \epsilon'_r, \epsilon'_w \rangle, v_r.m(v_a)]\!] \hookrightarrow_{CT} [\![E, c, \langle \epsilon'_r, \epsilon'_w \rangle \cup \langle \epsilon_r, \epsilon_w \rangle, v]\!]} \text{ E-MethCall}$$

$$\frac{[\![E, c, \langle \epsilon_r, \epsilon_w \rangle, Q_1]\!] \hookrightarrow_{CT} [\![E, c, \langle \epsilon'_r, \epsilon'_w \rangle, Q'_1]\!]}{[\![E, c, \langle \epsilon_r, \epsilon_w \rangle, Q_1; Q_2]\!] \hookrightarrow_{CT} [\![E, c, \langle \epsilon'_r, \epsilon'_w \rangle, Q'_1; Q_2]\!]} \text{ E-SeqStep}$$

$$\frac{}{[\![E, c, \langle \epsilon_r, \epsilon_w \rangle, v; Q_2]\!] \hookrightarrow_{CT} [\![E, c, \langle \bullet, \bullet \rangle, Q_2]\!]} \text{ E-SeqVal}$$

$$\frac{}{[\![E, c, \langle \epsilon_r, \epsilon_w \rangle, \texttt{err}(\epsilon_r, \epsilon_w); Q_2]\!] \hookrightarrow_{CT} [\![E, c, \langle \epsilon_r, \epsilon_w \rangle, \texttt{err}(\epsilon_r, \epsilon_w)]\!]} \text{ E-SeqErr}$$

Figure A.2: Selected rules for operational semantics of the postcondition Q

E-AssertPass applies when assertion evaluates to a truthy-y value. It also increments the counter for passed assertions. If e evaluates to a false-y value, it results in an error, in which case it returns the collected side effects with the error (E-AssertFail). Evaluation of a library method, gives a union of its effects with the already collected effects (E-MethCall). During the evaluation of a sequence $Q; Q$, if the first assertion evaluates to a value, the evaluation continues discarding all collected effects (E-SeqVal). If the evaluation of an assert yields an error, the evaluation of postcondition terminates with the error as final result (E-SeqErr). We define the big step semantics as follows: $Q \Downarrow \mathcal{R}$ if $\exists E', c, \langle \epsilon_r, \epsilon_w \rangle . [\![[x_r \to v], 0, \langle \bullet, \bullet \rangle, Q]\!] \hookrightarrow^*_{CT} [\![E', c, \langle \epsilon_r, \epsilon_w \rangle, \mathcal{R}]\!]$, in other words, evaluating the postcondition in an environment containing the return value of synthesis goal x_r, will evaluate to a result.

A.2 Type-Guided Synthesis

Figure A.3 shows all the type checking and type-directed synthesis rules. The repeated rules are same as § 2.3. T-Nil type checks the value `nil` and assigns it the type *Nil*. The rules T-True, T-False, T-Obj and T-Err do the same for the values `true`, `false`, $[A]$ and $\texttt{err}(\epsilon_r, \epsilon_w)$ respectively. Similarly T-NegB and T-OrB give the rules to type check conditionals that contain negation or disjunction. T-EffHole typechecks an effect hole with the *Obj* type. It can be narrowed to a more precise type when synthesis rules are applied. T-Seq does type checking and synthesis for sequences and T-App type checks or synthesizes terms in the receiver and arguments of the method call. T-If type checks if-else expressions.

A.3 Algorithm

Algorithm 4 describes the synthesis of candidates that pass a single spec. The algorithm uses a work list, which initially contains a tuple with the number of passed assertions

$$\boxed{\Sigma, \Gamma \vdash_{CT} e \rightsquigarrow e : \tau}$$

$$\frac{}{\Sigma, \Gamma \vdash_{CT} \texttt{nil} \rightsquigarrow \texttt{nil} : \mathit{Nil}}\ \text{T-Nil} \qquad \frac{}{\Sigma, \Gamma \vdash_{CT} \texttt{true} \rightsquigarrow \texttt{true} : \mathit{Bool}}\ \text{T-True}$$

$$\frac{}{\Sigma, \Gamma \vdash_{CT} \texttt{false} \rightsquigarrow \texttt{false} : \mathit{Bool}}\ \text{T-False} \qquad \frac{}{\Sigma, \Gamma \vdash_{CT} [A] \rightsquigarrow [A] : A}\ \text{T-Obj}$$

$$\frac{}{\Sigma, \Gamma \vdash_{CT} \texttt{err}(\epsilon_r, \epsilon_w) \rightsquigarrow \texttt{err}(\epsilon_r, \epsilon_w) : \mathit{Err}}\ \text{T-Err}$$

$$\frac{\Gamma(x) = \tau}{\Sigma, \Gamma \vdash_{CT} x \rightsquigarrow x : \tau}\ \text{T-Var} \qquad \frac{\Sigma, \Gamma \vdash_{CT} b \rightsquigarrow b' : \mathit{Bool}}{\Sigma, \Gamma \vdash_{CT} !b \rightsquigarrow !b' : \mathit{Bool}}\ \text{T-NegB}$$

