Practical Synthesis from Real-World Oracles

Bruce Collie

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2022

Abstract

As software systems become increasingly heterogeneous, the ability of compilers to reason about an entire system has decreased. When components of a system are not implemented as traditional programs, but rather as specialised hardware, optimised architecture-specific libraries, or network services, the compiler is unable to cross these abstraction barriers and analyse the system as a whole.

If these components could be modelled or understood as programs, then the compiler would be able to reason about their behaviour without concern for their internal implementation details: a homogeneous view of the entire system would be afforded. However, it is not often the case that such components ever corresponded to an original program. This means that to facilitate this homogenenous analysis, programmatic models of component behaviour must be learned or constructed automatically.

Constructing these models is an inductive program synthesis problem, albeit a challenging one that is largely beyond the ability of existing implementations. In order for the problem to be made tractable, information provided by the underlying context (i.e. the real component behaviour to be matched) must be integrated.

This thesis presents three program synthesis approaches that integrate contextual information to synthesise programmatic models for real, existing components. The first, Annote, exploits informally-encoded information about a component's interface (e.g. from documentation) by weaving that information into an extended type-and-attribute system for component interfaces. The second, Presyn, learns a pair of cooperating probabilistic models from prior syntheses, that aim to predict likely program structure based on a component's interface. Finally, Haze uses observations of common side-effects of component executions to bias the search for programs. These approaches are each evaluated against comparable synthesisers from the literature, on a set of benchmark problems derived from real components.

Learning models for component behaviour is only a partial solution; the compiler must also have some mechanism to use those models for program analysis and transformation. This thesis additionally proposes a novel mechanism for context-sensitive automatic API migration based on synthesised programmatic models, and evaluates the effectiveness of doing so on real application code.

In summary, this thesis proposes a new framing for program synthesis problems that target the behaviour of real components, and demonstrates three different potential approaches to synthesis in this spirit. The success of these approaches is evaluated against implementations from the literature, and their results used to drive a novel API migration technique.

Acknowledgements

Thanks must go first to my supervisor, Michael O'Boyle, for his support and advice throughout the duration of my PhD. The ideas in this thesis would not have made it onto the page in this form without his guiding hand and insightful questions. Secondly, thanks also those who I worked alongside to produce the papers comprising this thesis: Philip, Jackson and Ajitha, whose contributions cannot be understated.

The first part of my time at Edinburgh was a thoroughly enjoyable and social time which I was sad to cut short; being part of the Pervasive Parallelism scheme and the broader ICSA community was a real privilege. Thanks in particular to Jack, whose time in Edinburgh mirrored mine closely, for being an invaluable sounding board for both work and non-work discussions.

My entire family have been a source of great encouragement throughout my studies, from my first day of undergraduate to finishing my thesis, for which I am forever grateful. In particular, the curiousity and love for learning instilled in me by my dad has carried me through the last nine years of study without it ever feeling like a burden.

Finally, to my fiancée Alice, thank you for your love, patience and generosity of spirit. I love you, and nothing I do would be possible without you.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following papers and preprints:

[1] Bruce Collie and Michael F. P. O'Boyle. Augmenting Type Signatures for Program Synthesis. *arXiv:1907.05649 [cs]*, July 2019

[2] Bruce Collie, Philip Ginsbach, and Michael F. P. O'Boyle. Type-Directed Program Synthesis and Constraint Generation for Library Portability. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 55–67, September 2019. doi: 10.1109/PACT.2019.00013

[3] Bruce Collie and Michael F.P. O'Boyle. Retrofitting Symbolic Holes to LLVM IR. *arXiv:2006.05875 [cs]*, June 2020

[4] Bruce Collie, Philip Ginsbach, Jackson Woodruff, Ajitha Rajan, and Michael F. P. O'Boyle. M3: Semantic API Migrations. In *Proceedings of the Thirty-Fifth International Conference on Automated Software Engineering*, ASE '20, Virtual Event, Australia, 2020. ACM. ISBN 978-1-4503-6768-4. doi: 10.1145/3324884.3416618

[5] Bruce Collie, Jackson Woodruff, and Michael F. P. O'Boyle. Modeling Black-Box Components with Probabilistic Synthesis. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2020, pages 1–14, New York, NY, USA, November 2020. Association for Computing Machinery. ISBN 978-1-4503-8174-1. doi: 10.1145/3425898.3426952

[6] Bruce Collie and Michael F. P. O'Boyle. Program Lifting using Gray-Box Behavior. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 60–74, September 2021. doi: 10.1109/PACT52795.2021.00012

(*author*)

Table of Contents

Chapter 1

Introduction

1.1 Understanding Systems with Synthesis

Being able to model the behaviour of complex systems is a vital tool in software engineering. If a system can be modelled, then it can be reasoned about in a structured way; this reasoning then allows for changes to its implementation to be made, or for it to be combined with other systems while preserving abstraction.

The most common model for the behaviour of a system is to express that behaviour as a *program*. Programs offer a structured representation that can be analysed and manipulated using existing tools, without imposing the overhead and complexity of a fully formally-specified or verified system. Historically, research into optimising compilers has demonstrated the utility of programmatic reasoning: by analysing and transforming the local behaviour of a program, significant improvements can be made to the system modelled by that program.

When a system is implemented entirely as a locally-known program, this type of programmatic reasoning is available "for free", as every possible behaviour of the system is captured by its code. Additionally, little or no additional work is required by the maintainers of the system to support analysis. However, this scenario is becoming increasingly rare in practice. Today, more and more systems make use of *opaque* third-party components that do not share the same programmatic representation (for example, they might be libraries distributed as compiled binaries, calls to a network service, or interfaces to hardware). In these situations, existing methods for analysing behaviour programmatically break down.

It is clear that as the degree of heterogeneity in system implementation continues to grow, existing programmatic approaches to analysing system behaviour will become less useful. However, if a programmatic model for the behaviour of an opaque component could be obtained, then gradual progress towards reasoning about the whole system can be made. One way to do so is to require developers to annotate opaque components with formal descriptions of their expected behaviour. However, this places a significant burden on the system developers: an automated or partially automated method for modelling the behaviour of opaque components would be an improvement to this scenario.

A potential solution to the problem of extending programmatic reasoning to systems with opaque components is *program synthesis*: the automatic construction of a program that exhibits behaviour equivalent to that component. If such a program can be constructed for each component in a system through observations of its behaviour, then the "gaps" in the programmatic model of the system can be filled in.

However, program synthesis is an open research problem; even state-of-the-art systems for general synthesis are unable to synthesise programs matching the complexity of typical system components without substantial external assistance. To do so, additional contextual information from the system must be combined with observations of the component's behaviour.

1.2 Synthesis from Real Objects

The problem posed above (automatically discovering the behaviour of general components through synthesis) raises a subtle point relative to much of the existing program synthesis literature. Synthesis tasks where a specification is provided by observations of behaviour are typically known as *oracle-guided*. The oracles in question are almost always treated abstractly; observations of their behaviour are taken axiomatically without substantial examination of how that behaviour arises in practice.

However, in the context proposed above (modelling the behaviour of a particular component), the observed behaviours are obtained directly from the execution of a concrete component. This means that on one hand, observing behaviour requires resources (e.g. execution time or power) to be explicitly consumed; on the other hand, it means that additional information beyond the core functionality of the component can be acquired. Rather than just understanding *what* a component does, in this context some observation of *how* it goes about its work is possible.

The synthesis problems that must be solved to model the behaviour of real-world components are beyond the complexity typically achievable by existing synthesisers, and so a hook to reduce the problem to a more tractable one is necessary. The additional behavioural information exposed by real components about their internal mechanisms and interfaces provides this hook. By considering sources of information that, in isolation, cannot be used to determine the correctness of a potential solution, the space of programs to be searched during synthesis can be reduced in size. This type of reasoning is common in synthesis research; almost every technique requires some kind of domain knowledge to be exploited for the process to be viable.

1.3 Understanding New Opportunities

While modelling the behaviour of system components as programs using synthesis is an interesting problem in the abstract, it is necessary to establish a motivating scenario where this this capability is useful in practice.

Consider a hand-written application that exhibits a computational bottleneck which would be desirable to optimise. The developers of this application may be aware at a high level that a particular component (e.g. a library or specialised hardware device) may somehow be able to improve the performance of that bottleneck, but be unable to take advantage of it. There are several reasons why this might be the case: a lack of knowledge of the low-level details of the component, insufficient maintenance resources, or a desire to keep the application portable without locking it in to a particular platform.

If the behaviour of the optimised component could be learned using synthesis, then these problems can be ameliorated. By modelling its behaviour as a program, application developers can be more confident of the details required to use the component. Similarly, the effort required to modify the application code to use the component is reduced; this leads naturally to easier changes to the application and simplified portability.

This scenario leads naturally to a workflow, which will be explored in this thesis. First, programs that model the behaviour of components are synthesised (the precise method for this depends on the contextually available information about the component's behaviour). Then, the synthesised program can be used with existing library migration tooling to port and optimise applications.

1.4 Structure of this Thesis

The remainder of this thesis is laid out as follows.

Following the introduction, **Chapter 2** examines prior and related work in the literature. A careful examination of program synthesis and its associated theory is given. Several variants of synthesis including sketch-based, neural and stochastic are considered in detail along with associated results and benchmark achievements. Additionally, related work in automatic program optimisation, API migration and legacy code repair is examined.

Then, **Chapter 3** sets out the requisite background technical knowledge and context for this thesis, including important terminology and definitions. An introduction to program synthesis is given, as well as an overview of the optimising compiler toolchains that underpin the research in this thesis.

Chapters 4 to 7 form the bulk of the technical and research contributions. They cover, in large part, four research papers published during the course of this thesis. In these chapters, a methodology for automatic program optimisation and refactoring using program synthesis is developed.

First, **Chapter 4** considers the synthesis of a narrow set of performance-critical functions found in scientific applications. To do so, a domain-specific language is introduced that extends a host type system with arbitrary properties. A query language is used to heuristically direct synthesis based on the properties of a problem's type signature. Finally, an experimental analysis is performed to determine whether this regime can produce meaningful performance improvements.

In **Chapter 5**, this system is extended to include a probabilistic component where two cooperative models *predict* relevant properties of a synthesis problem. By doing so, less user input is required and greater generality can be achieved by the synthesiser. The synthesiser developed in this chapter is compared to four other leading implementations on a large set of synthesis benchmarks drawn from the literature.

Performance improvements are only one of the possible reasons to perform automated refactoring of code. Doing so in general is known as API migration, and is an open research area in software engineering. **Chapter 6** examines the synthesis results achieved previously in the context of an API migration problem.

Type signature annotations and probabilistic models are two orthogonal ways in which a synthesiser can be extended to incorporate information other than a correctness specification. In **Chapter 7**, a new synthesiser is developed that uses multiple sources of additional information (obtained dynamically from an executing oracle) to reduce the size of its search spaces.

Finally, **Chapter 8** summarises the contributions made in this thesis and identifies possible avenues for future research to explore.

1.5 Contributions

This thesis contributes three comparable approaches to program synthesis: Annote, Presyn and Haze; each one attacks the problem of synthesis from an oracle backed by a real-world component using different assumptions and techniques. Annote uses contextual information supplied by component vendors, Presyn learns predictive probabilistic models that encode problem structure, and Haze uses information obtained from dynamic executions of the oracle to partially constrain the structure of potential solutions.

To evaluate these approaches, large sets of benchmark tasks are collected from the literature. These benchmarks cover a broader set of problem domains than previous comparisons do, and are implemented so as to facilitate a fair comparison against other leading synthesisers. Presyn and Haze both demonstrate state-of-the-art performance across the set of benchmarks used, scaling to more difficult problems than previous work.

A methodology is developed to exploit synthesised programs for API migration tasks; by doing so, substantial performance improvements can be realised on real-world legacy programs. Additionally, the approach generalises well and allows for the discovery of complex API migrations that consider both existing library calls and their context within an application.

1.6 Summary

Modelling the behaviour of opaque components as programs using synthesis strikes a balance between practicality and strength of analysis. By doing so, component models can be analysed homogeneously, using the same tools as for “natural” application code. As an example use case, this leads to intuitive opportunities for library migration and portable code rejuvenation. However, synthesising the behaviour of arbitrary components is challenging, and requires the integration of contextual information where it is available. The remainder of this thesis proposes methods for the synthesis of programmatic models for opaque components, as well as their application to code rejuvenation and modernisation.

Chapter 2

Related Work

This chapter summarises the diverse body of work relevant to the content of this thesis, focusing on four key areas of interest. First, to contextualise the overall aims of this thesis, the role of traditional compiler analyses and techniques in improving the performance of legacy code is examined. Similarly, extensions to the traditional compilation model that have similar objectives are examined. Next, the field of general program synthesis is broken down into more specific subdomains relevant to this thesis. A more specific analysis is then given to lifting techniques, where code or specifications are recovered from obfuscated or low-level implementations. Finally, an overview of automated software engineering techniques that aim to achieve similar goals as this thesis is given.

2.1 Compiler Analysis and Code Rejuvenation

2.1.1 Traditional Compiler Analysis

Historically, programmers were more concerned with making sure that their programs were correct (and indeed, that they would run at all on the systems of the time). As programs became larger and more complex, it became more and more important for the compiler to automatically produce *optimal* machine code from user programs. One of the first practical optimising compilers is described by Wulf et al. [7] for the BLISS language, targeting the PDP-11.

Since then, enormous effort has been focused on the design and implementation of optimising compilers for every mainstream programming language. The LLVM project [8] implements an intermediate representation (IR) that allows for optimisations and analyses to be more easily expressed, eventually spawning the Clang C compiler. Similarly, projects such as GCC [9] and GHC [10] represent hundreds of thousands of developer-hours of effort towards producing faster compiled programs.

Typically, the optimisations applied by traditional compilers are *conservative* and *local* in nature. To avoid introducing new bugs into user programs through optimisation, compiler developers must be certain that their transformations are correct; for example, the Alive2 tool [11] aims to formally verify optimisations performed by LLVM, and has identified 47 novel bugs in the process of doing so. As a result, the pace of compiler development can be slow. The CAnDL [12] language offers a domain-specific language for compiler analyses; by doing so, the number of lines of code required to write an analysis is reduced greatly, allowing developers to iterate more quickly.

Research and development of compiler optimisations grew from the need to emit the best possible sequence of instructions for a particular program or function; this optimisation problem is inherently local, leading to a proliferation of research into low-level optimisations. Chakraborty [13] offers a summary of fifty years of research into peephole optimisation, where small windows of machine instructions are analysed with no consideration of the broader program context. Even as recently as 2010, analysing the entire content of a program at once was not a practical way to improve compiler output [14].

Despite these shortcomings, traditional compiler analyses and techniques continue to slowly adapt to the changing demands made of them. For example, CREV [15] offers state-of-the-art granularity in applying auto-vectorisation optimisations to scalar programs. Formal methods such as the polyhedral loop model [16] continue to provide fruitful mechanisms for verified reasoning; the `polyhedral.info` research community cite over 140 papers building on the core formalism. The application of traditional techniques such as constant-propagation and dead code elimination to as-yet unexplored problem domains has also proved to be profitable; by applying these optimisations to real-world graphics shaders, substantial performance improvements can be made on real programs [17].

Even so, a need for improved understanding of the *high-level semantics* of programs being compiled has been recognised by the community; for example, the MLIR project [18] builds on the infrastructural lessons learned from building LLVM, but allows for a far wider range of abstractions to be expressed in intermediate programs. By doing so, domain-specific semantic transformations can be implemented without costly pattern-recognition or lifting steps. Relatedly, Wang and O'Boyle [19] identify that the use of machine-learning techniques in compilers is increasingly prevalent, and allows for complex patterns or transformations to be obtained without a human expert.

2.1.2 Domain-Specific Languages and Compilers

To allow the compiler to understand and interpret high-level semantic information about a program's behaviour, a common strategy is to express programs in a domain-specific language (DSL) that better captures the abstract nature of the computation. This allows for optimisations to be applied that would be impossible or impractical to implement on a lower-level representation.

The Halide language [20] implements an embedded C++ DSL specifically for image-processing pipelines. By specifying high-level operations in a Halide program, their precise implementations and execution schedule can be computed or tuned by the compiler. The LIFT compiler IR [21] provides a functional representation that can be optimised specifically for a given target device using a series of rewrite rules. Domain-specific languages can be implemented in terms of the LIFT abstraction; for example, for stencil computations [22]. LiLAC [23] is a DSL for sparse linear algebra kernels; the kernels can be compiled to efficient harnesses to the appropriate library functions.

Perhaps the most widely-used domain-specific compilers at present are those for machine-learning applications, across a wide range of abstraction levels. At a high level, frameworks like TensorFlow [24] and PyTorch [25] compile descriptions of a neural network (explicit graphs and imperative Python code, respectively) to efficiently scheduled models. At a lower level, tools like TACO [26] and LGen [27] compile abstract specifications for tensor algebra kernels into efficient code in a general-purpose language.

2.1.3 Optimised Libraries and Hardware

As hardware becomes increasingly specialised, the software libraries running on that hardware have necessarily followed suit to maintain competitive performance. It is no longer sufficient to provide the fastest hardware or the best-optimised library; the co-design of the two is critical. Large commercial vendors such as Intel [28] and Nvidia [29] provide proprietary routines to optimise common computations on their own hardware, while efforts like OpenCL [30] attempt to provide unifying standardised versions of the same.