$$\frac{\begin{array}{c}\Sigma, \Gamma \vdash_{CT} b_1 \rightsquigarrow b_1' : \mathit{Bool} \\ \Sigma, \Gamma \vdash_{CT} b_2 \rightsquigarrow b_2' : \mathit{Bool}\end{array}}{\Sigma, \Gamma \vdash_{CT} b_1 \vee b_2 \rightsquigarrow b_1' \vee b_2' : \mathit{Bool}}\ \text{T-OrB}$$

$$\frac{}{\Sigma, \Gamma \vdash_{CT} \square : \tau \rightsquigarrow \square : \tau}\ \text{T-Hole} \qquad \frac{}{\Sigma, \Gamma \vdash_{CT} \lozenge : \epsilon \rightsquigarrow (\lozenge : \epsilon) : \mathit{Obj}}\ \text{T-EffHole}$$

$$\frac{v : \tau_1 \in \Sigma \qquad \tau_1 \leq \tau_2}{\Sigma, \Gamma \vdash_{CT} \square : \tau_2 \rightsquigarrow v : \tau_1}\ \text{S-Const} \qquad \frac{\Gamma(x) = \tau_1 \qquad \tau_1 \leq \tau_2}{\Sigma, \Gamma \vdash_{CT} \square : \tau_2 \rightsquigarrow x : \tau_1}\ \text{S-Var}$$

$$\frac{m : \tau_1 \to \tau_2 \in CT(A) \qquad \tau_2 \leq \tau_3}{\Sigma, \Gamma \vdash_{CT} \square : \tau_3 \rightsquigarrow (\square : A).m(\square : \tau_1) : \tau_2}\ \text{S-App}$$

$$\frac{\Sigma, \Gamma \vdash_{CT} e_1 \rightsquigarrow e_1' : \tau_1 \qquad \Sigma, \Gamma \vdash_{CT} e_2 \rightsquigarrow e_2' : \tau_2}{\Sigma, \Gamma \vdash_{CT} e_1; e_2 \rightsquigarrow e_1'; e_2' : \tau_2}\ \text{T-Seq}$$

$$\frac{\begin{array}{c}\Sigma, \Gamma \vdash_{CT} e_1 \rightsquigarrow e_1' : \tau \qquad \Sigma, \Gamma \vdash_{CT} e_2 \rightsquigarrow e_2' : \tau_3 \\ m : \tau_1 \to \tau_2 \in CT(A) \qquad \tau \leq A \qquad \tau_3 \leq \tau_1\end{array}}{\Sigma, \Gamma \vdash_{CT} e_1.m(e_2) \rightsquigarrow e_1'.m(e_2') : \tau_2}\ \text{T-App}$$

$$\frac{\begin{array}{c}\Sigma, \Gamma \vdash_{CT} e_1 \rightsquigarrow e_1' : \tau_1 \\ \Sigma, \Gamma[x \mapsto \tau_1] \vdash_{CT} e_2 \rightsquigarrow e_2' : \tau_2\end{array}}{\Sigma, \Gamma \vdash_{CT} \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \rightsquigarrow \texttt{let}\ x = e_1'\ \texttt{in}\ e_2' : \tau_2}\ \text{T-Let}$$

$$\frac{\begin{array}{c}\Sigma, \Gamma \vdash_{CT} b \rightsquigarrow b' : \mathit{Bool} \\ \Sigma, \Gamma \vdash_{CT} e_1 \rightsquigarrow e_1' : \tau_1 \qquad \Sigma, \Gamma \vdash_{CT} e_2 \rightsquigarrow e_2' : \tau_2\end{array}}{\Sigma, \Gamma \vdash_{CT} \texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightsquigarrow \texttt{if}\ b'\ \texttt{then}\ e_1'\ \texttt{else}\ e_2' : \tau_1 \cup \tau_2}\ \text{T-If}$$

Figure A.3: Type checking and type-directed synthesis rules

Algorithm 4 Synthesis of programs that passes a spec s

procedure GENERATE($\tau_1 \to \tau_2$, CT, Σ, s, maxSize)
  $\Gamma \leftarrow [x \mapsto \tau_1]$
  $e_0 \leftarrow \square : \tau_2$
  workList $\leftarrow [(0, e_0)]$
  while workList is not empty **do**
    $(c, e_b) \leftarrow$ pop(workList)
    $\omega_\tau \leftarrow \{(c, e_t) \mid \Sigma, \Gamma \vdash e_b \rightsquigarrow e_t : \tau\}$
    $\omega_{eval} \leftarrow \{(c, e_t) \in \omega_\tau \wedge \texttt{evaluable}(e_t)\}$
    $\omega_r \leftarrow \omega_\tau - \omega_{eval}$
    for all $(c, e_t) \in \omega_{eval}$ **do**
      $c_r, v_r \leftarrow$ EVALPROGRAM(e_t, s)
      if $v_r = \texttt{err}(\epsilon_r, \epsilon_w)$ **then**
        $\omega_r \leftarrow \omega_r \cup \{(c, e_f) \mid \Sigma, \Gamma, \epsilon_r \vdash e_t \twoheadrightarrow e_f\}$
      else
        return e_t
      end if
    end for
    $\omega_r \leftarrow \{(c, e_b) \in \omega_r \wedge \texttt{size}(e_b) \leq$ maxSize$\}$
    workList $\leftarrow$ reorder(workList + ω_r)
  end while
  return Error: No solution found
end procedure