For library developers who are unable or unwilling to lock their users into a single hardware platform, an appealing middle ground is to *tune* parameters of the library to better fit the hardware being targeted. For example, libraries performing linear algebra computations are commonly optimised by doing so; Xu et al. [31] demonstrate the optimisation of matrix-vector products on the GPU, while Byun et al. [32], Borštnik et al. [33] and Elafrou et al. [34] all explore the parameter space of sparse matrix implementations on different platforms.

Automatic tuning can be applied to more complex problem domains. Mullapudi et al. [35] optimise real-world image-processing pipelines for particular hardware, while the LIFT compiler [21] provides language-level support for tuning the performance of *arbitrary* applications.

However, the barrier to entry for developing novel hardware continues to decrease. This means that instead of developing and tuning a software library to run on general-purpose hardware, a specialised hardware accelerator can be built instead. By sacrificing generality, the scaling limitations of general-purpose compute [36, 37] can be overcome, leading to improved performance. In recent years, there has been a Cambrian explosion of these devices, in a hugely varied set of application domains: machine learning [38, 39, 40], physical system simulation [41] and regular-expression checking [42] are just three recent examples. Specialised hardware can be extremely specific; Koenig et al. [43] present the design of a hardware component that only computes an exact floating-point dot product.

2.2 General Program Synthesis

2.2.1 Inductive Synthesis

This thesis is concerned primarily with inductive synthesis, where the specification of a problem may be incomplete, and certainly cannot be transformed deterministically into a solution (as is the case for *deductive* synthesis). David and Kroening [44] offer a comprehensive summary of the history, challenges and outlook for the field of inductive synthesis; they conclude that while synthesis offers a powerful tool to solve problems that challenge human users, careful thought must be given to the design of synthesis tools and systems to allow them to scale and generalise.

Before beginning an overview of the inductive synthesis literature, it is useful to consider and keep in mind the three *dimensions* of synthesis identified by Gulwani [45]: "expression of user intent, space of programs over which to search, and the search technique". These dimensions capture succinctly the potential variations between synthesisers, and allow for meaningful comparisons to be drawn. Despite these dimensions, however, drawing direct comparisons between the achievements and results of synthesisers can often be a challenging problem in its own right. Pantridge et al. [46] explore the difficulty in comparing synthesisers that differ to a large extent in one or more of these dimensions. For example, they find that genetic synthesis methods often achieve more general results than comparable implementations, but require far greater computational resources to do so. Helmuth and Spector [47] propose and analyse the difficulty of a common set of benchmarks as an attempt to address this issue.

Perhaps the single greatest improvement to general inductive synthesis is the introduction of counterexample-guided feedback loops (known generally as CEGIS) [48, 49]. Under the CEGIS model, candidate programs are checked for correctness using a verification tool; if the candidate is incorrect, a counterexample is generated as evidence. The synthesiser integrates that example into its specification, then generates a new candidate. This process takes advantage of the ability of counterexamples to reject large *classes* of incorrect programs, meaning that the candidate search space can be pruned quickly.

Since its introduction, the CEGIS model has formed a starting point for further theoretical and practical research into inductive synthesis. Jha and Seshia [50] offer a theoretical extension of the CEGIS model (oracle-guided, OGIS) that considers oracles that can provide stronger or weaker information than a falsifying counterexample. Theoretical bounds on time and space complexity for idealised synthesisers that consume different classes of oracle information are given. Abate et al. [51] observe that the ability of the verification component in a CEGIS loop to produce useful constraints is a potential bottleneck. Their solution, CEGIS($\mathcal{T}$) parameterises a CEGIS loop with a stronger underlying theory solver that can provide better constraints on the structure of candidate programs (for example, by using a Fourier-Motzkin solver to eliminate linear inequalities).

The core ideas behind CEGIS can be abstracted behind higher-level tools. The Rosette project [52] extends the Racket programming language with a set of "solver-aided" primitives that can be used to implement the verification component of a CEGIS synthesiser. The result of doing so is a DSL for implementing specialised synthesisers; example applications include synthesising memory consistency models [53] or interpretable solutions to logic puzzles [54].

While CEGIS is a useful and powerful technique, it suffers from some key limitations in certain problem contexts. For example, Solar-Lezama [55, Lecture 10] highlights potential pathological cases for CEGIS loops: if each additional counterexample causes only a marginal decrease in the size of the search space (i.e. desired solution behaviours are very sparse), then the suggest-verify cycle is inefficient. David and Kroening [44] identify the need to work "meta-level" tasks (such as the identification of solution *structure*) into the CEGIS loop more effectively. Additionally, synthesised solution programs must admit verification of some kind.

Inductive synthesis is by no means restricted to the synthesis of "traditional", Turing-complete imperative or functional programs; as long as a target language supports verification, then programs in that language can be synthesised using a CEGIS loop. Bastani et al. [56] learn context-free grammars for the inputs accepted by a program, while Bavishi et al. [57] synthesise operations in a DSL for *repairing* programs. Other work has examined the synthesis of sound parallelisation strategies for existing programs [58, 59].

2.2.2 Sketching

The initial work on what would become the CEGIS methodology was carried out during the development of Sketch [48], a synthesis-enabled language that popularised the *sketching* approach to synthesis. Under this approach, a partial program with syntactic "holes" is supplied to the synthesiser, which is responsible for instantiating the holes with concrete values to produce a correct solution. In the original version of Sketch, the underlying language supports finite bit-vectors, and terminating programs can be verified by way of a formal semantics. If a user of Sketch is able to provide an adequate sketch to the system, a satisfying solution can be identified by discharging the underlying bit-vector program to a SAT solver.

Subsequent work extended Sketch to support concurrent semantics and complex data structures, as well as a broader notion of sketches and holes that permitted disconnected components to form solutions [49]. Solar-Lezama [60, 61] gives a detailed summary of this early work on sketching, and identifies key areas for future work; of particular relevance are the observation that high-level semantic insight will become more useful for synthesising complex programs, and the need for non-functional specifications to guide or inform synthesis.

While the initial presentation of Sketch suggested that sketches would be supplied by an end-user of the tool, the higher-level idea that synthesis could be split into two phases (identifying, in turn, the structure and details of a solution) has been developed and generalised to other scenarios. Wang et al. [62] use a permissive over-approximation of SQL query semantics to identify partial queries that could potentially meet a specification. These partial queries can then be instantiated by a search-based synthesis step. A generalisation of these ideas is iSQL, [63] which combines the same two-phase structure with interactive user feedback.

Other work has focused on extending the capabilities and semantics of the core sketch abstraction. By popularising a sketch language to include uninterpreted functions that are backed by "native" code, Singh et al. [64] demonstrate an effective mechanism for decomposing sketching synthesis problems into smaller components. The Synapse tool [65] defines a notion of *metasketches*, where the space of programs represented by a given sketch is equipped with an additional cost and gradient function that allows search to be directed to "better" sketches by way of gradient descent.

The syntax-guided synthesis (SyGuS) framework [66] provides a general abstraction and specification format for sketch-like synthesis. Under this model, all sketches are treated as productions of a context-free grammar, and particular synthesisers are viewed as instances of the parameterised SyGuS model (e.g. by having a different background theory). Work on the abstract SyGuS model exists outside of specific instantiations; for example, Lee et al. [67] use probabilistic CFG search to accelerate general syntax-guided synthesis.

2.2.3 Synthesis of Imperative Programs

Much of the synthesis literature deals with the synthesis of programs expressed in tree-like abstract syntax formulations. This is for good reason; implementing semantic analysis and search procedures over AST-like structures is simple, and domain-specific languages can be easily specified. For example, the FlashFill project uses a Turing-incomplete DSL of string-processing operations [68, 69]. Nevertheless, there is considerable interest in the synthesis of stateful, imperative programs.

One approach to the synthesis of imperative programs is to constrain the potential behaviour of those programs; the original Sketch [48] semantics constrain programs to finite loop bounds to allow for a SAT-solver compatible finite encoding. Newer work, such as Simpl [70] uses similar ideas, but with more sophisticated static analysis and abstract interpretation to discharge invalid programs and prune its search space. Shi et al. [71] use a similar over-approximation-based initial synthesis technique as Scythe [62], but applied to imperative control flow rather than SQL queries.

An entirely separate approach to synthesis is when synthesising linear sequences of assembly-like instructions. In this domain, different algorithms lead to state-of-the-art performance. For example, Makespeare [72] uses a hill-climbing genetic algorithm to evolve sequences of x86 instructions implementing complex programs with looping control flow. In more localised contexts, boolean SAT solvers can be used to precisely model short sequences of instructions such that an *optimal* replacement sequence can be identified. This task is *superoptimisation* [73, 74].

2.2.4 Programming by Example

One of the most intuitive statements of program synthesis is to specify correctness with a set of *examples*: given input x, a correct solution should produce output y. The task of generalising correctly from a set of examples to a solution is *programming by example* (PBE).

Much of the literature in the area deals with automating programmatic tasks that are easy to specify, but tedious or fiddly to implement. For example, the FlashFill project [68, 69] implements a PBE system for spreadsheet formulas; users provide a small number of examples, and a general formula is synthesised for them. Similar implementations exist (and have achieved widespread adoption and practical usage) in a number of other problem domains: TF-Coder [75] synthesises tensor computations that comprise neural network models, while Feng et al. [76] generate manipulations of dataframes in R.

As well as directly specifying the correctness of potential solutions, in some problem contexts it is possible to make inferences of the structure of those solutions based on the provided examples. λ^2 [77] synthesises transformations of algebraic data structures by pruning away constructors and functions that would produce output examples with an incorrect shape; an even more formal approach is taken by Frankle et al. [78], who interpret a PBE problem as one of type-checking, and use deductive techniques to discover correct solutions. Conversely, DeepCoder [79] uses *less* precise methods: a neural network is trained on input-output examples to predict whether certain functions will or will not be present in a correct solution.

Other work attempts to make the most efficient use possible of a small set of examples (for example, if providing those examples is difficult or expensive). An et al. [80] apply specific perturbations to provided examples, attempting to force synthesised solutions to respect abstract properties (such as permutation-invariance) without stating them explicitly. Perelman et al. [81] wait until a candidate program has been suggested before gathering further examples, under the assumption that doing so will reveal insights into the structure of that solution.

2.2.5 Neural Synthesis

An increasing trend in program synthesis is the use of machine learning and neural network techniques to improve synthesis, under the broad hypothesis that being able to learn patterns from large corpora in combination with existing formal techniques will allow for more challenging problems to be solved. Early work in neural program synthesis typically used the neural component to condition or train a sub-component of a traditional synthesiser. DeepCoder [79] is a good example of this model. More recently, improvements to the representational capacity of neural networks has led to the direct generation of programs from those networks becoming more popular. For example, tokens might be emitted by the decoder component of an LSTM, or an AST from a more complex tree decoder.

One of the reasons why neural approaches are popular is their ability to consider multiple facets of a specification homogeneously (by encoding them to the same internal representation). For example, Chen et al. [82] consider observations of a target program's execution to condition their network's generation of syntactic tokens, while Bunel et al. [83] incorporate the grammar of the target language into their model to reject syntactically invalid programs.

SketchAdapt [84] attempts to strike a balance between traditional synthesis (slow, provably correct) and direct neural generation (fast, potentially imprecise) by selecting the best strategy to use for a given problem. For example, if a new problem resembles a previous one, a solution can be directly extracted from the neural model; if no prior context is available, then a traditional synthesis procedure is invoked to search for a solution. DreamCoder [85] builds on these ideas by constructing an explicit two-phase synthesis loop. In the first phase, solutions to sub-problems are identified, while in the second they are combined and generalised to form a library of reusable components.

Neural methods can be applied widely to other aspects of program synthesis. Shin et al. [86] examine the best way to augment and structure input-output examples for neural PBE systems, to avoid bias in training processes. Kalyan et al. [87] use neural methods to predict the best sequence of rules to apply in a deductive synthesis environment.

2.2.6 Multi-modal Synthesis

A recent trend in synthesis is to use *non-correctness* specifications in addition to the usual correctness specifications. By adding additional constraints that alone are not sufficient to uniquely specify a solution, the search space can still be reduced in size. These additional constraints are new *modalities* of information. The multi-modal hypothesis is not unique to synthesis; machine learning research has demonstrated the use of additional input modalities to assist the training of neural networks [88].

Much of the initial work examining multi-modal synthesis focuses on the combination of natural language processing (NLP) techniques alongside traditional synthesis methods. Manshadi et al. [89] demonstrate one of the first attempts to do so; they observe that the respective ambiguities in natural language and programming-by-example specifications tend to cancel each other out to some extent. Further work [90] builds on these ideas, formalising the approach as "compositional" synthesis. Chen et al. [91] move away from traditional NLP techniques, instead applying a neural network to interpret natural language specifications for data wrangling problems. Their work treats partial sketches provided by the user as another, distinct modality.

Similar approaches have been used with great success to synthesise regular expressions. Chen et al. [92] build sketches of regular expressions by interpreting natural language specifications, while Ye et al. [93] attack the more general problem of identifying the solution that best matches the intent of the user's specification (i.e. is optimal with respect to a metric).

Some less common synthesis techniques can be understood through the lens of a multi-modal framework. For example, Heule et al. [94] demonstrate an approach to synthesising formal semantic models for instruction set architectures that comprises multiple phases. In later phases, learned outputs from earlier phases are used to build more complex results that would be intractable from their correctness specifications alone.

Other, overloaded, uses of the term "multi-modal synthesis" exist; Thakoor et al. [95] refer to a scenario where a set of ambiguous examples is provided, and the goal of synthesis is to produce a *set* of programs that minimally covers the entire set of examples (i.e. can explain the full range of distinct cases in the dataset).

2.2.7 Lifting

Synthesis and compilation are related processes; synthesis produces a high-level program by induction from incomplete examples, while compilation produces a low-level program by deduction from a high-level one. The term *lifting*[1] refers to a similar scenario, where a high-level program is produced by induction from a low-level one (i.e. the inverse of compilation). Because semantic information is lost during compilation, lifting a program to recover it is a more challenging problem.

The typical application of lifting is to recover a high-level representation of a compiled program that can then be recompiled more optimally. For example, Mendis et al. [96] observe that significant portions of Adobe's Photoshop image processing tool could be run more efficiently if they were to be reimplemented using the Halide compiler [20]. By observing dynamic traces of the application executing, they are able to recover formal abstract models of the relevant stencil kernels. These can then be recompiled for a substantial performance gain.

Lifting does not necessarily begin with compiled code; ordinary source code originally written for performance can be just as low-level from the perspective of static analysis tools. Kamil et al. [97], Ahmad et al. [98] use a similar target as Mendis et al. [96] (the Halide compiler), but lift from Fortran and C++ descriptions of image-processing libraries respectively. By incorporating the original programs into their tools, they are able to prove formally that the lifted solutions are in fact equivalent. This level of verification is not always possible.

[1] Often, the term decompilation is seen in the literature. Typically, this refers to the recovery of the original source code, while lifting may recover a representation never used in the original compilation process.

Much work in lifting does fall under the umbrella of decompilation (i.e. reversing part of a compilation pipeline). For example, Hasabnis and Sekar [99] demonstrate a method to learn decompilation rules through observations of a compiler's code generation phase. By doing so, they produce a general, portable framework for training lifters. Dasgupta et al. [100] summarise the current state-of-the-art for binary lifters of this kind, and propose a mechanism to compare implementations against each other for bug discovery.

2.3 Automated Software Engineering

2.3.1 API Migration

Perhaps the single most studied task in automated software engineering is API migration; the task of transforming an application that makes calls to API x with calls to API y, while maintaining its behaviour. Doing so has a number of use cases. For example, old, out-of-date API usages can be modernised, different languages can be supported, or library vendors can be changed. The subject of API migration at large is summarised and organised into a taxonomy of changes by Robillard et al. [101].

Typically, API migration tools are driven by statistical methods. This is because modelling the behaviour and semantics of many partial programs in potentially incomplete programs is intractable. In some limited cases, static observations of syntactic structure can be useful; Xing and Stroulia [102] compose a library of patterns likely to appear in library change logs, and map them onto user code refactorings. By doing so, applications can be automatically brought up to speed when their downstream dependencies are modified.

This approach typically does not scale, and most approaches use large corpora to drive their implementation. MAM [103] analyses the code of an application with multiple versions written in different languages, and identifies equivalent API pairs across the versions. By doing so over a large set of applications, common rules for cross-language migration can be identified. ᴊv2cs [104] develops this idea further by building comparable neural embeddings of Java and C# code contexts (i.e. the surrounding code for every vocabulary item in the language). Then, API calls with similar embeddings can be suggested as candidates for migration. Nguyen et al. [105] successfully generalise the embedding-based approach to several other automated software engineering tasks, such as method name suggestion.

As well as *performing* API migrations, a common theme in the literature is to infer properties of code based on observations of changes made to that code. For example, similar methods and APIs between different libraries can be identified by examining migrations from one library to another [106, 107]. Orthogonally, Alrubaye and Mkaouer [108] aim to identify the subset of changes made to an application that represent API migrations, in order to automatically generate migration log training data.

2.3.2 Program Repair and Rejuvenation

In contrast to the usually statistical nature of API migration, the process of *repairing* a faulty program tends to require a more semantic approach that integrates models of the program's behaviour. Often, synthesis or similar approaches are used to do so alongside more traditional API migration tools.