procedure EVALPROGRAM(e, $\langle S, Q \rangle$)
  $P \leftarrow \texttt{def}\ m(x) = e$, $E \leftarrow []$
  $(c, \mathcal{R}) \leftarrow [\![E, 0, \langle \bullet, \bullet \rangle, P; S; Q]\!] \hookrightarrow^*_{CT} [\![E', c, \langle \epsilon_r, \epsilon_w \rangle, \mathcal{R}]\!]$
  return $(c, \mathcal{R})$
end procedure

$$
\begin{array}{lll}
\texttt{evaluable}\ \ e_1; e_2 & = & \texttt{evaluable}\ e_1 \wedge \texttt{evaluable}\ e_2 \\
\texttt{evaluable}\ \ e_1.m(e_2) & = & \texttt{evaluable}\ e_1 \wedge \texttt{evaluable}\ e_2 \\
\texttt{evaluable}\ \ \square : \tau & = & \texttt{false} \\
\texttt{evaluable}\ \ \texttt{let}\ x = e_1\ \texttt{in}\ e_2 & = & \texttt{evaluable}\ e_1 \wedge \texttt{evaluable}\ e_2 \\
\texttt{evaluable}\ \ \texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & = & \texttt{evaluable}\ b \wedge \texttt{evaluable}\ e_1 \wedge \texttt{evaluable}\ e_2 \\
\texttt{evaluable}\ \ !b & = & \texttt{evaluable}\ b \\
\texttt{evaluable}\ \ b_1 \vee b_2 & = & \texttt{evaluable}\ b_1 \wedge \texttt{evaluable}\ b_2 \\
\texttt{evaluable}\ \ \_ & = & \texttt{true}
\end{array}
$$

$$
\begin{array}{lll}
\texttt{size}\ \ e_1; e_2 & = & \texttt{size}\ e_1 + \texttt{size}\ e_2 \\
\texttt{size}\ \ e_1.m(e_2) & = & \texttt{size}\ e_1 + \texttt{size}\ e_2 + 1 \\
\texttt{size}\ \ \texttt{let}\ x = e_1\ \texttt{in}\ e_2 & = & \texttt{size}\ e_1 + \texttt{size}\ e_2 \\
\texttt{size}\ \ \texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & = & \texttt{size}\ b + \texttt{size}\ e_1 + \texttt{size}\ e_2 \\
\texttt{size}\ \ !b & = & \texttt{size}\ b \\
\texttt{size}\ \ b_1 \vee b_2 & = & \texttt{size}\ b_1 + \texttt{size}\ b_2 \\
\texttt{size}\ \ \_ & = & 0
\end{array}
$$

Figure A.4: Helper functions used by RbSyn

starting at 0 initially and the initial hole of type τ_2. The first tuple is popped off the work list and applies type or effect guided synthesis rules, the $\rightsquigarrow$ relation, to a base expression e_b to build the set of new expressions ω_τ. Next, expressions with no holes are filtered into a list of *evaluable* expressions ω_{eval}. Then, EvalProgram is called to make a program $\texttt{def}\ m(x) = e_t$ and evaluate spec s in an environment E, starting from a passed assertion count of 0.

If the evaluation of the postcondition results in an error $\texttt{err}(\epsilon_r, \epsilon_w)$, the algorithm proceeds to introduce an effect hole, using the relation $\twoheadrightarrow$ to build the set ω_ϵ. If a program passes all the assertions then it means a correct solution has been found so, Generate returns it. Finally, the algorithm collects all the *remainder* expressions ω_r with holes and filters the programs that exceed the maximum permissible size *maxSize*. This bounds the search to particular search space size. It then takes the filtered programs and programs from the remainder of the work list and reorders them. The programs are sorted by the number of passed assertions c in the decreasing order

$$\langle e_1, b_1, \Psi_1\rangle \oplus \langle e_2, b_2, \Psi_2\rangle = \langle b_1, b_1 \vee b_2, \Psi_1 \cup \Psi_2\rangle \quad \text{if } e_1 \equiv \texttt{true}, e_2 \equiv \texttt{false} \text{ and } b_1 \equiv !b_2 \tag{A.1}$$

$$\langle e_1, b_1, \Psi_1\rangle \oplus \langle e_2, b_2, \Psi_2\rangle = \langle b_2, b_1 \vee b_2, \Psi_1 \cup \Psi_2\rangle \quad \text{if } e_1 \equiv \texttt{false}, e_2 \equiv \texttt{true} \text{ and } b_1 \equiv !b_2 \tag{A.2}$$

$$\langle e_1, b_1, \Psi_1\rangle \oplus \langle e_2, b_2, \Psi_2\rangle = \langle e_1, b_1, \Psi_1\rangle \oplus \langle e_2, b_g, \Psi_2\rangle \text{ if } b_g \equiv !b_1 \text{ and } \forall \langle S_i, Q_i\rangle \in \Psi_2.\texttt{def } m(x) = b_g \vdash S_i; \texttt{assert } x_r \Downarrow v \tag{A.3}$$

$$\langle e_1, b_1, \Psi_1\rangle \oplus \langle e_2, b_2, \Psi_2\rangle = \langle e_1, b_g, \Psi_1\rangle \oplus \langle e_2, b_2, \Psi_2\rangle \text{ if } b_g \equiv !b_2 \text{ and } \forall \langle S_i, Q_i\rangle \in \Psi_1.\texttt{def } m(x) = b_g \vdash S_i; \texttt{assert } !x_r \Downarrow v \tag{A.4}$$

Figure A.5: Branch pruning rules.

and then by the program size in the increasing order. This assumes that a program that is more likely correct and smaller will be selected earlier for processing in the work list. Lastly, if GENERATE doesn't find a correct program in that search space it will return an error for the same.

Figure A.4 shows the formal definitions of `hasHole`, and `size` functions.

A.4 Branch pruning rules

Figure A.5 formally describes the rules that allows RBSYN to do term rewriting in λ_{syn} for branch pruning. These are useful particularly for reducing boolean programs. Rules A.1 and A.2 allows us to rewrite expressions into their branch condition if the expression body is `true` or `false` reducing expressions like `if` b `then true else false` to b. Rules A.3 and A.4 guess a conditional that is the negation of the other, if the negation holds for the tests. Any $\oplus$ term where two branch conditions are negations of each other reflect a shorter program `if` b_1 `then` e_1 `else` e_2 in λ_{syn}. If it is a boolean program, then it might even enable application of rules A.1 and A.2 producing a single line program.

Bibliography

[1] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. "Recursive program synthesis". In: *International conference on computer aided verification*. Springer. 2013, pp. 934–950. DOI: 10.1007/978-3-642-39799-8_67.

[2] Rajeev Alur, Pavol Černỳ, and Arjun Radhakrishna. "Synthesis through unification". In: *International Conference on Computer Aided Verification*. Springer. 2015, pp. 163–179. DOI: 10.1007/978-3-319-21668-3_10.

[3] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. "Scaling Enumerative Program Synthesis via Divide and Conquer". In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*. Vol. 10205. Lecture Notes in Computer Science. 2017, pp. 319–336. DOI: 10.1007/978-3-662-54577-5_18.

[4] Rajeev Alur et al. "Syntax-guided synthesis". In: *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 2013, pp. 1–8. URL: https://ieeexplore.ieee.org/document/6679385/.

[5] Rajeev Alur et al. "SyGuS-Comp 2017: Results and Analysis". In: *Proceedings Sixth Workshop on Synthesis, SYNTCAV 2017, Heidelberg, Germany, 22nd July 2017*. Vol. 260. EPTCS. 2017, pp. 97–115. DOI: 10.4204/EPTCS.260.9.

[6] Greg Anderson et al. "Neurosymbolic Reinforcement Learning with Formally Verified Exploration". In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. Ed. by Hugo Larochelle et al. 2020. URL: https://proceedings.neurips.cc/paper/2020/hash/448d5eda79895153938a8431919f4c9f-Abstract.html.

[7] Owen Arden et al. "Sharing Mobile Code Securely with Information Flow Control". In: *IEEE Symposium on Security and Privacy, (S&P 2012), 21-23 May 2012, San Francisco, California, USA*. IEEE Computer Society, 2012, pp. 191–205. DOI: 10.1109/SP.2012.22.

[8] Aslan Askarov and Andrei Sabelfeld. "Gradual Release: Unifying Declassification, Encryption and Key Release Policies". In: *2007 IEEE Symposium on Security and Privacy (S&P 2007), 20-23 May 2007, Oakland, California, USA*. IEEE Computer Society, 2007, pp. 207–221. DOI: 10.1109/SP.2007.22.

[9] Michael Backes, Boris Köpf, and Andrey Rybalchenko. "Automatic Discovery and Quantification of Information Leaks". In: *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*. IEEE Computer Society, 2009, pp. 141–153. DOI: 10.1109/SP.2009.18.

[10] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. "Widening operators for powerset domains". In: *International Journal on Software Tools for Technology Transfer* 9.3-4 (2007), pp. 413–414. DOI: 10.1007/s10009-007-0029-y.

[11] Matej Balog et al. "DeepCoder: Learning to Write Programs". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL: https://openreview.net/forum?id=ByldLrqlx.

[12] Shraddha Barke, Hila Peleg, and Nadia Polikarpova. "Just-in-time learning for bottom-up enumerative synthesis". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 227:1–227:29. DOI: 10.1145/3428295.

[13] Rohan Bavishi et al. "AutoPandas: neural-backed generators for program synthesis". In: *Proc. ACM Program. Lang.* 3.OOPSLA (2019), 168:1–168:27. DOI: 10.1145/3360594.

[14] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. "νZ - An Optimizing SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Vol. 9035. Lecture Notes in Computer Science. Springer, 2015, pp. 194–199. DOI: 10.1007/978-3-662-46681-0_14.

[15] Robert L Bocchino Jr et al. "A type and effect system for deterministic parallel Java". In: *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. 2009, pp. 97–116. DOI: 10.1145/1640089.1640097.

[16] Alexander Borgida, John Mylopoulos, and Raymond Reiter. "On the frame problem in procedure specifications". In: *IEEE Transactions on Software Engineering* 21.10 (1995), pp. 785–798. DOI: 10.1109/32.469460.

[17] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. "Effects as capabilities: effect handlers and lightweight effect polymorphism". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 126:1–126:30. DOI: 10.1145/3428194.

[18] Niklas Broberg, Bart van Delft, and David Sands. "Paragon - Practical programming with information flow control". In: *Journal of Computer Security* 25.4-5 (2017), pp. 323–365. DOI: 10.3233/JCS-15791.