Repairing faults in individual programs is an important goal in automated software engineering. Phoenix [57] generates fixes for complaints generated by static analysis tools by mining many examples of code repairs from a corpus. Examples that trigger static analysis errors *before* a particular patch is made are used to provide information about how to perform repairs. Synthesis is used to combine learned examples into interpretable strategies for performing repairs. Nguyen et al. [109] transform faulty programs into problems that can be solved by traditional compiler analyses, generating repairs if the compiler is able to prove properties of the transformed program. Shaw et al. [110] develop a library of safe, composable program transformations that when combined, can be used to eliminate buffer overflow errors in C applications.

At a higher level of abstraction, bugs in specifications and implementations can be discovered using similar abstractions. For example, Alive2 [11] implements a formal semantics for LLVM intermediate representation. Compiler optimisations can then be implemented in terms of this formal semantics, and thereby be checked formally for correctness. Similarly, [53] combine informal litmus tests with partially-sketched models of memory consistency semantics; from these inputs, full specifications for valid memory models can be extracted and verified by hand. By doing so, ambiguities or mistakes in the specification can be identified.

As well as *repairing* bugs in faulty programs, automated software engineering can be used to rejuvenate or optimise correct (but sub-optimal) programs. For example, the Halide-targeting family of lifters described previously [96, 97, 98] fall into this category when their suggested changes are made to a program. Angstadt et al. [111] synthesise hardware implementations of common string operations performed by an application, then migrate the application code to call the hardware implementation. Ginsbach et al. [23, 112] describe a method for identifying common idiomatic computational patterns in application code, then automatically replacing identified instances with calls to optimised library functions using a custom compiler pass.

2.4 Summary

This chapter has examined prior work relevant to this thesis, under three broad headings. First, Section 2.1 summarised existing approaches to compiler-based optimisation and analysis of legacy code, as well as non-traditional compilation schemes designed to integrate better with contemporary hardware and software. Then, Section 2.2 examined inductive program synthesis and lifting techniques across a number of problem domains, with particular emphasis on those schemes where high-level information is recovered from obfuscated or lowered representations. Finally, Section 2.3 gave an overview of automated software engineering techniques that aim to achieve similar goals to this thesis (by using different underlying approaches).

Chapter 3

Technical Background[1]

This chapter describes the shared technical and implementation details referenced in Chapters 4 to 7. Throughout those chapters, appropriate references back to sections of this chapter are made whenever an implementation detail is relied upon.

3.1 Overview

The structure of this chapter is as follows: first, Section 3.2 describes the compiler and language context upon which both the program synthesis and API migration aspects of this thesis are built, including an overview of the LLVM intermediate representation (IR) [8] and a synthesis-specific compiler extension. Then, Section 3.3 adapts a paper section introducing a relevant tool implemented by Philip Ginsbach, a co-author of Collie et al. [2].

As the majority of this thesis is concerned with program synthesis and related techniques, Section 3.4 gives a general overview of important background knowledge and terminology from the field. Of particular importance is *sketching* program synthesis, where synthesised solutions are built up from smaller components. Section 3.5 provides the semantics of a sketch-based synthesis language used throughout the thesis. Finally, based on the previous sections, Section 3.6 describes a general framework for program synthesis.

Chapters 4 to 7 present several approaches to program synthesis that share some technical underpinnings. This chapter explains the design choices that underpin these details; in particular, the choice of LLVM IR as a target language for synthesis. By targeting LLVM, the CAnDL code search engine can be used with a novel genetic algorithm to identify application code with a structure compatible with synthesised examples. Additionally, the design constraints that lead to the specific type of synthesis

[1] Parts of this chapter are adapted from published research in Collie et al. [2], Collie and O'Boyle [3].

carried out in this thesis are contextualised in terms of the wider literature.

The design of a sketching language for synthesis is presented. This language allows for arbitrary imperative programs to be built up compositionally from individual fragments; the semantics of the particular fragments used to do so can be defined by the consuming application (in this thesis, synthesisers). By providing a general framework, synthesisers can share boilerplate code while implementing their own specific search procedures and fragment definitions.

3.2 Compiler Integration

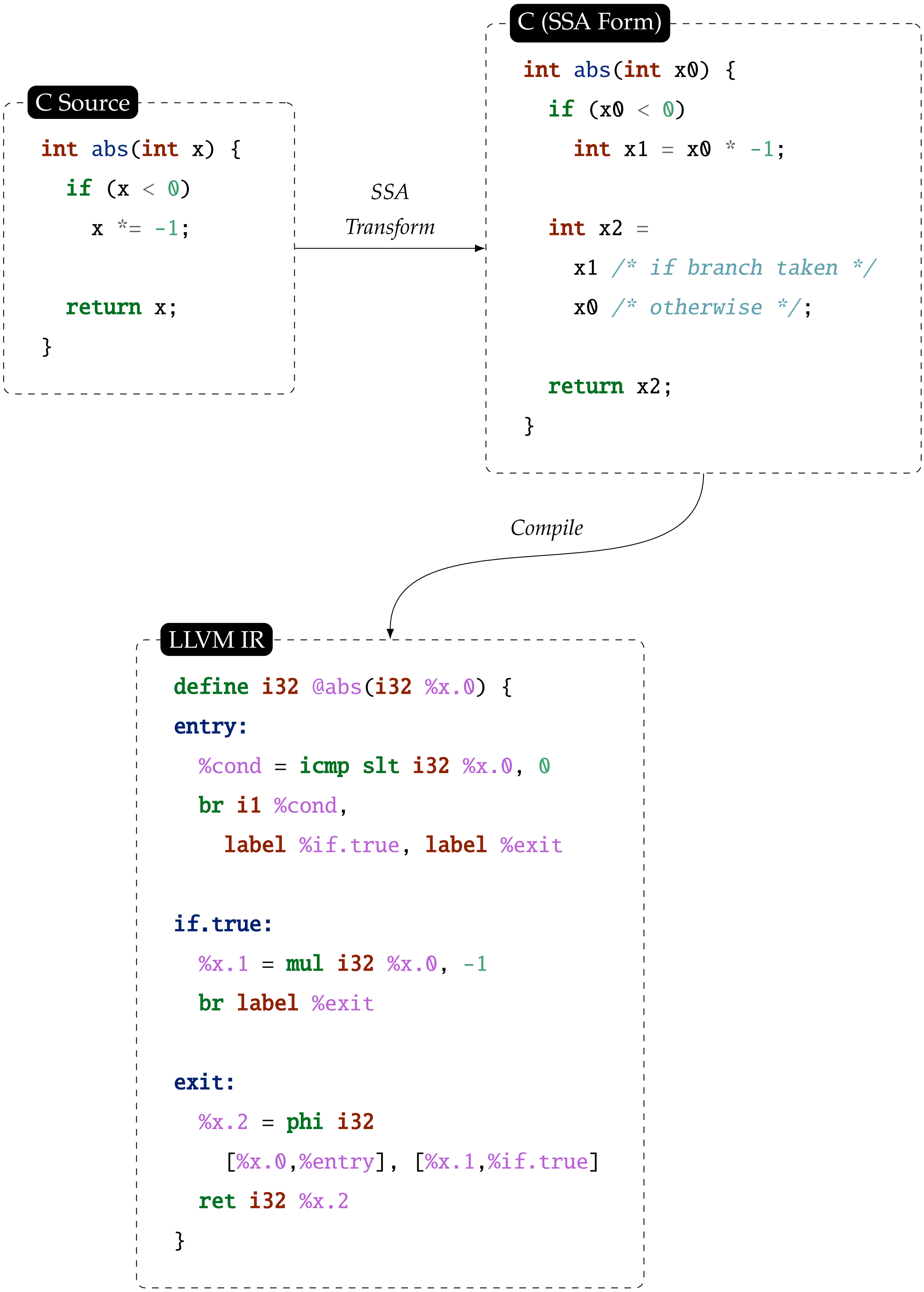
The initial motivation for this thesis (as given in Chapter 1) is to create a "smarter" compiler that is more able to automatically take advantage of new libraries and heterogeneous hardware. This thesis presents several related approaches to doing so; each of these is built as an extension to the open-source Clang/LLVM compiler toolchain. In this chapter, an introduction to this toolchain is given along with an analysis of the advantages and disadvantages of this choice.

Since its introduction, the Clang/LLVM toolchain has become the *de facto* choice for compilers research; the first-party tooling support and comprehensive documentation means that new techniques can be easily applied at different stages of the compilation pipeline. For example, Anderson et al. [113] add instrumentation code during compilation to check temporal logic assertions in C programs, Schardl et al. [114] extend LLVM's internals with first-class concurrency support, and Sasnauskas et al. [73] interpose a superoptimiser with the compiler to optimise short instruction sequences. Developing these diverse applications within GCC would be more cumbersome than within Clang/LLVM.

3.2.1 LLVM Intermediate Representation

At the heart of LLVM's comparative advantage for researchers and compiler developers is its *intermediate representation* (IR). To quote the LLVM project's own documentation:

> LLVM is a Static Single Assignment (SSA) based representation that provides type safety, low-level operations, flexibility, and the capability of representing 'all' high-level languages cleanly. It is the common code representation used throughout all phases of the LLVM compilation strategy.
> (*Official LLVM Documentation* [115])



```
int abs(int x) {
  if (x < 0)
    x *= -1;

  return x;
}
```

```
int abs(int x0) {
  if (x0 < 0)
    int x1 = x0 * -1;

  int x2 =
    x1 /* if branch taken */
    x0 /* otherwise */;

  return x2;
}
```

```
define i32 @abs(i32 %x.0) {
entry:
  %cond = icmp slt i32 %x.0, 0
  br i1 %cond,
    label %if.true, label %exit

if.true:
  %x.1 = mul i32 %x.0, -1
  br label %exit

exit:
  %x.2 = phi i32
    [%x.0,%entry], [%x.1,%if.true]
  ret i32 %x.2
}
```

Figure 3.1: A small C program, shown along with an LLVM IR program it could validly be compiled to. Conditionals are lowered to basic block branches, and mutable variables are converted to single static assignment form with ϕ nodes. Types similar to the basic integer types available in C are retained in the LLVM IR, as the overall structure of a function (call with parameters, return value).

This neatly summarises the design goals of the IR: to provide a uniform, target- and source language-independent representation that high-level languages can be lowered to during compilation. This section gives an overview of the most important features of LLVM IR as it is used in the remainder of the thesis. Figure 3.1 shows a running example referenced throughout the section that demonstrates the transformation from C to LLVM IR.

3.2.1.1 Program Structure

LLVM IR programs are arranged in a simple hierarchical structure. Their top-level object is a **module** (analogous to a C translation unit), each of which contains a set of **global values** and **functions**. A function comprises several **basic blocks**; these are each linear sequences of **instructions** that must be terminated with a control-flow instruction (e.g. a branch or return). Functions accept parameters and return a value similarly to C functions.

Control flow within an LLVM function is encoded with branches between basic blocks (either conditionally or unconditionally); the successor relationship between basic blocks directly provides the control-flow graph (CFG) for a function. Higher-level control flow structures such as loops are implemented with conditional backward edges in the CFG.

3.2.1.2 Static Single Assignment Form

Many traditional compiler analyses and optimisations are far harder to express in the presence of mutable state; establishing precisely which assignments to a variable are reaching definitions requires an iterative data-flow equation to be solved for that variable. If, however, variables are assigned to exactly once, then use-def analyses can be read straight from the structure of the program.

To address this issue, compilers commonly represent code in *static single assignment* (SSA) form, where this property is enforced for every variable. Translating a program with mutable variables to one in SSA form is mechanical; every reassignment to a variable is converted to a freshly named assignment. For example, in Figure 3.1, the variable `x` in the original C source code is reassigned if the branch is taken; in SSA form (shown in hypothetical C-like syntax to the right), a new variable `x1` is created instead.

When control flow join points are reached in SSA programs (e.g. `return x` in the C code in Figure 3.1), multiple versions of a variable may reach the join point. It is a *run-time* property which of these versions is used; `x2` in the SSA-form C program can take either `x0` or `x1` depending on whether or not the branch was taken. The typical expression of such join points in SSA programs is as a ϕ (`phi`) node, where the reaching values are encoded explicitly.[2] In the LLVM code at the bottom of Figure 3.1, the value of `%x.2` depends on the previous control flow through the function.

3.2.1.3 Syntax and Semantics

Instructions in LLVM IR can produce SSA **values** when executed. For example, in Figure 3.1, the result of the instruction `mul i32 %x.0, -1` produces the new value `%x.1`. Instructions only produce single values, and each value is produced only once (this is precisely a restatement of the SSA property).

In textual form, LLVM IR resembles a traditional 3-address code; opcodes (e.g. `mul` or `icmp slt`) are followed by comma-separated lists of arguments. These arguments can be explicitly-typed references to other values (function-local values are written as `%name`, and global ones as `@name`), or literal constants.

Control-flow labels are written as `name:` and must be positioned at the start of a basic block.[3] ϕ nodes take a list of *pairs* of values (e.g. `[%x.0,%entry]` in Figure 3.1) as arguments; each pair associates an immediate predecessor basic block with a value to be selected if control flow passes through that block to reach the ϕ node.

LLVM IR programs can be compiled (ahead-of-time or just-in-time), or interpreted using an abstract model of an infinite-register machine. For the purposes of this thesis, no "advanced" features of the language or abstract machine are relied on; the semantics of any LLVM examples shown should be interpreted intuitively as pseudo-assembly.

3.2.1.4 Type System

Programs in LLVM IR are strongly, statically typed. The type of every value is known statically during compilation, and instructions can only accept correctly-typed arguments. While *opcodes* are overloaded in textual form (`mul i32` and `mul i8` use the same syntax), individual instructions are not polymorphic; a 32-bit integer can only be added to another of the same size, and cannot be added to an 8- or 64-bit one.

Transparent aggregate (structure) types can be defined and referenced in a program as if they were built-in LLVM types. Additionally, SIMD[4] vector types can be constructed, as can statically-sized dependent array types (e.g. `[32 x i8]`).

Pointers analogous to C pointers exist in LLVM IR, with the key difference that the computation of offsets (pointer arithmetic) and dereferences are implemented using

[2] Lowering ϕ nodes during code generation is a separate question to encoding them in the IR, and is not relevant to the discussion in this section.

[3] A block with a label at an intermediate instruction (such that control would implicitly fall through) can instead be written without loss of generality as two blocks with an unconditional branch between them.

[4] Single Instruction, Multiple Data

two distinct instructions (`getelementptr` or GEP, and `load`).

While the type system is strongly typed, some escape hatches exist in the form of bit-level cast functions and explicit arithmetic conversions. For example, the underlying bits of an 32-bit integer could be explicitly reinterpreted as a packed vector of four 8-bit integers. Similarly, a 32-bit float can be sign-extended to a 64-bit double-precision value. These operations, importantly, are all well-defined within the type system.

3.2.2 Summary

The Clang/LLVM toolchain represented a substantial change in perspective among compiler developers and researchers. By exposing a well-defined and documented intermediate representation, tools that extend or modify the behaviour of the compiler are made far easier to develop. This section has given a brief summary of the important design features of LLVM's intermediate representation, which is referenced throughout the remainder of this thesis.

3.3 Recognizing Programs with CAnDL

The CAnDL domain-specific language [12] is designed to describe *patterns* in LLVM IR programs; for example, the natural-language specification "an `add i32` instruction where both operands are the same value" can be expressed precisely in CAnDL. From these descriptions, an SMT[5]-like constraint solver can be used to efficiently search for regions of code that satisfy the pattern.

This section provides a brief introduction to the relevant features and usage of CAnDL as it is used in this thesis, and presents in detail a technique to *generalise* a set of similar LLVM IR programs into a CAnDL program that describes their common features.

3.3.1 Introduction

In its original presentation, Ginsbach et al. [12] motivate CAnDL as a tool for simplifying traditional detect-and-replace compiler analyses. For example, they suggest a floating point optimisation based on the following equality:

$$\sqrt{x * x} = |x| \tag{3.1}$$

That is, calls to the `sqrt` function where the argument is a value multiplied by itself could validly be replaced with a single call to the `abs` function. Detecting this pattern

[5] Satisfiability Modulo Theories; an extension of boolean SAT-solving techniques to a broader set of theories, such as integer arithmetic or bounded arrays.

```
Constraint SqrtOfSquare (
    opcode{sqrt_call} = call
  ∧ {sqrt_call}.args[0] = {sqrt_fn}
  ∧ function_name{sqrt_fn} = sqrt
  ∧ {sqrt_call}.args[1] = {square}
  ∧ opcode{square} = fmul
  ∧ {square}.args[0] = {a}
  ∧ {square}.args[1] = {a}
) End
```

```
{
  "sqrt_call": "%8",
  "square": "%5",
  "a": "%0"
}
```

Figure 3.2: CAnDL constraint program that discovers code implementing the pattern $\sqrt{a * a}$. Informally, the program can be read as "find `call` instructions where the callee is `@sqrt`, and the single argument is the result of an `fmul` instruction with the same value as both its arguments". CAnDL compiles this constraint to an efficient backtracking search procedure. The right hand side shows an example of a CAnDL search result.

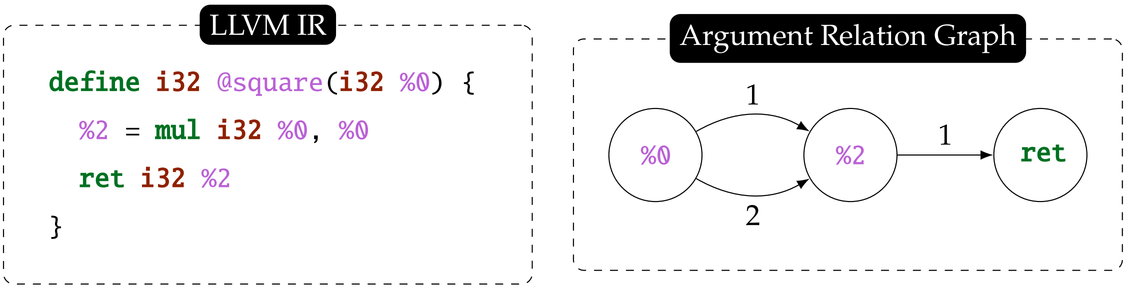
using handwritten compiler analyses is unwieldy; searching for satisfying values requires multiple nested predicates and backtracking. CAnDL addresses this problem by providing a high-level DSL, in which patterns can be described and searched for with a common search procedure.

Ginsbach et al. [12] provide the CAnDL program in Figure 3.2 to search for instances of this pattern in application code. When a successful match is found, variables in the constraint program (e.g. `{sqrt_call}`) are bound to concrete values in the LLVM IR program. Each such key-value mapping describes a single instance of the underlying CAnDL pattern in the code being searched.

For the purposes of this thesis, the precise mechanism by which CAnDL performs its search is not important to discuss in detail; its implementation is treated as a black box that produces matching instances (key-value mappings from variable names to IR values) when given a CAnDL constraint and a program in LLVM IR.

3.3.2 Generating CAnDL

In their original presentation, Ginsbach et al. [12] demonstrate a library of handwritten CAnDL constraints that describe two types of compiler optimisation (traditional peephole optimisations, and larger optimisations for graphics shader code). Doing so



LLVM IR

```
define i32 @square(i32 %0) {
  %2 = mul i32 %0, %0
  ret i32 %2
}
```

Figure 3.3: An LLVM IR program together with a multigraph abstracting the argument relation over its IR values. Function parameters and returns are sources and sinks respectively in the graph.

requires a programmer to implement a constraint program for every new pattern of interest. This thesis instead aims to *automatically* generate CAnDL descriptions from an existing LLVM IR fragment, such that additional instances similar to it can be identified in application code.

First, a graph-based abstraction of LLVM IR is defined. This representation permits a mechanical conversion from LLVM IR programs into CAnDL constraints. However, the constraints generated by doing so are too specific (only matching precisely the code from which they were originally generated). To address this issue, a genetic algorithm that produces *approximate* matches between abstract IR graphs is introduced. This algorithm allows for the common features between multiple similar or equivalent implementations to be extracted.

By combining several synthesis results for the same problem (each of which has slightly different structure) with graph-based approximation, a CAnDL constraint can be obtained that includes the most important features of a computation. The remainder of this section describes this technique in detail.

3.3.2.1 Graph Abstraction

The first step in generating CAnDL from an LLVM IR program is to define a graph abstraction of that program; the CAnDL solving procedure is stated in its original form over similar graphs. For a program P, define a graph G with vertices V and (labelled) edges E as follows:

$$\begin{aligned} V &\triangleq \{\text{all LLVM IR values in } P\} \\ E &\triangleq \{(n,x,y) \in \mathbb{N} \times V \times V \mid x \text{ is the } n^{\text{th}} \text{ argument of } y\} \\ G &\triangleq (V,E) \end{aligned} \tag{3.2}$$

As shorthand, define notation for edges in this multigraph structure:

$$x \xrightarrow{n} y \triangleq (n, x, y) \in E \tag{3.3}$$

Intuitively, this structure captures the data flow in the program by encoding the argument relation between IR values. While CAnDL implements language constructs to constrain control flow relationships, this encoding does not. Figure 3.3 shows a small LLVM IR function together with the corresponding graph encoding; the value `%0` is connected to the value `%1` with two edges to show that it is used as *both* instruction arguments.

In the graph encoding, all appropriate metadata is retained for each LLVM value (e.g. the graph contains the information that `%2` is a `mul` instruction). This permits a mechanical conversion from the graph structure to a textual CAnDL program: first, the graph's vertices are walked, and an opcode constraint is emitted for each one. Then, the edges are traversed and the argument relation over the vertices is emitted. One constraint is emitted for each vertex and edge in the graph.

Doing so produces a precise (but insensitive) set of CAnDL constraints that detect exact matches of the original program. However, to capture a wider class of possible programs that are functionally equivalent, the constraints must be relaxed.

3.3.2.2 Matching LLVM IR Fragments Together

To generate a constraint description that captures the common structure shared by a set of functionally equivalent LLVM IR programs, their respective graph representations must be matched together. Because the programs are functionally equivalent (i.e. they share the same behaviour, but may have different structure), regions of their graph representations that share similar structures are likely to represent the core algorithmic intent of the functions. Conversely, graph regions with significantly different structure are likely to be unrelated to the algorithmic intent.

First, a combined graph is created from a set of LLVM IR programs by directly combining their vertices and edges; by construction the graph regions corresponding to different functions are totally disjoint (there are no edges between vertices that belong to different functions). Next, an equivalence relation $\sim$ over the vertices in this combined graph is defined; this relation partitions the vertices into a set of equivalence classes $V/\sim$. Ideally, vertices corresponding to the same behaviour across the set of programs will be grouped into the same equivalence class.

The usual notation ($\bar{u}$) is used for the set of vertices equivalent to u:

$$\bar{u} \triangleq \{v \in V \mid u \sim v\} \tag{3.4}$$

Then, write $\bar{x} \xrightarrow{n} \bar{y}$ when the following conditions are both satisfied on the combined graph:

$$\begin{aligned} &\forall a \in \bar{x} \,.\, \exists b \in \bar{y} \,.\, a \xrightarrow{n} b \\ &\forall b \in \bar{x} \,.\, \exists a \in \bar{y} \,.\, a \xrightarrow{n} b \end{aligned} \tag{3.5}$$

These properties ensure that each instruction in class $\bar{x}$ is the nth argument of an instruction in $\bar{y}$, and for each instruction in $\bar{y}$, the nth argument is in $\bar{x}$.

3.3.2.2.1 Deriving An Optimisation Target To identify the "most equivalent" regions of the graphs that have been combined (i.e. those regions with the greatest conserved substructure), a metric to rank candidate equivalence relations must be defined. This metric enforces several intuitive properties of the "best" relations. Firstly, instructions that share an equivalence class should have the same opcode:

$$a \sim b \implies op(a) = op(b) \tag{3.6}$$

Secondly, the argument relationship should be preserved when vertices are partitioned into equivalence classes:

$$x \xrightarrow{n} y \implies \bar{x} \xrightarrow{n} \bar{y} \tag{3.7}$$

Thirdly, the argument relation should not be transitively collapsed when nodes are grouped into equivalence classes:

$$\begin{aligned} a \xrightarrow{n} b &\implies \bar{a} \neq \bar{b} \\ a \xrightarrow{n} c \wedge b \xrightarrow{m} c \wedge \bar{a} = \bar{b} &\implies a = b \vee n = m \end{aligned} \tag{3.8}$$

Of course, fulfilling all three criteria precisely is only possible when all the relevant graphs are in fact identical. Instead, an approximate result is the best possible outcome. To rank these approximations using the properties defined in Equations (3.6) to (3.8), a metric m is defined that measures how strictly the properties are adhered to (and therefore punishes deviations from them):

$$m(\sim) = (p_1 \cdot \left|V/\!\sim\right| \tag{3.9}$$

$$+p_2 \cdot \left|\{v \in V/\!\sim \mid \exists x, y \in \bar{v} \colon op(x) \neq op(y)\}\right| \tag{3.10}$$

$$+p_3 \cdot \left|\{\bar{v} \in V/\!\sim \mid \exists y \in \bar{v}, x, n \colon x \xrightarrow{n} y \wedge \neg\bar{x} \xrightarrow{n} \bar{y}))\}\right| \tag{3.11}$$

$$+p_4 \cdot \left|\{\bar{v} \in V/\!\sim \mid \text{third conditions not satisfied}\}\right|)^{-p_5} \tag{3.12}$$

The first parameter controls how much the equivalence relation is encouraged to merge vertices together into larger equivalence classes. Then, the next three parameters control how strictly each property is enforced. The fifth parameter is simply used to change the distribution of the resulting scores, without changing their relative order. By trial and error, the following values were assigned to each of the parameters: $p_1 = 1.0$, $p_2 = 0.5$, $p_3 = 0.5$, $p_4 = 0.5$.

Once this metric is defined, the best possible matching between constraint graphs can be obtained by maximising the value of m. To do so, a simple genetic algorithm is used; from the initial combined graph (i.e. the trivial partition), mutations are applied that alter the partition structure. At a high level, three possible mutations can be applied. First, equivalence classes can be **separated** into their constituent parts, asserting that the nodes in that class are no longer equivalent. Conversely, two classes can be **merged** into a larger class to assert the equivalence of the nodes in those classes. Finally, to establish the structure of the argument relation, nodes with incoming edges to an equivalence class can be **fixed** together into a class (that is, if nodes a and b satisfy $a \xrightarrow{n} x_0 \wedge b \xrightarrow{n} x_1$ for some n, x_0, x_1 with $x_0 \sim x_1$, they are moved to a new class such that $a \sim b$).

These possible mutations are applied iteratively from the trivial partition; at each step a new population of partitions is produced. This population is ranked according to how well each of its elements satisfies m. Then, new elements are selected randomly in proportion to their scores. Finally, the best-scoring element of the final population is returned as the "correct" partition.

3.3.2.3 Generating Constraints from a Matching

To emit a constraint program from the generated equivalence relation, it must be mapped back onto a graph with the correct structure. To do so, a new graph $G/\sim$ is defined as:

$$G/\sim \triangleq (V/\sim, E/\sim) \tag{3.13}$$

That is, the new graph has vertices corresponding to equivalence classes in the highest-scoring partition. The edges of this graph are defined by the following property:

$$(n, \bar{a}, \bar{b}) \in E/\sim \text{ iff } \bar{a} \xrightarrow{n} \bar{b} \tag{3.14}$$

Intuitively, this new graph encodes the argument relation between equivalence classes; if there are consistently numbered edges between the elements of two equivalence classes, then there is an edge between those classes in the new graph.

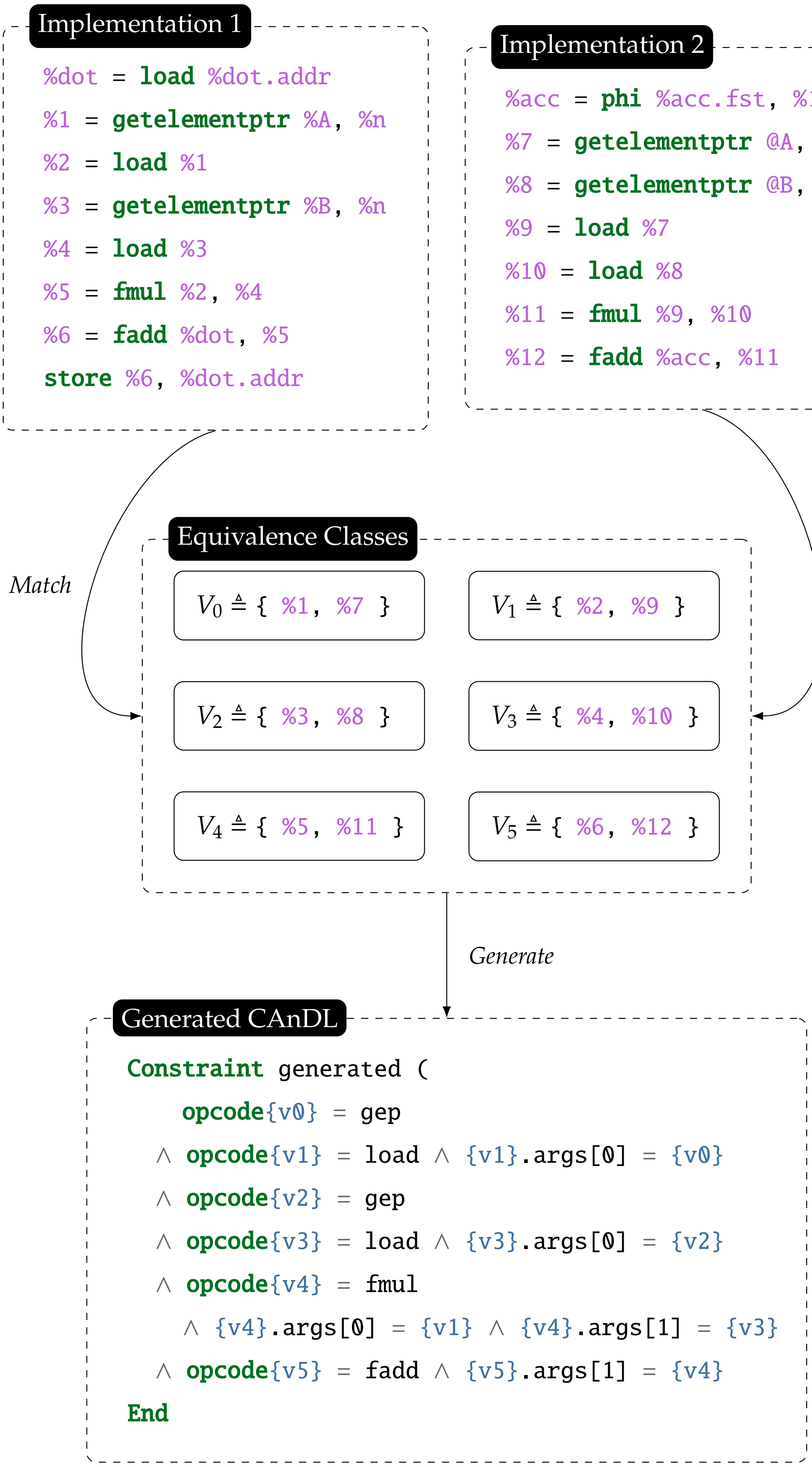
Finally, this graph must be processed further to ensure that it adequately captures the shared structure of the individual graphs used to generate it. To do so, a threshold *d* is defined, and all equivalence classes with fewer than *d* elements are removed. As used in this thesis, *d* is defined as the number of functions used to generate the original graph. In effect, this removes nodes for which no equivalent behaviour in other graphs could be identified, leaving the essential structure of the shared algorithmic skeleton intact.

After the thresholding operation has been applied, the mechanical constraint emission process can be run on the reduced graph to produce CAnDL constraints as before.

3.3.2.3.1 Example Consider the example in Figure 3.4. At the top of the figure, two simplified[6] pieces of LLVM IR code from different loop bodies are shown. The first is from the body of a loop that computes the dot product of two vectors given as pointers, while the second is from the innermost loop of a naive matrix multiplication kernel. They are successfully matched together with the graph matching algorithm introduced previously, producing a set of equivalence classes (shown in the middle of the figure after discarding those with fewer than two elements).

Finally, constraints are generated mechanically. Features that were specific to one program are discarded; notably, the instructions `%dot`, `%acc`, and the store instruction as seen in Figure 3.4. These constraints can now be used to find equivalent code sections (i.e. dot products) in other application source code [112].

[6] Control flow, surrounding context and types are omitted, and some syntax is shortened for the purposes of the figure.



```
%dot = load %dot.addr
%1 = getelementptr %A, %n
%2 = load %1
%3 = getelementptr %B, %n
%4 = load %3
%5 = fmul %2, %4
%6 = fadd %dot, %5
store %6, %dot.addr
```

```
%acc = phi %acc.fst, %11
%7 = getelementptr @A, %k, %i
%8 = getelementptr @B, %j, %k
%9 = load %7
%10 = load %8
%11 = fmul %9, %10
%12 = fadd %acc, %11
```

$V_0 \triangleq$ { %1, %7 }

$V_1 \triangleq$ { %2, %9 }

$V_2 \triangleq$ { %3, %8 }

$V_3 \triangleq$ { %4, %10 }

$V_4 \triangleq$ { %5, %11 }

$V_5 \triangleq$ { %6, %12 }

```
Constraint generated (
    opcode{v0} = gep
  ∧ opcode{v1} = load ∧ {v1}.args[0] = {v0}
  ∧ opcode{v2} = gep
  ∧ opcode{v3} = load ∧ {v3}.args[0] = {v2}
  ∧ opcode{v4} = fmul
    ∧ {v4}.args[0] = {v1} ∧ {v4}.args[1] = {v3}
  ∧ opcode{v5} = fadd ∧ {v5}.args[1] = {v4}
End
```

Figure 3.4: Example showing how general CAnDL constraints can be generated from multiple similar LLVM IR functions. The top two boxes show abbreviated LLVM IR code for two implementations of a dot-product function. These implementations are matched together to produce the six equivalence classes in the center of the figure; all other values form single-element equivalence classes and are discarded. From these classes, the CAnDL constraints at the bottom of the figure can be generated.

3.4 Program Synthesis

Removing (or decreasing) the need for human programmers by allowing computers to write their own programs has been an explicit goal of computer science researchers as far back as the 1960s [116]. If a program can somehow be automatically constructed to meet a specification, then there should be no bugs in that program, and no need for it to be developed by a human programmer. Of course, the problem statement drastically understates the difficulty of doing so: even in extremely limited contexts, the number of potential programs is infinite, and identifying correct examples from that space is computationally intractable.