[19] José González Cabañas et al. "Unique on Facebook: formulation and evidence of (nano)targeting individual users with non-PII data". In: *IMC '21: ACM Internet Measurement Conference, Virtual Event, USA, November 2-4, 2021*. ACM, 2021, pp. 464–479. DOI: 10.1145/3487552.3487861.

[20] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. "Optimizing database-backed applications with query synthesis". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. Vol. 48. 6. ACM New York, NY, USA, 2013, pp. 3–14. DOI: 10.1145/2499370.2462180.

[21] Stephen Chong and Andrew C Myers. "Security policies for downgrading". In: *Proceedings of the 11th ACM conference on Computer and communications security*. 2004, pp. 198–209. DOI: 10.1145/1030083.1030110.

[22] David Clark, Sebastian Hunt, and Pasquale Malacaria. "Quantitative Information Flow, Relations and Polymorphic Types". In: *Journal of Logic and Computation* 15.2 (2005), pp. 181–199. DOI: 10.1093/logcom/exi009.

[23] Adele Cooper. *Facebook Ads: A Guide to Targeting and Reporting*. https://web.archive.org/web/20110521050104/http://www.openforum.com/articles/facebook-ads-a-guide-to-targeting-and-reporting-adele-cooper. 2011.

[24] Patrick Cousot and Radhia Cousot. "Static determination of dynamic properties of programs". In: *Proceedings of the 2nd International Symposium on Programming, Paris, France*. Dunod. 1976.

[25] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[26] Patrick Cousot et al. "The ASTREÉ Analyzer". In: *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*. Vol. 3444. Lecture Notes in Computer Science. Springer, 2005, pp. 21–30. DOI: 10.1007/978-3-540-31987-0_3.

[27] Aaron Craig et al. "Capabilities: Effects for Free". In: *Formal Methods and Software Engineering - 20th International Conference on Formal Engineering Methods, ICFEM 2018, Gold Coast, QLD, Australia, November 12-16, 2018, Proceedings*. Ed. by Jing Sun and Meng Sun. Vol. 11232. Lecture Notes in Computer Science. Springer, 2018, pp. 231–247. DOI: 10.1007/978-3-030-02450-5_14.

[28] Bart van Delft, Sebastian Hunt, and David Sands. "Very static enforcement of dynamic policies". In: *Principles of Security and Trust - 4th International Conference, POST 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*. Springer. 2015, pp. 32–52. DOI: 10.1007/978-3-662-46666-7_3.

[29] Dominique Devriese and Frank Piessens. "Information flow enforcement in monadic libraries". In: *Proceedings of TLDI 2011: 2011 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Austin, TX, USA, January 25, 2011*. ACM, 2011, pp. 59–72. DOI: 10.1145/1929553.1929564.

[30] Diaspora Inc. *Diaspora: A privacy-aware, distributed, open source social network*. https://github.com/diaspora/diaspora. 2020.

[31] Yu Feng et al. "Component-based synthesis for complex APIs". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 2017, pp. 599–612. DOI: 10.1145/3093333.3009851.

[32] Yu Feng et al. "Component-based synthesis of table consolidation and transformation tasks from examples". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. ACM, 2017, pp. 422–436. DOI: 10.1145/3062341.3062351.

[33] Yu Feng et al. "Program synthesis using conflict-driven learning". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. Vol. 53. 4. ACM New York, NY, USA, 2018, pp. 420–435. DOI: 10.1145/3192366.3192382.

[34] John K Feser, Swarat Chaudhuri, and Isil Dillig. "Synthesizing data structure transformations from input-output examples". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* 50.6 (2015), pp. 229–239. DOI: 10.1145/2737924.2737977.

[35] Jean-Christophe Filliâtre and Andrei Paskevich. "Why3—where programs meet provers". In: *European symposium on programming*. Springer. 2013, pp. 125–128. DOI: 10.1007/978-3-642-37036-6_8.

[36] Jeffrey Foster et al. *RDL: Types, type checking, and contracts for Ruby*. https://github.com/tupl-tufts/rdl. 2020.

[37] Jonathan Frankle et al. "Example-directed synthesis: a type-theoretic interpretation". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* 51.1 (2016), pp. 802–815. DOI: 10.1145/2837614.2837629.

[38] Nat Friedman. *Introducing GitHub Copilot: your AI pair programmer*. URL: https://github.blog/2021-06-29-introducing-github-copilot-ai-pair-programmer/.

[39] Joel Galenson et al. "Codehint: Dynamic and interactive synthesis of code snippets". In: *Proceedings of the 36th International Conference on Software Engineering*. 2014, pp. 653–663. DOI: 10.1145/2568225.2568250.

[40] GitLab B.V. *GitLab is an open source end-to-end software development platform with built-in version control, issue tracking, code review, CI/CD, and more*. https://gitlab.com/gitlab-org/gitlab. 2020.

[41] Joseph A. Goguen and José Meseguer. "Security Policies and Security Models". In: *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*. IEEE Computer Society, 1982, pp. 11–20. DOI: 10.1109/SP.1982.10014.

[42] Colin S. Gordon. "Designing with Static Capabilities and Effects: Use, Mention, and Invariants (Pearl)". In: *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*. Ed. by Robert Hirschfeld and Tobias Pape. Vol. 166. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 10:1–10:25. DOI: 10.4230/LIPIcs.ECOOP.2020.10.