Because of this difficulty, synthesis research typically entails the identification of a simplifying assumption or contextualisation of a problem; if some aspect of a problem statement can be leveraged to reduce the size of the underlying search space, then synthesis can often become a tractable problem. Doing so for larger and more complex problems, while using fewer resources (e.g. time, memory, or human user intervention) and being more general in expression is the core of modern synthesis research.

This section first lists the terminology common to synthesis research that the remainder of this thesis assumes a working knowledge of. Then, a brief overview of the primary techniques and applications of synthesis research is given, with particular emphasis on how these approaches scale to attack more complex problems.

3.4.1 Terminology

Program synthesis is a long-standing, well-studied field in the literature, and so a rich vocabulary of terms pertaining to synthesis has evolved over time. Occasionally, these terms become overloaded across publications and implementations; for the sake of disambiguation this section gives a high-level description of program synthesis with the terms used *in this thesis*. Places in the text where a citation's use of vocabulary diverges from this will be identified explicitly, but discussed in the language of this thesis.

Program In the context of synthesis, the definition of a program can be interpreted very broadly. While much of the literature deals with general-purpose, Turing-complete languages, other work often deals with domain-specific languages that are designed to solve specific classes of problem (such as network protocol design [117], identifying solutions to logic puzzles [54], or shared-memory consistency models [53]). Generally, the only common definition is the association of a semantics with syntax trees over a language.

Specification Synthesis relies on being able to determine whether or not a program is *correct* or not. The set of rules by which this correctness can be checked are a *specification*. Similarly to the language in which programs are expressed, many different types of specification are possible. In some cases, SAT solvers or formal proofs can be used to demonstrate that a program definitely satisfies a logical assertion, while in other languages the best possible approach is to interpret a program and examine its behaviour against a set of test cases. Specifications typically involve some degree of meta-theory in their statement (e.g. if a problem specification were to require the synthesis of an integer addition function, the specification meta-theory might involve a definition of $+ : \mathbb{N} \to \mathbb{N}$).

Hint / Help / Information For many synthesis problems, it is possible for the human user interacting with the synthesiser to provide additional information that guides the synthesiser's search process. This information typically does not fully specify whether a solution is correct, but constrains the likely *structure* of a correct solution. For example, when synthesising programs in a functional language, a hint might state that particular functions are likely to appear somewhere in a correct solution.

Problem & Solution The task of transforming a *particular* problem specification into a program is a synthesis *problem*; a correct program produced by doing so is a *solution*. Evaluating the success of a synthesiser typically entails attempting a known set of problems and recording whether or not a solution could be found for each one.

Search Space The set of all possible programs expressible in a language (modulo any problem-specific constraints) is the search space over which a synthesiser operates. Typically, the size of this set is very large and so well-chosen heuristics or search algorithms are required to identify correct solutions from it.

Oracle For some classes of synthesis problem, it may not be possible to fully elaborate their specification before synthesis begins (for example, if there are an infinite number of rules a program may or may not satisfy). In these cases, the problem statement often provides for an *oracle*, an abstract object that responds to queries by providing further details of the relevant specification. For example, an oracle might provide ground truth input-output pairs a solution must satisfy, or could provide a falsifying counterexample if shown a potential solution. The taxonomy of potential oracles is broad, and provides for the statement of many different types of problem [50, 118].

Reference & Target In some synthesis problems, the specification states that a solution should re-implement precisely the behaviour of some other system. The system whose behaviour should be re-implemented is referred to as a *reference* implementation or *target* for synthesis.

3.4.2 Overview of Synthesis

Chapters 4 to 7 describe three approaches to program synthesis, that share several common design choices. This section gives a brief overview of the contextual factors that lead to these design choices, and a comparison to the techniques associated with the counterfactual choices.

Broadly, program synthesis techniques can be divided into two categories: *inductive* or *deductive*. Under deductive synthesis, the specification for a program can be transformed by a sequence of deterministic steps into precisely that program. Perhaps the most widely used example of deductive synthesis in practice is in mechanised formal verification, where certain logical statements can be proved automatically. While some deductive synthesis contexts entail intractably large search trees, all can be solved through the application of rules. This is not the case for inductive synthesis, where a solution program must be *induced* from scratch to satisfy a specification. There are no specific deduction rules available, and the exact mechanism by which a solution is constructed may not be deterministic. Inductive synthesis is the most common style of synthesis: typically, real-world scenarios do not admit clear derivations. The target functions examined in this thesis are no exception, and so this thesis deals entirely with inductive synthesis.

Within inductive synthesis, a subdivision can be drawn between problems specified entirely by a list of examples (programming by example, or PBE) and all other problems with more universal specifications (generally, that a property holds for *all* solutions, rather than pointwise specification via examples). Typically, PBE problems arise where the goal of synthesis is to match user intent through handwritten examples, or to mimic the behaviour of some existing system of unknown construction. In scenarios where the aim of synthesis is to have solutions satisfy abstract properties, the specification usually admits stronger verification procedures. In this thesis, the PBE scenario arises naturally for two reasons: the shared aim of the work in Chapters 4 to 7 is to match the behaviour of existing components, and the synthesised functions that do so do not admit formal verification techniques (that is, verifying abstract properties would be difficult even if those properties could be stated).

Now considering only PBE synthesis problems, a further distinction exists between systems that exploit the *structure* of the examples they are given (for example, by expecting a human user to supply meaningful examples, or by having maximal counterexamples be generated by an oracle of some kind), and those where examples are cheap but have no inherent structure. This thesis falls squarely into the second division; in mimicking the behaviour of existing components, many examples can be generated, but with no particular structure.

Finally, in some synthesis contexts the *correctness* specification is not the only available specification: there may be other constraints on a solution's structure that do not uniquely specify a solution on their own, but can be used to accelerate synthesis somehow. For each of the approaches in Chapters 4 to 7, this is the case. This last decision neatly summarises the flavour of synthesis described in this thesis: inductive synthesis, uniquely specified only by a set of cheaply-obtainable examples, but with rich non-correctness specifications available.

3.5 A Sketch Language

Perhaps the most common type of hint given to program synthesisers is a *sketch*: a high-level, partial program structure to which low-level details can be added to produce a working program. This idea is language-agnostic, and can be instantiated in different ways depending on the output language of a particular synthesiser. This section introduces a sketching framework shared between the synthesis approaches in Chapters 4 to 7, and is structured as follows. First, a brief overview of general sketch-based synthesis is given. Then, the design of a flexible, sketch-enabled language for synthesising imperative programs is presented. Finally, the compilation of this language to LLVM IR (to enable integration with compiler tooling as suggested in Section 3.2) is demonstrated, along with a high-level overview of how search-based synthesis can be used to instantiate correct programs from sketches.

3.5.1 Sketching

In sketching program synthesis, the synthesiser begins with a partial structure for solutions; by doing so, high-level insights from the programmer or user can be combined with low-level implementation details better suited to an automated procedure [60, 61]. An example of this is shown for a C-like language in Figure 3.5, where incomplete details of a program's low-level operation are filled in by a synthesis procedure.

```
1 int sketch(
2   int n, int *xs, int *ys)
3 {
4   int r = ??;
5   while (??) {
6     {?}
7   }
8   return r;
9 }
```

```
1 int solution(
2   int n, int *xs, int *ys)
3 {
4   int r = 0;
5   while(--n > 0) {
6     r = r + xs[n] + ys[n];
7   }
8   return r;
9 }
```

Figure 3.5: A sketch (left) and potential solution (right) to a synthesis problem with the informal specification "sum all the elements in two arrays with the same length". In the sketch, only high-level details are provided: the presence of a loop of some kind, and a return value of the correct type that is initialised at the top of the function. All other elements of the program are left incomplete (as `??` markers). When instantiated to a solution, lower-level implementation details such as the loop condition and initial value of `r` are filled in.

3.5.1.1 Holes

In Figure 3.5, the most important feature of sketching is demonstrated: the presence of *holes* in the partial program. Holes are sub-trees of a program's abstract syntax tree (AST) with no concrete value, such that replacing a hole with a valid AST fragment produces a valid program; another way to frame the abstraction is that a program with holes represents the set of *all possible* programs sharing the same concrete AST section as the sketch. For example, in Figure 3.5, instantiating the hole on line 4 as `int r = n * n;` would also produce a program belonging to the same set as the solution on the right-hand side (albeit one not meeting the original specification).

Holes can be treated differently depending on the underlying host language to which they have been added, and the requirements of their application. In some cases, it is possible to create a well-defined semantics by which programs with holes may be interpreted (for example, by keeping track of holes as closure environments as they are executed as far as possible [119]). Other applications only require that programs with holes can be manipulated while preserving the validity of the AST (e.g. in Figure 3.5, the hole on line 4 should not become `int r = float;`).

3.5.1.2 Two-Phase Synthesis

Originally, sketches were presented as a way for the programmer to express their high-level knowledge of a potential solution. However, as the methodology evolved, synthesisers began to demonstrate *two-phase* techniques whereby likely program sketches are identified automatically by an initial synthesis phase, then instantiated as they would be in a traditional method. For example, Wang et al. [62] search for abstract SQL queries (sketches) that would over-approximate the results for a specification (i.e. return a superset of the required output), then filter these sketches by constraining the predicates within them.

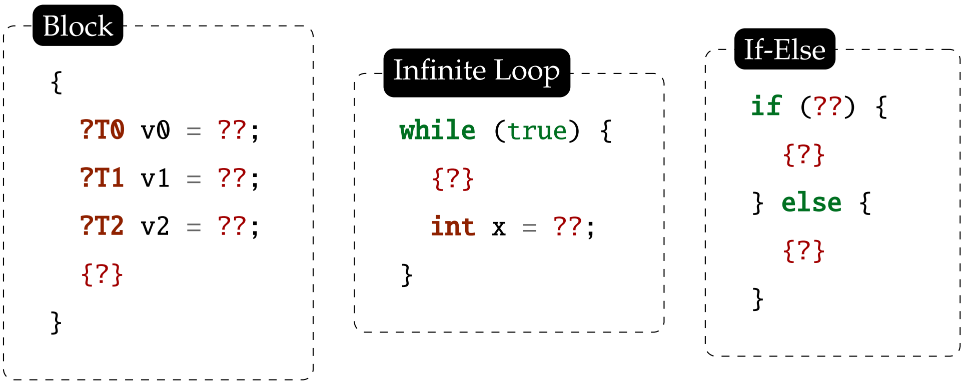
This style of two-phase synthesis encodes two key ideas: firstly, that some approximation to a hypothetical user's intent can often be identified through search, and secondly, that splitting synthesis into multiple searches at different levels of abstraction can reduce the size of the relevant search space. Returning to Scythe [62], searching first for query over-approximations, then for predicates produces an exponential reduction in the size of the total search space.

As a rough argument for why this is the case, consider a language with C AST constructors (node types), and a program sketch with N holes to be instantiated. There are $O(C^N)$ possible programs that could result from this instantiation. Now, the search process could be split into two parts: first, a search for sketches that permits $c \ll C$ AST constructors at each of the N holes, producing p sketches with $n \ll N$ holes, and secondly, a search through those sketches for correct programs as before. The size of the resulting search space is $O(c^N + pC^n)$, which is $\in O(C^N)$.

Reducing search spaces through this type of two-phase method does require a suitable method to search for sketches (e.g. rejecting provably invalid sketches, or using user information as an incomplete prior on what sketches should be considered). The remainder of this section describes the implementation of a sketch language designed to support general two-phase synthesis methods, as well as to integrate well with compiler tooling as suggested in Section 3.2.

3.5.2 Fragments

In many two-phase sketching synthesisers, sketches are designed to be built from combinations of smaller atomic elements. For example, the DeepCoder system [79] identifies sketches (roughly) as compositions of individual functions and combinators; similarly, Scythe composes well-formed SQL query elements into abstract queries. Composing sketches from smaller parts in this way means that fine-grained analyses can be made of the individual components, an easier problem than attempting to



```
{
  ?T0 v0 = ??;
  ?T1 v1 = ??;
  ?T2 v2 = ??;
  {?}
}
```

```
while (true) {
  {?}
  int x = ??;
}
```

```
if (??) {
  {?}
} else {
  {?}
}
```

Figure 3.6: Three examples of simple imperative program fragments, given in the syntax of a C-like pseudocode. In this syntax, `{?}` represents a fragment hole (a point at which another fragment could be nested), while `??` represents a computation hole (a point at which a value should be chosen to produce a concrete program). Types for computation holes may either be fixed (`int`) or left unknown until a later stage of synthesis (`?T0` , `?T1` , etc.).

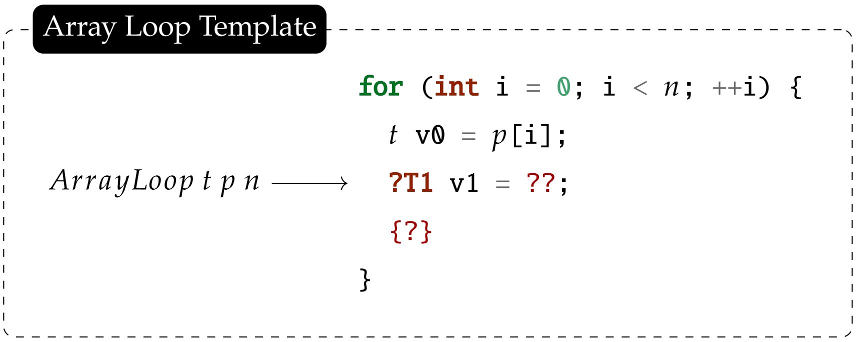
evaluate entire sketches (e.g. DeepCoder evaluates whether or not individual functions such as `map` are present in a potential solution).

Often, synthesisers for imperative languages such as Simpl [70] instead treat sketches as indivisible skeletons into which computations can be inserted. To support these twin goals (two-phase synthesis and eventual support for imperative languages to facilitate compiler integrations), a system for describing atomic, composable components of imperative programs is required. This thesis defines a language of *fragments* to implement sketch-based synthesis of imperative programs.

3.5.2.1 Definitions

At a high level, a fragment is simply a region of imperative code that can contain two kinds of hole (*fragment* and *computation*, respectively). Distinguishing two kinds of hole is a convenient mechanism to enable two-phase synthesis. Only another fragment can be used to instantiate a fragment hole, which preserves computation holes to be instantiated by a subsequent synthesis mechanism that takes a sketch as input (see Figure 3.6). For clarity, fragments are listed as C-like pseudocode when they are presented in this thesis; using this notation, Figure 3.6 shows some possible examples of fragments.

The fragments in Figure 3.6 have no free variables (that is, when written as code they do not reference any variables not defined in the fragment itself). This means that while they can always be combined with other fragments while avoiding syntactic



```
for (int i = 0; i < n; ++i) {
  t v0 = p[i];
  ?T1 v1 = ??;
  {?}
}
```

Figure 3.7: Example of a fragment template definition. Parameters passed to the template function are inserted into the fragment body at the appropriate places (distinguished here in *italic* type). The resulting fragment has a single computation hole of unknown type, as well as a fragment hole.

clashes or ambiguity, they are not modular or usefully reusable. To address this, the remainder of this thesis deals largely with fragment *templates*, defined as arbitrary functions onto the set **Frag** of all possible fragments.

For example, consider the informal fragment description "a loop over the elements of an array with a known size". To properly express this as a fragment, three pieces of information are required: the type of element stored in the array, a pointer to the array, and the size of the array as an integer. The array-loop fragment template can therefore be understood informally as a dependent function (where τ is the set of types understood by the fragment system, and $\star$ is the "pointer to" postfix type constructor):

$$ArrayLoop\colon (t\colon \tau) \to t\star \to \mathbb{N} \to \mathbf{Frag} \tag{3.15}$$

In Figure 3.7, a possible definition for the $ArrayLoop$ fragment template in terms of these parameters is shown. Generally, in the rest of this thesis, fragments and fragment templates are treated equivalently for the sake of clarity; cases where an issue arises from doing so are noted explicitly.

3.5.3 Compilation and Search

Fragments as discussed so far are abstract objects; they can be named and instantiated from a template (e.g. $ArrayLoop$ as shown in Figure 3.7). While it is convenient to show fragments as pseudocode in a hypothetical C-like language, this representation does not actually exist; instead, the semantics of any given fragment are defined by a compilation function that produces a concrete program in an extended dialect of LLVM IR from that fragment.

The compilation function is total (i.e. any fragment can always be compiled to LLVM, with no exceptions). For $\mathbf{L}^+$ the set of programs in the language targeted by compilation:

$$compile\colon \mathbf{Frag} \rightarrow \mathbf{L}^{+} \tag{3.16}$$

Compiling a fragment produces an LLVM IR program, but one with computational holes. The pseudocode shown in Figures 3.6 and 3.7 can be seen as demonstrating *where* these holes will be located in the eventual compiled IR, rather than as holes with specific semantics in the C-like pseudocode language. An extended dialect of LLVM ($\mathbf{L}^{+}$) is required to support these holes. The semantics and implementation of this dialect are given in Section 3.6.1, and are treated at a high level in this section.