[43] Marco Guarnieri, Srdjan Marinovic, and David A. Basin. "Securing Databases from Probabilistic Inference". In: *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*. IEEE Computer Society, 2017, pp. 343–359. DOI: 10.1109/CSF.2017.30.

[44] Marco Guarnieri et al. "Information-Flow Control for Database-Backed Applications". In: *IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, June 17-19, 2019*. IEEE, 2019, pp. 79–94. DOI: 10.1109/EuroSP.2019.00016.

[45] Sumit Gulwani. "Automating string processing in spreadsheets using input-output examples". In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. Ed. by Thomas Ball and Mooly Sagiv. ACM, 2011, pp. 317–330. DOI: 10.1145/1926385.1926423.

[46] Sumit Gulwani. "Automating string processing in spreadsheets using input-output examples". In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* 46.1 (2011), pp. 317–330. DOI: 10.1145/1925844.1926423.

[47] Sumit Gulwani, William R Harris, and Rishabh Singh. "Spreadsheet data manipulation using examples". In: *Communications of the ACM* 55.8 (2012), pp. 97–105. DOI: 10.1145/2240236.2240260.

[48] Sankha Narayan Guria, Jeffrey S. Foster, and David Van Horn. *Absynthe: Abstract Interpretation-Guided Synthesis*. 2023. arXiv: 2302.13145 [cs.PL].

[49] Sankha Narayan Guria, Jeffrey S. Foster, and David Van Horn. "RbSyn: Type- and Effect-Guided Program Synthesis". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 344–358. ISBN: 9781450383912. DOI: 10.1145/3453483.3454048.

[50] Sankha Narayan Guria et al. “ANOSY: approximated knowledge synthesis with refinement types for declassification”. In: *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*. Ed. by Ranjit Jhala and Isil Dillig. ACM, 2022, pp. 15–30. DOI: 10.1145/3519939.3523725.

[51] William R Harris and Sumit Gulwani. “Spreadsheet table transformations from examples”. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* 46.6 (2011), pp. 317–328. DOI: 10.1145/1993316.1993536.

[52] Kangjing Huang et al. “Reconciling enumerative and deductive program synthesis”. In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. Ed. by Alastair F. Donaldson and Emina Torlak. ACM, 2020, pp. 1159–1174. DOI: 10.1145/3385412.3386027.

[53] Civilized Discourse Construction Kit Inc. *Discourse: A platform for community discussion*. https://github.com/discourse/discourse. 2020.

[54] Michael B James et al. “Digging for fold: synthesis-aided API discovery for Haskell”. In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020), pp. 1–27. DOI: 10.1145/3428273.

[55] Susmit Jha et al. “Oracle-guided component-based program synthesis”. In: *2010 ACM/IEEE 32nd International Conference on Software Engineering*. Vol. 1. IEEE. 2010, pp. 215–224. DOI: 10.1145/1806799.1806833.

[56] Andrew Johnson et al. “Exploring and Enforcing Security Guarantees via Program Dependence Graphs”. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. ACM, 2015. DOI: 10.1145/2737924.2737957.

[57] Milod Kazerounian et al. “Type-level computations for Ruby libraries”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, pp. 966–979. DOI: 10.1145/3314221.3314630.

[58] Jinwoo Kim et al. “Semantics-guided synthesis”. In: *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–32. DOI: 10.1145/3434311.

[59] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. “Strongly Typed Heterogeneous Collections”. In: *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*. Haskell '04. Snowbird, Utah, USA: Association for Computing Machinery, 2004, pp. 96–107. ISBN: 1581138504. DOI: 10.1145/1017472.1017488.

[60] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. “SMT-based model checking for recursive programs”. In: *Formal Methods Syst. Des.* 48.3 (2016), pp. 175–205. DOI: 10.1007/s10703-016-0249-4.

[61] Boris Köpf and Andrey Rybalchenko. "Approximation and Randomization for Quantitative Information-Flow Analysis". In: *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*. IEEE Computer Society, 2010, pp. 3–14. DOI: 10.1109/CSF.2010.8.

[62] Martin Kucera et al. "Synthesis of Probabilistic Privacy Enforcement". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. ACM, 2017, pp. 391–408. DOI: 10.1145/3133956.3134079.

[63] Woosuk Lee et al. "Accelerating search-based program synthesis using learned probabilistic models". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. Ed. by Jeffrey S. Foster and Dan Grossman. ACM, 2018, pp. 436–449. DOI: 10.1145/3192366.3192410.

[64] Nico Lehmann et al. "STORM: Refinement Types for Secure Web Applications". In: *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. USENIX Association, 2021, pp. 441–459. URL: https://www.usenix.org/conference/osdi21/presentation/lehmann.

[65] Peng Li and Steve Zdancewic. "Encoding Information Flow in Haskell". In: *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), 5-7 July 2006, Venice, Italy*. IEEE Computer Society, 2006, p. 16. DOI: 10.1109/CSFW.2006.13.

[66] Sheng Liang, Paul Hudak, and Mark Jones. "Monad Transformers and Modular Interpreters". In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. San Francisco, California, USA: Association for Computing Machinery, 1995, pp. 333–343. ISBN: 0897916921. DOI: 10.1145/199448.199528.

[67] Justin Lubin et al. "Program sketching with live bidirectional evaluation". In: *Proceedings of the ACM on Programming Languages* 4.ICFP (2020), 109:1–109:29. DOI: 10.1145/3408991.

[68] Piotr Mardziel et al. "Dynamic enforcement of knowledge-based security policies using probabilistic abstract interpretation". In: *Journal of Computer Security* 21.4 (2013), pp. 463–532. DOI: 10.3233/JCS-130469.

[69] J.L. Massey. "Guessing and entropy". In: *Proceedings of 1994 IEEE International Symposium on Information Theory*. 1994, pp. 204–. DOI: 10.1109/ISIT.1994.394764.

[70] Bertrand Meyer. "Framing the Frame Problem". In: *Dependable Software Systems Engineering*. Vol. 40. IOS Press, 2015, pp. 193–203.

[71] Jaideep Nijjar and Tevfik Bultan. "Bounded verification of Ruby on Rails data models". In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. 2011, pp. 67–77. DOI: 10.1145/2001420.2001429.

[72] Peter W. O'Hearn. "Incorrectness logic". In: *Proc. ACM Program. Lang.* 4.POPL (2020), 10:1–10:32. DOI: 10.1145/3371078. URL: https://doi.org/10.1145/3371078.

[73] Hakjoo Oh et al. "Design and implementation of sparse global analyses for C-like languages". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. ACM, 2012, pp. 229–238. DOI: 10.1145/2254064.2254092.

[74] OpenAI. *GPT-4 Technical Report*. 2023. arXiv: 2303.08774 [cs.CL].

[75] Alfonso Ortega, Marina de la Cruz, and Manuel Alfonseca. "Christiansen Grammar Evolution: Grammatical Evolution With Semantics". In: *IEEE Trans. Evol. Comput.* 11.1 (2007), pp. 77–90. DOI: 10.1109/TEVC.2006.880327.

[76] Peter-Michael Osera and Steve Zdancewic. "Type-and-example-directed program synthesis". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. Ed. by David Grove and Stephen M. Blackburn. ACM, 2015, pp. 619–630. DOI: 10.1145/2737924.2738007.

[77] James Parker, Niki Vazou, and Michael Hicks. "LWeb: information flow security for multi-tier web applications". In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), 75:1–75:30. DOI: 10.1145/3290388.

[78] Daniel Perelman et al. "Test-driven synthesis". In: vol. 49. 6. ACM New York, NY, USA, 2014, pp. 408–418. DOI: 10.1145/2666356.2594297.

[79] Phitchaya Mangpo Phothilimthana et al. "Swizzle Inventor: Data Movement Synthesis for GPU Kernels". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*. ACM, 2019, pp. 65–78. DOI: 10.1145/3297858.3304059.

[80] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. "Program synthesis from polymorphic refinement types". In: vol. 51. 6. ACM New York, NY, USA, 2016, pp. 522–538. DOI: 10.1145/2908080.2908093.

[81] Nadia Polikarpova and Ilya Sergey. "Structuring the synthesis of heap-manipulating programs". In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–30. DOI: 10.1145/3290385.

[82] Nadia Polikarpova et al. "Liquid information flow control". In: *Proceedings of the ACM on Programming Languages* 4.ICFP (2020), 105:1–105:30. DOI: 10.1145/3408987.

[83] Oleksandr Polozov and Sumit Gulwani. "FlashMeta: A Framework for Inductive Program Synthesis". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2015. Pittsburgh, PA, USA: Association for Computing Machinery, 2015, pp. 107–126. ISBN: 9781450336895. DOI: 10.1145/2814270.2814310.

[84] Corneliu Popeea and Wei-Ngan Chin. "Inferring Disjunctive Postconditions". In: *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, 11th Asian Computing Science Conference, Tokyo, Japan, December 6-8, 2006, Revised Selected Papers*. Vol. 4435. Lecture Notes in Computer Science. Springer, 2006, pp. 331–345. DOI: 10.1007/978-3-540-77505-8_26.

[85] François Pottier and Vincent Simonet. "Information flow inference for ML". In: *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*. ACM, 2002, pp. 319–330. DOI: 10.1145/503272.503302.

[86] Mukund Raghothaman and Abhishek Udupa. "Language to Specify Syntax-Guided Synthesis Problems". In: *CoRR* abs/1405.5590 (2014). arXiv: 1405.5590. URL: http://arxiv.org/abs/1405.5590.

[87] Jeff Reback et al. *pandas-dev/pandas: Pandas 1.4.4*. Version v1.4.4. Aug. 2022. DOI: 10.5281/zenodo.7037953.

[88] Brianna M Ren and Jeffrey S Foster. "Just-in-time static type checking for dynamic languages". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2016, pp. 462–476. DOI: 10.1145/2908080.2908127.

[89] Andrew Reynolds and Cesare Tinelli. "SyGuS Techniques in the Core of an SMT Solver". In: *Proceedings Sixth Workshop on Synthesis, SYNTCAV 2017, Heidelberg, Germany, 22nd July 2017*. Ed. by Dana Fisman and Swen Jacobs. Vol. 260. EPTCS. 2017, pp. 81–96. DOI: 10.4204/EPTCS.260.8.