Because fragments contain syntactic holes after being compiled, they must then be instantiated with concrete values by another synthesis procedure; the role of the sketch framework ends at compilation. In the remainder of this thesis, the holes in compiled fragments are filled by search-based synthesis procedures, but there is no hard requirement for this to be the case; other mechanisms such as interactive programming [119] or SAT-solving [74] could be used to instantiate holes in different problem contexts.

3.5.4 Compositionality

The ultimate goal of fragments as defined in this section is to be composed together into a larger sketch, but the mechanism by which this occurs has as yet only been alluded to (in the form of *fragment holes* `{?}`). This section explains the process of composition between fragments, and how larger sketches can be constructed by doing so.

Two requirements are placed on the compiled code generated by fragment implementations for them to be valid components of sketches: firstly, their compiled control flow must be single-entry, single-exit. This means that when control *enters* a fragment, it proceeds unambiguously through that fragment until the exit. Secondly, if a fragment contains a fragment *hole*, it must be possible to specify the compilation of that fragment even if the hole is never instantiated (that is, filling a fragment hole is optional).

If both of these requirements are met, then a composition operator $\circ$ can be defined between two fragments:

$$\_ \circ \_ \colon \mathbf{Frag} \rightarrow \mathbf{Frag} \rightarrow \mathbf{Frag} \tag{3.17}$$

The composition operator highlights a key point of the fragment-based sketching process: the composition of two fragments is itself a fragment, and so must be subject to the same requirements and operations as the fragments that it was created from.

Because of this, arbitrary compositions of fragments can be created and compiled to LLVM IR by consumers of the sketching framework. No details about the specific types of fragment involved are necessary, allowing the consumers to be fully abstract when constructing sketches.

Individual fragment semantics are fully specified (for a fragment f) by providing definitions for the following operations:

$$\begin{aligned} \mathit{compile}\ f &: \mathbf{L}^+ \\ f \circ \_ &: \mathbf{Frag} \longrightarrow \mathbf{Frag} \end{aligned} \tag{3.18}$$

3.5.4.1 Example

As an example, first consider a fragment *skip* : **Frag**. Intuitively, this fragment is the "do nothing" unit element for control flow. To give it a useful semantics, definitions for $\mathit{compile}$ and $\mathit{compose}$ must be given.[7] A possible definition for *skip* is:

$$\begin{aligned} \mathit{compile}(\mathit{skip}) &\triangleq \texttt{\{\}} \\ \mathit{skip} \circ f' &\triangleq f' \end{aligned} \tag{3.19}$$

This is intuitive: compiling skip produces an empty statement that has no effect when executed, and composing it with any other fragment f' yields f'. As a more complex example, consider a fragment template *fixed-loop* : $\mathbb{N} \longrightarrow$ **Frag**. This template represents a loop with a fixed number of iterations (specified by the template parameter); the body of the loop is established through composition. Variants of this fragment are used in Chapters 4 to 7.

Writing down a definition for $\mathit{fixed\text{-}loop}$ is more difficult than for skip. The intuitive action of composition on such a loop structure is for the body of the loop to be instantiated with the second fragment. However, the result of doing so must also be a fragment. To define this composition, an auxiliary fragment is defined:

$$\begin{aligned} \mathit{compile}(\mathit{fixed\text{-}loop}(n)) &\triangleq \texttt{for(int i=0; i<}n\texttt{; ++i) \{\}} \\ \mathit{fixed\text{-}loop}(n) \circ f &\triangleq \mathit{fixed\text{-}loop}(n)_f \end{aligned} \tag{3.20}$$

This auxiliary fragment defines both its compilation and composition recursively using the child fragment f:

$$\begin{aligned} \mathit{compile}(\mathit{fixed\text{-}loop}(n)_f) &\triangleq \texttt{for(int i=0; i<}n\texttt{; ++i) \{}\mathit{compile}(f)\texttt{\}} \\ \mathit{fixed\text{-}loop}(n)_f \circ f' &\triangleq \mathit{fixed\text{-}loop}(n)_{f \circ f'} \end{aligned} \tag{3.21}$$

Recursive definitions of this sort are common to many fragment definitions, and allow for potentially large trees of fragments to be constructed, while only requiring

[7] For the sake of notational convenience, definitions for $\mathit{compile}$ in this section target C-like pseudocode.

each individual fragment to define its local composition and compilation effects. Section 3.6.2 gives details on how these operations are implemented in practice for real fragment definitions.

3.5.5 Summary

This section has presented an abstract definition for a language of *fragments*, which can be arbitrarily composed to form larger, more complex sketches. By providing definitions for two operations (compilation to a dialect of LLVM IR and composition with another fragment), a range of imperative code structures can be implemented independently of other fragment definitions. Next, in Section 3.6, the implementation details of this scheme are discussed.

Computational holes provide the interface between the sketch language and onward synthesis methods: by specifying points within a fragment's definition where program values should be computed, a compiled sketch can be handed to an external solver or search procedure for instantiation into an executable program.

3.6 Implementation Details

Previously, Section 3.5 described (in an abstract sense) the definition of a generic fragment language that can be used by applications to build program sketches; applications provide their own fragment implementations and take advantage of the genericity of the underlying framework to compose and compile them into sketches.

Compiled fragments and program sketches are programs in an extended dialect of LLVM IR that supports *holes*. By including holes, programs in this dialect can be handed to an external solver or search procedure that is responsible for selecting concrete values for those holes. This provides a clean interface and separation of concerns between different phases in synthesiser implementations; each of the synthesisers in Chapters 4 to 7 relies on this interface between their identification of program sketches and search for candidate solutions.

This section explains the underlying implementation details used throughout the rest of this thesis. First, the extended LLVM IR dialect targeted by fragment compilation is defined, along with the additional operations required to safely manipulate the holes within its programs. Then, the mechanism by which concrete fragments are implemented is defined. Once an external procedure instantiates the holes in a program sketch, it can be executed. The shared infrastructure to execute and test these compiled programs as part of a synthesis workflow is detailed. Finally, an overview of related implementation details and other associated boilerplate code is given.

Known Type

```
%v1 = add i32 %0, 1
%v2 = call i32 @hole.0()
%v3 = add i32 %v1, %v2
```

Unknown Type

```
%v1 = add i32 %0, 1
%v2 = call %hole.t @hole.1()
%v3 = call i32 @add(%v1, %v2)
```

Figure 3.8: Two abbreviated portions of LLVM IR code that contain hole values, encoded as calls to special functions with no definition. On the left, the hole value's type is known ahead of time, while on the right it has unknown type and so is assigned a placeholder type. When operating on values of unknown type, special stand-in functions take the place of LLVM's usual instructions, and the RAUW primitive operation must be extended to handle type assignment.

3.6.1 Augmenting LLVM IR with Holes

Previously, in Section 3.5, a dialect of LLVM IR that supports *holes* was discussed as the target for fragment compilation. In the context of LLVM IR, a hole is an SSA value whose exact run-time value is not yet known. Holes can be referenced as if they were first-class LLVM values by other instructions, but a program containing them cannot be compiled or executed without instantiating each hole with a concrete value.

Figure 3.8 demonstrates the encoding of holes used in this dialect. Each hole is represented by a call to a function with no definition (that is treated as though it might return nondeterministic values); the values produced by these calls can have their types identified ahead of time (e.g. `i32` on the left hand side of the figure), or can have *unknown* types. Leaving hole types unknown is necessary to support fully-generic fragment implementations that can operate data of many different types, and be composed within multiple different parent fragments.

LLVM IR was not designed to support values with unknown types, and so several problems arise when attempting to construct programs that contain holes. The solutions to these problems lead naturally to the encoding of holes demonstrated in Figure 3.8; in the remainder of this section, each of these problems is introduced, and the solution explained.

3.6.1.1 Basic Encoding

To encode and keep track of symbolic placeholders (holes) in an LLVM program, concrete SSA program values need to be created to represent those holes. So that they can be easily distinguished from "normal" values, holes are encoded as calls to functions with no definition (i.e. they only have a declaration), and a distinguished naming convention.

An example of this encoding is shown on the left-hand side of Figure 3.8: the value `%v2` is a hole, and can be identified as such by the target of the LLVM call instruction that it is generated by.

3.6.1.2 Static Typing

The first problem with this encoding is that all values in LLVM IR must be explicitly annotated with their type. However, for holes, their type may not be known ahead of time (so as to allow fragments to implement type-generic abstract operations). Addressing this problem is one of the key design choices when designing languages with support for holes; for example, the Hazel [119] interactive programming environment allows for expressions to be typed with incomplete types that themselves contain holes.

In LLVM, however, explicit type annotations mean that a concrete type must be selected for each value when it is created. To do so, a type-level analogue to the value-level hole encoding is used. A distinguished *hole type* is created, and assigned as the type for any hole whose type is not constrained ahead of time by its parent fragment. Returning to the example in Figure 3.8, the value `%v2` on the right-hand side has type `%hole.t`; a unique, opaque type guaranteed to be unused anywhere else in the program.

A different non-deterministic, definitionless function is created for each unique type of hole in a program. For example, in Figure 3.8, the functions `@hole.0` and `@hole.1` correspond to holes of type `i32` and unknown type (`%hole.t`) respectively.

3.6.1.3 Operations on Values

While encoding unknown types as a single global unique type allows for values of notionally unknown type to be encoded, doing so once again introduces problems to the core LLVM IR language. In LLVM, instructions (such as `add`, `load`, or `fmul`) can only be applied to built-in types; there is no mechanism to add two values of type `%hole.t`, for example.

This means that if holes have unknown type, then instructions that use them as arguments cannot be created. However, fragments may wish to do so. For example, a fragment representing an elementwise in-place mutation of an array might wish to include a `store` instruction in every instantiation. If the value to be stored has unknown type, this is not possible.

To address this issue, an additional set of special LLVM IR functions is defined. Each of these functions represents an individual LLVM opcode, but with unspecified argument types. As an example, compare the values named `%v3` on each side of

Figure 3.8. On the left-hand side, both arguments are of the same type (`i32`), and so an `add` instruction is valid. However, on the right, `%v2` has unknown type, and so a call to the special function `@add` is made instead. Because one argument is of type `i32`, the return type of the call can be inferred. If both arguments were of unknown type, the return type would be as well.

Analogous special functions are defined for all LLVM opcodes, such that any valid program can be lifted into the "unknown type" domain by making calls as in Figure 3.8. The sketching and fragment framework has one final role to play after holes are instantiated by an external procedure: calls to the special opcode functions are type-checked, and resolved by replacing them by actual instructions.

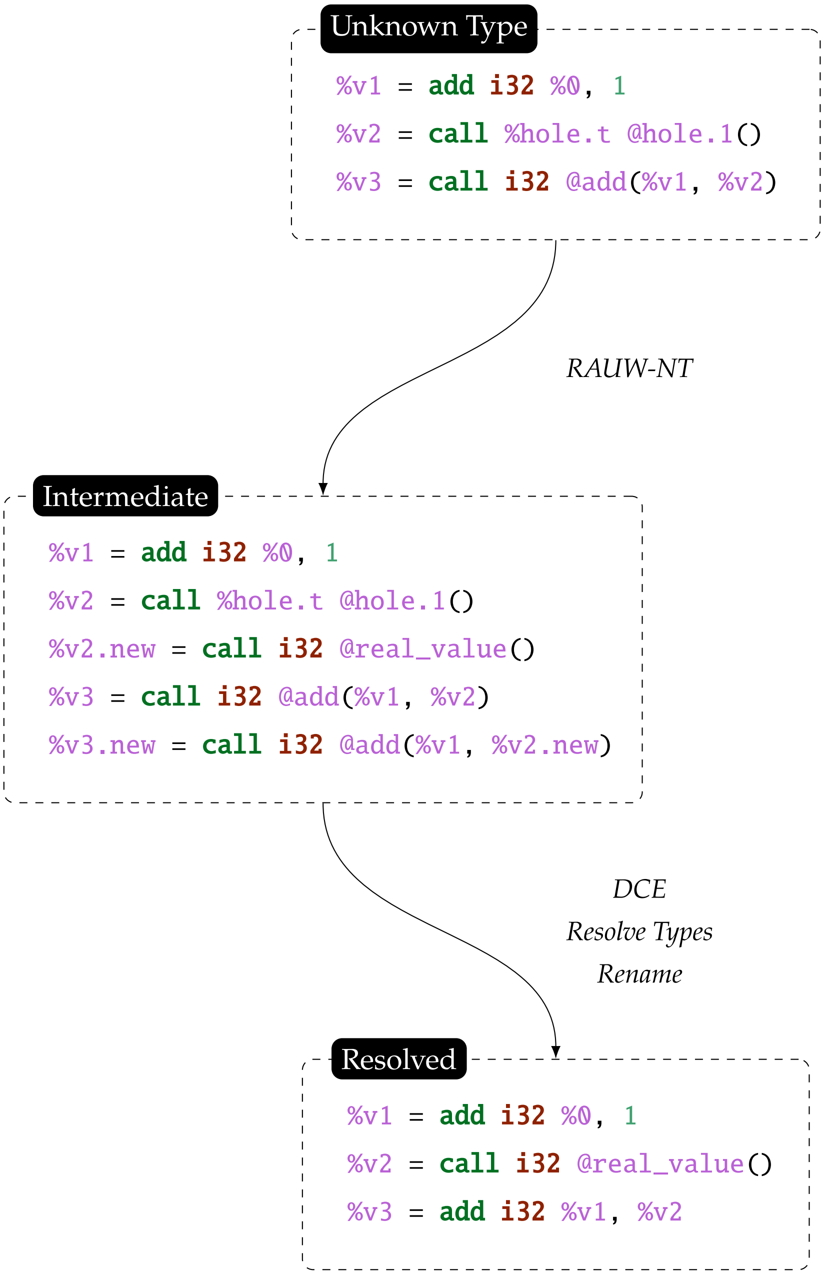
3.6.1.4 Assigning Types

One final problem arises from the encoding of instructions as special functions: from the perspective of the underlying program, some values may appear to *change type* when holes are instantiated. This is a fundamental property of holes as they are designed in the language extension; if a concrete value is assigned to a hole with previously unknown type, that value must have a type, and so the hole's apparent type must somehow change. As discussed previously, LLVM IR is statically typed, and so the type of any value can never actually change. To implement a mechanism to allow holes and special functions to *appear* do so, a new IR manipulation primitive is required.

In traditional LLVM IR, the replace-all-uses-with primitive (RAUW) is one of the fundamental operations for modifying programs. It replaces a value with another value of the same type, ensuring that all subsequent uses of the original value then refer to the new value.

To allow holes to appear to change type, a new primitive operation RAUW-NT (New Type) is introduced. It extends RAUW by adding a new specialisation for holes of unknown type; for these, the new value can have any type. Subsequent uses of the original hole are checked for safety when doing so. For example, in Figure 3.9, the hole value `%v2` is used as an argument to the `@add` function (which returns `i32`). This means that only types that can be safely added to `i32` can be used to instantiate the hole.

Implementing RAUW-NT is a two-phase process, both of which are shown in Figure 3.9. First, new corresponding values are created alongside the original value and all instructions that depend on it (`%v2.new`, `%v3.new`). Then, the entire dependency graph for the old value is removed, and all types checked and special functions resolved.



```
%v1 = add i32 %0, 1
%v2 = call %hole.t @hole.1()
%v3 = call i32 @add(%v1, %v2)
```

```
%v1 = add i32 %0, 1
%v2 = call %hole.t @hole.1()
%v2.new = call i32 @real_value()
%v3 = call i32 @add(%v1, %v2)
%v3.new = call i32 @add(%v1, %v2.new)
```

```
%v1 = add i32 %0, 1
%v2 = call i32 @real_value()
%v3 = add i32 %v1, %v2
```

Figure 3.9: An example of the intermediate step taken when assigning a concrete type to a value of unknown type, using the RAUW-NT IR manipulation extension.

Combining RAUW-NT and special functions that encode instructions can lead to type information being propagated backwards as well as forwards in the program: if one argument to a special function has its type instantiated, then any hole arguments to that function have their eventual type constrained.

Implementing RAUW-NT requires that destructive modifications are made to the program, and so it does not respect IR validity properties in the same way as LLVM's own internal operations do. Despite this, RAUW-NT preserves the validity of the IR across calls (that is, any unsafe operations are kept inside the extension).

3.6.1.5 Summary

This section has demonstrated a modular, reusable abstraction for manipulating LLVM IR programs that contain *holes* of potentially unknown type. Doing so in a type-safe, robust manner requires a number of extensions to be made to the core LLVM libraries. Crucially, the validity of an IR program manipulated by these extensions is preserved between operations; any unsafe or invalid states are not exposed to the user.

While this thesis proposes search-based synthesis from program sketches as the primary use case for these extensions, other applications would be enabled by the addition of holes to LLVM IR. For example, interactive programming or expert-guided micro-optimisations could be implemented by instantiating holes through a different mechanism. Alternatively, a macro-based DSL for C-like languages could be used to provide a higher-level interface to hole-based programming (i.e. a syntax more similar to the examples of pseudocode in this chapter).

3.6.2 Fragment Implementations

In Section 3.5.4, the abstract definition for sketch fragments in terms of their fundamental operations (compilation to LLVM IR with holes, and composition with other fragments). This section explains how these operations are implemented in practice, and how tools can provide new fragment types compatible with the generic framework.

Fragment Implementation The mechanism provided for new fragment types to be implemented and loaded is as C++ classes satisfying a specific virtual interface (listed in Figure 3.10).