[90] Andrew Reynolds et al. "Counterexample-Guided Quantifier Instantiation for Synthesis in SMT". In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*. Ed. by Daniel Kroening and Corina S. Pasareanu. Vol. 9207. Lecture Notes in Computer Science. Springer, 2015, pp. 198–216. DOI: 10.1007/978-3-319-21668-3_12.

[91] Alejandro Russo. "Functional pearl: two can keep a secret, if one of them uses Haskell". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. ACM, 2015, pp. 280–288. DOI: 10.1145/2784731.2784756.

[92] Andrei Sabelfeld and Andrew C. Myers. "Language-based information-flow security". In: *IEEE Journal on Selected Areas in Communications* 21.1 (2003), pp. 5–19. DOI: 10.1109/JSAC.2002.806121.

[93] Andrei Sabelfeld and David Sands. "Declassification: Dimensions and principles". In: *Journal of Computer Security* 17.5 (2009), pp. 517–548. DOI: 10.3233/JCS-2009-0352.

[94] Claude E. Shannon. "A mathematical theory of communication". In: *ACM SIGMOBILE Mobile Computing and Communications Review* 5.1 (2001), pp. 3–55. DOI: 10.1145/584091.584093.

[95] Geoffrey Smith. "On the Foundations of Quantitative Information Flow". In: *Foundations of Software Science and Computational Structures, 12th International Conference, FOSSACS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. Vol. 5504. Lecture Notes in Computer Science. Springer, 2009, pp. 288–302. DOI: 10.1007/978-3-642-00596-1_21.

[96] Sunbeom So and Hakjoo Oh. "Synthesizing Imperative Programs from Examples Guided by Static Analysis". In: *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings*. Ed. by Francesco Ranzato. Vol. 10422. Lecture Notes in Computer Science. Springer, 2017, pp. 364–381. DOI: 10.1007/978-3-319-66706-5_18.

[97] Armando Solar-Lezama. "Program sketching". In: *Int. J. Softw. Tools Technol. Transf.* 15.5-6 (2013), pp. 475–495. DOI: 10.1007/s10009-012-0249-7.

[98] Armando Solar-Lezama et al. "Combinatorial sketching for finite programs". In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*. ACM, 2006, pp. 404–415. DOI: 10.1145/1168857.1168907.

[99] Deian Stefan et al. "Flexible dynamic information flow control in Haskell". In: *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011*. ACM, 2011, pp. 95–106. DOI: 10.1145/2034675.2034688.

[100] Ian Sweet et al. "What's the Over/Under? Probabilistic Bounds on Information Leakage". In: *Principles of Security and Trust - 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. Vol. 10804. Lecture Notes in Computer Science. Springer, 2018, pp. 3–27. DOI: 10.1007/978-3-319-89722-6_1.

[101] Emina Torlak and Rastislav Bodik. "Growing solver-aided languages with rosette". In: *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. 2013, pp. 135–152. DOI: 10.1145/2509578.2509586.

[102] Emina Torlak and Rastislav Bodik. "A lightweight symbolic virtual machine for solver-aided host languages". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* 49.6 (2014), pp. 530–541. DOI: 10.1145/2594291.2594340.

[103] Niki Vazou, Patrick Maxim Rondon, and Ranjit Jhala. "Abstract Refinement Types". In: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. Vol. 7792. Lecture Notes in Computer Science. Springer, 2013, pp. 209–228. DOI: 10.1007/978-3-642-37036-6_13.

[104] Niki Vazou et al. “Refinement Types for Haskell”. In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP '14. Gothenburg, Sweden: Association for Computing Machinery, 2014, pp. 269–282. ISBN: 9781450328739. DOI: 10.1145/2628136.2628161.

[105] Niki Vazou et al. “Refinement reflection: complete verification with SMT”. In: *Proceedings of the ACM on Programming Languages* 2.POPL (2018), 53:1–53:31. DOI: 10.1145/3158141.

[106] Niki Vazou et al. “Theorem Proving for All: Equational Reasoning in Liquid Haskell (Functional Pearl)”. In: *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*. Haskell 2018. St. Louis, MO, USA: Association for Computing Machinery, 2018, pp. 132–144. ISBN: 9781450358354. DOI: 10.1145/3242744.3242756.

[107] Martin T. Vechev, Eran Yahav, and Greta Yorsh. “Abstraction-guided synthesis of synchronization”. In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. Ed. by Manuel V. Hermenegildo and Jens Palsberg. ACM, 2010, pp. 327–338. DOI: 10.1145/1706299.1706338.

[108] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. “Synthesizing highly expressive SQL queries from input-output examples”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. Ed. by Albert Cohen and Martin T. Vechev. ACM, 2017, pp. 452–466. DOI: 10.1145/3062341.3062365.

[109] Chenglong Wang et al. “Visualization by example”. In: *Proc. ACM Program. Lang.* 4.POPL (2020), 49:1–49:28. DOI: 10.1145/3371117.

[110] Xinyu Wang, Isil Dillig, and Rishabh Singh. “Program Synthesis Using Abstraction Refinement”. In: *Proc. ACM Program. Lang.* 2.POPL (2017). DOI: 10.1145/3158151.

[111] Yuepeng Wang et al. “Synthesizing database programs for schema refactoring”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, pp. 286–300. DOI: 10.1145/3314221.3314588.