Composition The `base_fragment::compose` method specifies the semantics of each individual fragment type under composition. Typically, this entails storing the second fragment as a member variable so that it can be referenced during compilation (in effect, building a tree of child fragments).

```
class base_fragment {
  virtual std::unique_ptr<base_fragment>
    compose(std::unique_ptr<base_fragment> other) const = 0;

  virtual llvm::BasicBlock* compile(
    llvm::BasicBlock* entry) const = 0;
};
```

Figure 3.10: The interface that new fragment implementations should satisfy to be compatible with the sketching framework. Composition creates a new fragment (consuming its argument; there is no sharing within the fragment tree), while compilation accepts a basic block representing the entry point at which to attach the results of compilation (fragments must have single-entry, single-exit control flow).

Compilation Because fragments must have single-entry, single-exit control flow, a fragment hole `{?}` can be viewed logically as the exit of a single basic block. If the hole is instantiated with a child fragment, then its control can exit to the same point. Otherwise, control simply passes through the exit and continues. Compilation can therefore be expressed as a method that inserts additional LLVM IR at the specified point.

All fragment types described throughout the remainder of this thesis are C++ classes implemented in this way. They are able to dynamically build trees of child fragments, and compile to LLVM IR with holes at a specific entry point.

3.6.3 Execution & Testing

As well as the sketching and fragment-handling mechanisms described previously, the general framework shared by Chapters 4 to 7 contains utilities to handle the compilation, execution and testing of sketches whose holes have been instantiated (in effect, general LLVM IR programs). These utilities are largely boilerplate code, with little bearing on the conceptual work carried out in each chapter. A brief summary of them is as follows:

JIT Compilation By compiling instantiated sketches in-process, the overhead of writing code to a file and calling out to an external compiler can be avoided. As well as enabling the analyses and tools below, this allows implementations that consume the general synthesis framework to more rapidly iterate over possible sketches.

Safety & Bounds Checking Fragment implementations can opt in to a static bounds-checking method when generating LLVM IR. If they do so, a shared, global limit on the size of arrays that can be validly accessed by instantiated sketches is set. Accesses outside this limit are safely caught without generating a signal, and reported back to the calling code through an additional return value.

Type Safety When users construct or generate arguments to be passed to an instantiated, compiled sketch, the framework is able to check the type of these arguments to ensure they are compatible with the sketch.

Testing A general test harness is provided to execute compiled sketches with generated test data from the consuming application, observe the results and compare them to a source of ground truth (either another sketch, a dynamically loaded symbol, or a set of examples).

3.7 Summary

This chapter has presented the background technical details on which the main contributions of this thesis are built. Section 3.2 introduced LLVM IR [8] as a way of integrating synthesis and API migration with existing compiler workflows. Then, Section 3.3 presents a previously-published tool for generalisation of constraint patterns for code search (as implemented by Philip Ginsbach, a co-author of Collie et al. [2]). With respect to program synthesis, Section 3.4 surveyed important terminology from the literature to provide context for Sections 3.5 and 3.6, in which a sketch-based synthesis framework was introduced. This framework is referred to throughout the remainder of the thesis.

Chapter 4

Synthesising Performance-Critical Functions with Type Annotations[1]

This chapter develops a program synthesis-based approach to the problems of legacy code rejuvenation and performance portability. By using synthesis to automatically model the behaviour of library functions, existing constraint-based code search tooling can be used to efficiently discover compatible code

Because the interface of the targeted library functions is known, but their implementation is not, a domain-specific annotation and query language is used to formalise interface properties that would be used informally by developers using the library. By examining these properties, synthesis can be efficiently directed by heuristics towards likely program sketches. This allows for the implementations of functions with complex behaviour to be easily synthesised.

Synthesised functions are converted to general constraint descriptions using a novel genetic algorithm. By doing so, boilerplate and irrelevant code can be safely discarded while retaining key algorithmic structure.

An evaluation of this approach is carried out on widely-used scientific computing and machine learning applications. For the most common performance bottleneck functions in these applications, by synthesising implementations and matching them against application code, refactorings can be correctly implemented that produced speedups of up to 10×.

4.1 Introduction

Fast numerical libraries have been a cornerstone of scientific computing for decades [120, 121]. They provide efficient implementations of key algorithmic components and

[1] This chapter is based on published research in Collie and O'Boyle [1], Collie et al. [2]

allow a separation of concerns. This comes at a cost, however, as it may tie programs into vendor-specific software ecosystems and results in non-portable, polluted code. A new library API means that the original "vanilla" code has to be recovered, and then modified to use the new libraries.

The risk of being tied into an out-of-date library API often leads developers to release multiple versions of their code targeted at different underlying APIs (for example, machine learning frameworks such as PyTorch [122] and Darknet [123] do so to support a wide range of inference platforms). This requires the maintenance of multiple code bases and complex build systems. However, the recent proliferation of specialised hardware platforms and accelerators entails a similar effect on the number of relevant library APIs [124, 125, 126, 127]. In the long-term, a multi-version code base is not sustainable or economical to maintain.

This chapter develops an alternative approach: a compiler-based scheme that discovers opportunities to use new accelerator libraries in user code, with little prior knowledge of the libraries. Furthermore, it can recover behaviourally equivalent code from programs that use existing libraries and automatically port them to new interfaces. In order to reduce developer burden, it attempts to minimise the level of manual intervention required from users.

Program synthesis is a well studied area that deals with searching a program space to find candidate programs that match a specification [67]. The implementation in this chapter, Annote, uses a number of generic control-flow components and a set of heuristics defining when they should be applied. These heuristics are driven by a library's type signature and lightweight annotations provided by the library vendor. Crucially, these annotations are easily extracted from documentation and require no knowledge of a library's internals. Because the libraries targeted for recovery are not adversarial in nature (that is, their structure is not *deliberately* obfuscated), these interface annotations and heuristics are able to effectively capture the likely behaviours of the library.

Once a library's behaviour is known, the next step is to see if the developer's program has structures that match its behaviour. This can be achieved by automatically describing the synthesised program as a set of constraints which can then be used to search the application code. As the synthesised program may not easily match existing code, many equivalent versions are generated, then normalised and then generalised to a common description that determines the most appropriate constraints. When a match is found, a suggestion is made to the developer that replacement code could take advantage of a different library.

The synthesis and constraint generation approach in this chapter allows large existing code bases to be targeted, showing significant performance improvement. By applying the approach to existing large applications from scientific computing and deep learning written in C, C++ and Fortran, speedups ranging from 1.1× to over 10× improvement can be demonstrated by applying suggested refactorings.

4.2 Motivating Example

To understand the motivation for the techniques described in this chapter, it is useful to consider a motivating example from real code. Specifically, this section illustrates how the approach helps in porting code that uses an existing library API to a new library API with increased functionality.

Consider the code sample on the left of Listing 4.1. This is an inner loop taken from a subroutine in NWChem [128], a widely used chemical simulation suite that makes explicit calls to BLAS libraries. The code contains manual loops over arrays and calls to BLAS routines (`dcopy` and `daxpy`). To improve performance, it may be profitable to port this code to use Intel's MKL libraries (as shown on the right hand side, which makes use of the extended MKL functionality `mkl_daxpby`).

At a high level, the code in Listing 4.1 performs a mixture of elementwise and vector-wide operations on structures representing a system of molecules. In this section, for exposition, the vector-wide operations are written equationally with names roughly corresponding to the original variable names. Because no dependencies are carried between the regions of code discussed in this way, the equational reasoning used is a sound notation for the analyses and transformations performed. This is not necessarily the case for general code; a more detailed discussion of when these assumptions break down is given later in the chapter once an intuition has been established.

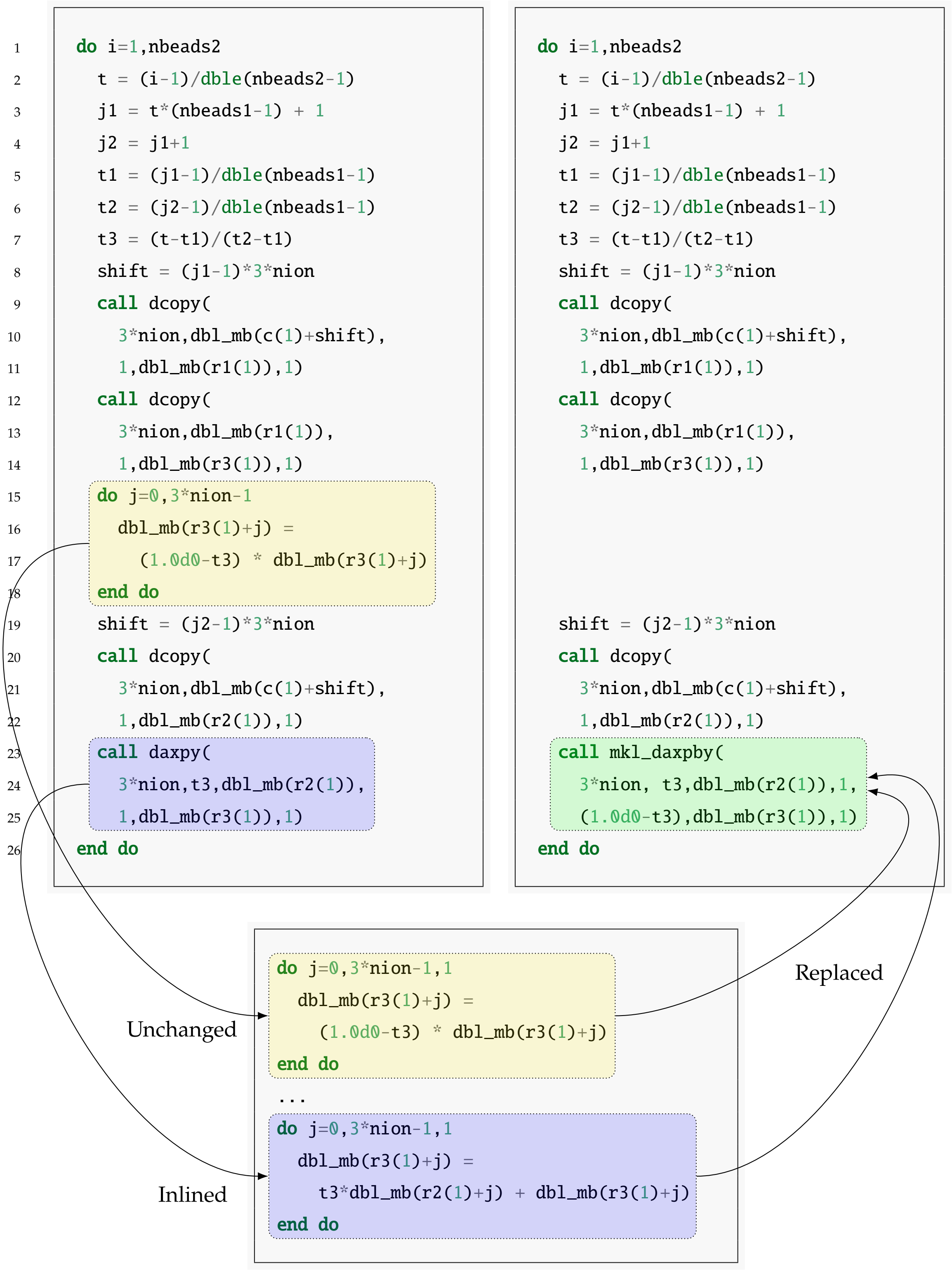
4.2.1 Original Intent

After the initial system state is established, on the left hand side of Listing 4.1, there are two highlighted sections of code. The first of these is a loop that performs the following abstract vector computation:

$$\mathbf{r_3} \leftarrow (1 - t_3)\mathbf{r_3} \tag{4.1}$$

The second highlighted piece of code is a call to `daxpy` . *If* the underlying source code for the function `daxpy` were available, it could be determined to perform the abstract vector operation:

$$\mathbf{y} \leftarrow a\mathbf{x} + \mathbf{y} \tag{4.2}$$



```
do i=1,nbeads2
  t = (i-1)/dble(nbeads2-1)
  j1 = t*(nbeads1-1) + 1
  j2 = j1+1
  t1 = (j1-1)/dble(nbeads1-1)
  t2 = (j2-1)/dble(nbeads1-1)
  t3 = (t-t1)/(t2-t1)
  shift = (j1-1)*3*nion
  call dcopy(
    3*nion,dbl_mb(c(1)+shift),
    1,dbl_mb(r1(1)),1)
  call dcopy(
    3*nion,dbl_mb(r1(1)),
    1,dbl_mb(r3(1)),1)
  do j=0,3*nion-1
    dbl_mb(r3(1)+j) =
      (1.0d0-t3) * dbl_mb(r3(1)+j)
  end do
  shift = (j2-1)*3*nion
  call dcopy(
    3*nion,dbl_mb(c(1)+shift),
    1,dbl_mb(r2(1)),1)
  call daxpy(
    3*nion,t3,dbl_mb(r2(1)),
    1,dbl_mb(r3(1)),1)
end do
```

```
do i=1,nbeads2
  t = (i-1)/dble(nbeads2-1)
  j1 = t*(nbeads1-1) + 1
  j2 = j1+1
  t1 = (j1-1)/dble(nbeads1-1)
  t2 = (j2-1)/dble(nbeads1-1)
  t3 = (t-t1)/(t2-t1)
  shift = (j1-1)*3*nion
  call dcopy(
    3*nion,dbl_mb(c(1)+shift),
    1,dbl_mb(r1(1)),1)
  call dcopy(
    3*nion,dbl_mb(r1(1)),
    1,dbl_mb(r3(1)),1)




  shift = (j2-1)*3*nion
  call dcopy(
    3*nion,dbl_mb(c(1)+shift),
    1,dbl_mb(r2(1)),1)
  call mkl_daxpby(
    3*nion, t3,dbl_mb(r2(1)),1,
    (1.0d0-t3),dbl_mb(r3(1)),1)
end do
```

```
do j=0,3*nion-1,1
  dbl_mb(r3(1)+j) =
    (1.0d0-t3) * dbl_mb(r3(1)+j)
end do
...
do j=0,3*nion-1,1
  dbl_mb(r3(1)+j) =
    t3*dbl_mb(r2(1)+j) + dbl_mb(r3(1)+j)
end do
```

Listing 4.1: Motivating example showing how code from NWChem can be inlined, matched and refactored to take advantage of a new library interface. In the code on the left, a manually written loop and BLAS library call are normalised by partial inlining (bottom). Then, the inlined code is matched against the `mkl_daxpby` function and replaced with a single call (right).

Substituting in the parameters $\mathbf{r_3}$, t_3 and $\mathbf{r_2}$ used in the example code to the equation for `daxpy` produces:

$$\mathbf{r_3} \leftarrow t_3\mathbf{r_2} + \mathbf{r_3} \tag{4.3}$$

The intent of the code (with respect to the vector `r3`) is made clear by these equations: it is updated to reflect the proportionately scaled sum of two vectors. Equationally, substituting the updated value of $\mathbf{r_3}$ from Equation (4.1) into Equation (4.3) yields:

$$\mathbf{r_3} \leftarrow t_3\mathbf{r_2} + (1 - t_3)\mathbf{r_3} \tag{4.4}$$

4.2.2 Updated Code

While `daxpy` is in the standard BLAS specification (and is therefore widely supported by compliant implementations), vendor-specific libraries often implement an extended set of functions. For example, when porting NWChem to MKL, an extended version of the function (`daxpby`) is available [28]. This extended function performs an additional vector scaling operation in addition to the work done by `daxpy` . Given equationally:

$$\mathbf{y} \leftarrow a\mathbf{x} + b\mathbf{y} \tag{4.5}$$

By substituting in parameters $\mathbf{r_3}$, $\mathbf{r_2}$, t_3 and $(1 - t_3)$ from the code in Listing 4.1, an identical vector operation to the combined intent in Equation (4.5) appears:

$$\mathbf{r_3} \leftarrow t_3\mathbf{r_2} + (1 - t_3)\mathbf{r_3} \tag{4.6}$$

This equivalence implies that `daxpby` implements precisely the behaviour required by the source-level loop and call to `daxpy` on the LHS of Listing 4.1. A maintainer working on porting the code could identify this, and legally replace the loop and `daxpy` call with a single call to `daxpby` . The code on the RHS of Listing 4.1 highlights the result of doing so. By applying this one transformation, 4 lines of code have been removed, one fewer loop is executed, and it is likely that `daxpby` is more performant than `daxpy` .

4.2.3 Procedure

The description above implies the presence of an expert maintainer who is able to correctly identify the preserved behaviour, and apply the resulting change to the code. This is, unfortunately, a time-consuming procedure: NWChem contains in excess of 4.8M lines of Fortran.[2] While many of these lines have been generated automatically (for example, producing derivatives of energy functionals), this is largely a one-off

[2] Measured using `cloc` [129] at revision `a74cd6faad1ce1778207250bc9b789a93b6451fc`.

process, and generated code is checked into and maintained as part of the source tree. Opportunities for automatically improving the code are therefore likely to be fruitful given the underlying repetition from this code-generation process.

This chapter, therefore, presents a semi-automatic scheme to identify similar opportunities in application code. Doing so entails two distinct phases:

Match and Replace: If a source-level description of both `daxpy` and `daxpby` is available, the original call is inlined directly. In Listing 4.1, the resulting code after inlining is summarised in the bottom-center box.[3] Then, the inlined code can be matched against the source-level description of `daxpby` from MKL. This is achieved using an existing, graph-based constraint solver [12], which applies a series of normalising compiler passes (such as loop fusion, which applied in this example), then detects compatible regions of code, and suggests potential refactorings to the developer.

Synthesis: In practice, it cannot be guaranteed that there is a suitable source level description for every library function called by an application. This may be due to the library provider not releasing an appropriate description, it no longer being available, or simply being poorly documented. It may also be defined in a manner suitable for human consumption, but not compiler automation. It is certainly the case that there is not agreement among all library developers about a universal language to describe the semantics of their libraries.

If the source code of the two libraries is in fact unavailable, program synthesis is used to generate programs corresponding to both (for example, `daxpy` and `daxpby` in the example in Listing 4.1) Then, the inlining and pattern matching procedure is performed as described above. By using synthesis in this way, the two halves of the problem (searching for replacements, and dealing with opaque libraries) can be decoupled and approached separately.

Thus, the developer is able to port their code to a new, extended library without having to identify the opportunity manually (they need only agree to the suggested replacement). For the code in Listing 4.1, for example, it results in a 20% local performance improvement on an Intel Xeon E5-2620.[4] If the code is to be ported again or an improved library is released, then the procedure can be repeated, avoiding legacy API tie-in. At the heart of the approach is the use of program synthesis and graph based generalised constraint matching.

[3] Calls to other functions (e.g. dcopy) will also be inlined, but are not shown in the figure for clarity.

[4] Measured by extracting the example code as a microbenchmark, and quoted for the sake of the example. More rigorous, full-application performance data is given in full in Section 4.6.

4.3 Type Annotation Language

To be able to synthesise implementations of library functions given only their interface, a definition of what a function's interface *actually is* is required. One possible such definition is to use only the function's type signature; knowing the type signature of a function allows correctly-typed input-output examples to be collected with respect to a particular calling convention.

However, a type signature alone is not an adequate specification for synthesis; the space of programs with any given signature is infinite. In some synthesis applications, the function oracle is able to provide partial information in a feedback loop (e.g. providing minimal, informative counterexamples to a proposed solution [50, 118]). The oracles provided by library functions cannot do so in general, and can provide only correctness checks for a proposed solution.

Fortunately, the type signature is not the only component to a library function's interface. Users of a library are aware of many properties of each interface that cannot be expressed in the type system (for example, the pointer `int *xs` points to exactly `int N` valid allocated elements), but are required to make correct use of the function.

Special-casing individual such sources of information in the synthesiser is not scalable. Ideally, they would be encoded more formally alongside the type signature in a well-defined common format to allow automated tools (such as a synthesiser) to make use of them.

To that end, this section introduces a minimal, flexible annotation DSL for type signatures. Additionally, semantics for a logical query language over this DSL are given, and examples of their application to synthesis are shown. By doing so, synthesis targeting functions with type signatures and additional associated information can be reduced to a better-defined, type-directed approach.

4.3.1 Context

For maximum interoperability, the calling convention used by targeted library functions is the ubiquitous C foreign function interface (FFI). This means that the starting context for the type annotation DSL is a simplified subset of the C type system. From a set of concrete base types, (`int`, `float`, etc.), pointers (`int*`) and aggregates (`struct{int x; int y;}`) can additionally be constructed and passed as function arguments.

Using this subset of C types, function signatures can be written interchangeably (depending on the context) as `Tr (T0 p0, ..., Tn pn)` or $(p_0 : \tau_0, \ldots, p_n : \tau_n) \to \tau_r$. Each parameter in the signature is assigned a unique identifier (as would be the case

```c
void (int n,
      float a, float *x,
      float b, float *y);
```

$$
\begin{aligned}
\mathit{size} &\subseteq P \times P \\
\mathit{size} &\triangleq \{(\mathtt{x},\mathtt{n}),(\mathtt{y},\mathtt{n})\} \\
\mathit{output} &\subseteq P \\
\mathit{output} &\triangleq \{\mathtt{y}\}
\end{aligned}
$$

Figure 4.1: The C function signature for the extended BLAS function `daxpby`, and the property relations that hold for it.

for a function signature in real code).

4.3.2 Property Relations

To extend C type signatures with additional properties, a system of free-form untyped relations is defined as follows. Let f be a function with type signature $T \triangleq (\tau_0, \ldots, \tau_n) \rightarrow \tau_r$, taking parameters $(p_0 : \tau_0, \ldots, p_n : \tau_n)$. Then, define:

$$
\begin{aligned}
P &\triangleq \{p_0, \ldots, p_n\} \\
C &\triangleq \text{set of all C types} \\
S &\triangleq \text{set of all C string literals} \\
N &\triangleq \text{set of all C numeric literals} \\
U &\triangleq P \uplus C \uplus S \uplus N
\end{aligned}
$$

Then, f can be equipped with a set of relations R_f in addition to its type signature, such that R_f expresses the "additional" information known about the interface of f. Each relation $r_i \in R_f$ satisfies $r_i \subseteq U^k$ for some $k > 0$. Additionally, a naming function I is defined such that $I(r_i)$ is a unique identifier for each r_i.

Less formally, named relations contain untyped sets of "atoms", where those atoms can be function parameters, literal values or C types (i.e. they are $\in U$). The particular semantics of each relation are in a sense *extrinsic*; the construction of each entry has no meaning in itself, but properties of the signature may be interpreted by other systems by inspection of the relations.

This specification is intentionally simple; it is the smallest definition that allows for sufficiently useful properties to be encoded, while maintaining a close relationship to the function's type signature. Indeed, the existing signature syntax is kept for familiarity only, as type annotations could be made homogeneously in the same relational environment.[5]

[5] i.e. $\mathit{type} \subseteq P \times C \triangleq \{(p_0, \tau_0), \ldots\}$

$$\frac{u \in R \qquad \Gamma \sim \{(v,u)\}}{\Gamma[v \mapsto u] \vdash R(v)}\ \text{(bind)} \qquad\qquad \frac{u \in R}{\Gamma \vdash R(u)}\ \text{(lit)}$$

$$\frac{\Gamma \vdash R(x) \qquad \Gamma' \vdash T(y) \qquad \Gamma \sim \Gamma'}{\Gamma \cup \Gamma' \ \vdash R(x) \wedge T(y)}\ \text{(conj)}$$

$$\frac{u \notin R}{\Gamma \vdash \neg R(u)}\ \text{(neg1)} \qquad \frac{\Gamma \vdash \neg R(u)}{\Gamma[v \mapsto u] \vdash \neg R(v)}\ \text{(neg2)} \qquad \frac{\neg \exists u \,.\, \Gamma \vdash R(u))}{\Gamma \vdash \neg R(\_)}\ \text{(neg3)}$$

$$\frac{P \text{ is a base relation} \qquad P(x) \text{ holds}}{\Gamma \vdash P(x)}\ \text{(base)}$$

Figure 4.2: Unification rules for the property query language defined in this section. All relations are shown as unary relations for clarity; the same inference rules can be trivially lifted to any arity. Metavariables u, v are implicitly restricted to be $\in U$ and $\in V$ respectively.

Returning to the application code given in Listing 4.1 at the beginning of this chapter, an example of how properties of the function `daxpby` can be encoded in this system is given in Figure 4.1. Two properties are defined for the function ($size$ and $output$; the synthesis interpretation of these is given in Section 4.4), ranging over parameter names in particular.

4.3.3 Queries

As discussed above, the semantics of each relation are defined extrinsically by other systems applying meaning to them through interpretation. The primary mechanism through which this can be achieved is a *query language*, in the spirit of a simple logic programming language. By constructing queries in this language, the structure of the relations associated with a function can be deconstructed and inspected.

The semantics of this query language are defined simply, in terms of unification rules over first-order terms from the annotation language. A *query* Q is written $R(x_0, x_1, \ldots)$, and can be understood informally to mean "does the relation R contain the entry $(x_0, x_1, \ldots)$?". To allow for unification, queries can range over the set:

$$U_V \triangleq U \cupdot V \qquad (4.7)$$

for some set V of variable identifiers not already in U. That is, queries can *bind* identifiers $\in V$ to the concrete elements contained in the relation.

To define the semantics of this query language, a definition for unification is required. A *unification context* Γ is a set of ordered pairs:

$$\Gamma \subseteq (V \times U) \tag{4.8}$$

A context represents a mapping from symbolic identifiers to concrete elements of U. Two operations on contexts are defined: extension and compatibility checking. Extending a context is defined only when name clashes do not occur:

$$\Gamma[x \mapsto v] \triangleq \Gamma \cup (x, v) \text{ iff } (x, v) \notin \Gamma \tag{4.9}$$

Then, define two contexts Γ, Γ' to be compatible (written as $\sim$) when:

$$\Gamma \sim \Gamma' \triangleq \forall (v, u) \in \Gamma \,.\, (v, u') \in \Gamma' \implies u = u' \tag{4.10}$$

That is, when two contexts bind the same symbolic identifier, they must bind it to the same concrete element in U. Identifiers they do not share are irrelevant.

Given these definitions, checking a query can be defined inductively. Write:

$$\Gamma \vdash R(x_0, \ldots, x_n) \tag{4.11}$$

for the judgement "The relation R holds for $x_0, \ldots$ in context Γ". For a query to succeed, there must exist a context Γ that satisfies the judgement. Figure 4.2 shows inductive rules defining when this holds; an algorithm to construct Γ follows naturally from these rules.

In Figure 4.2, "base relations" are interpreted outwith the context of the annotated property relations. For example, the relation $\textit{is-pointer}(T)$ holds whenever the parameter type T is a pointer type (rather than a scalar). The definitions for these relations are intuitive and can be inferred from the relation name when they appear.

Interpreting the query unification rules can be best understood with a worked example. In Figure 4.3, a type signature is shown along with a set of property annotations that apply to it. Then, a query used by Annote during synthesis is given, along with a derivation for a valid unification from that query.

4.3.4 Summary

This section has demonstrated a DSL for annotating function type signatures with arbitrary properties, and an efficient, robust unification-based query algorithm for these properties. While used in the remainder of this chapter to drive program synthesis, the annotation and query DSL could be applied to more general static analysis methods in future work. For example, static analyses attempting to prove general properties of code could be effectively directed by property annotations and queries where inference is intractable. Existing techniques using interface definition languages could be adapted to the general annotation framework in this chapter.

```
void f(int n, int *x, int *y, int *z)
```

(a) Type signature of a function compatible with Annote.

$$\begin{aligned} size &\subseteq P \times P \\ size &\triangleq \{(\mathtt{x},\mathtt{n}),(\mathtt{y},\mathtt{m})\} \\ output &\subseteq P \\ output &\triangleq \{\mathtt{y}\} \end{aligned}$$

(b) Property annotations for the above type signature, showing which parameters represent pointer size bounds, and which represent outputs from the function.

$$\begin{aligned} &\neg size(A,\_) \ \wedge \\ &type(A,T) \ \wedge \\ &is\text{-}pointer(T) \end{aligned}$$

(c) An example of a valid query for the above type signature, as used by Annote during synthesis.

$$\dfrac{\dfrac{\neg\exists u\ .\ size(\mathtt{z},u)}{\emptyset[A \mapsto \mathtt{z}] \vdash \neg size(A,\_)} \qquad \dfrac{\mathtt{z}\text{ has type }\mathtt{int*}}{\emptyset[A \mapsto \mathtt{z}, T \mapsto \mathtt{int*}] \vdash type(A,T)} \qquad \dfrac{\mathtt{int*}\text{ is a pointer type}}{\emptyset[T \mapsto \mathtt{int*}] \vdash is\text{-}pointer(T)}}{\Gamma \triangleq \emptyset[A \mapsto \mathtt{z}, T \mapsto \mathtt{int*}] \vdash \neg size(A,\_) \wedge type(A,T) \wedge is\text{-}pointer(T)}$$

(d) Abbreviated proof tree showing a valid unification for the query, property set and type signature shown above, with compatibility (∼) checks omitted for brevity.

Figure 4.3: Worked example of query unification being applied to a concrete type signature and set of properties. Annote uses a query similar to the one shown above during synthesis.

4.4 Synthesis

Annote uses the signature annotation and query DSL described previously to drive the heuristic selection of program sketches in its synthesis process. This section describes in detail the synthesis algorithms used by Annote to elaborate a correctness specification (i.e. IO examples and type annotations) into a program sketch, and finally into a correct solution.

4.4.1 Overview

To synthesise programs, Annote follows a two-phase, sketch-based methodology. Starting with a specification (type signature, property annotations and an oracle from which input-output examples can be generated), it generates many input-output pairs by calling the oracle repeatedly with different, random input data. These pairs specify correctness for possible solutions as observational equivalence (if a solution matches the oracle on every *observed* example, it is judged to be correct).

Next, a set of heuristic rules are matched against the annotated properties. Successfully matched rules yield an instantiated program fragment; the set of all fragments matched forms the starting point from which program sketches are constructed. Enumerating possible compositions of the available program fragments gives a set of sketches, each of which potentially represents the structure of a correct synthesis solution.

For each sketch, candidate programs are searched for by enumerating possible instruction values at each hole in the sketch (using the abstractions described in Section 3.6.1 to safely instantiate holes in the possible absence of type information). Each candidate program is tested using the same set of input-output pairs as the reference function; if all examples match, then the candidate is observationally equivalent and can therefore be returned as a correct solution.

The remainder of this section describes each of the components of this synthesis workflow in detail.

4.4.2 Sketching Control Flow

The program sketches used by Annote to direct its search for candidate solutions are implemented in terms of the fragment language and semantics first defined in Section 3.5. In particular, fragment templates are combined with the query language defined in Section 4.3.3 such that the bound variables from successfully unified queries can be used to instantiate fragments from templates. Enumerating possible compositions of these fragments produces a set of sketches.

This section gives details of the template instantiation extension to the property query DSL described previously, as well as the algorithm to produce composed program sketches. Additionally, it provides full listings of the fragment classes, annotations and heuristic rules used to drive this portion of the synthesiser.

4.4.2.1 Fragment Instantiation

Section 4.3.3 specified a query DSL for property relations attached to a type signature. When a query is successfully matched against a set of relations, the unification algorithm produces a set of bound identifiers. Annote implements a set of heuristic rules, each of which has the general form:

$$Q(q_0, \ldots, q_n) \implies F(f_0, \ldots, f_m) \tag{4.12}$$

Algorithm 1 Generating fragments

```
function GenerateFragments($props, rules$)
    $frags \leftarrow \emptyset$
    for each $rule$ in $rules$ do
        $matches \leftarrow$ Match($rule.query, props$)
        for each $match$ in $matches$ do
            $f \leftarrow rule.template$ instantiated with $match$
            $frags \leftarrow frags \cup f$
        end for
    end for
    return $frags$
end function

function Match($query, props$)
    return set of all valid unifications of $query$ against $props$
end function
```

where Q is any query in the DSL from Section 4.3.3, F is a fragment template class as defined in Section 3.5, and each q_i, f_i are query identifiers such that:

$$\{f_0, \ldots, f_m\} \subseteq \{q_0, \ldots, q_n\} \tag{4.13}$$

The query Q, when matched successfully, binds values to each q_i. The fragment produced by instantiating the template with the corresponding values bound to each f_i is added to the active set of fragments for the current synthesis problem. Algorithm 1 details the fragment instantiation and collection algorithm in full.

By way of example, consider an example rule used by Annote:

$$\begin{aligned} & \mathit{size}(X,N) \wedge \mathit{size}(Y,N) \wedge \\ & \mathit{type}(X,T) \wedge \mathit{type}(Y,S) \wedge \mathit{type}(N,\texttt{int}) \\ & \mathit{is\text{-}pointer}(T) \wedge \mathit{is\text{-}pointer}(S) \\ \implies & \texttt{zip\_loop}(N,T,X,S,Y) \end{aligned} \tag{4.14}$$

The intent of this rule is to instantiate a "zip" loop to iterate jointly over two pointers, each of which has the same statically-known size N. Each parameter for the `zip_loop` fragment class is bound by the variables in the query.

4.4.2.2 Annotations

Annote uses 5 property annotations to describe function interface properties, each of which could be readily derived from library documentation for the relevant functions.

The semantics of each of these is given below, along with a brief explanation of why these are *interface* properties (rather than internal implementation details):

`size(xs, n)`: the parameter `xs` points to valid, allocated memory with at least `n` elements; loading a value from memory anywhere in the valid region is memory-safe. If a caller passes a region of memory to a function that is not consistent with this annotation, the internal implementation will almost certainly attempt to access invalid memory.

`output(x)`: the pointer `x` is an output parameter for the function. This can be communicated to a certain extent by the C type system's `const` keyword, but this is weakly enforced; calling code must still be aware when allocated memory should be writable when calling a function with output parameters.

`enum(x, c0, ..., cN)`: the parameter `x` must take one of the distinct constant values `c0`...`cN`. Again, weak enforcement and communication of this property exists in the C type system, but the calling code

`pack(xs, c)`: each logical entry in the array pointed to by `xs` contains `c` physical elements. Similarly to `size`, this ensures that all memory accesses made by the called function are valid.

`indices(xs)`: elements of `xs` in memory are logically array indices. This annotation is the least common, and is the only one not directly linked to a *safety* property of the called function. However, it was used only to synthesise one function class (`spmv`), and remained readily available in documentation for these functions.

All of these annotations are communicated explicitly in written form by the user documentation of each library, because of their relationship to safety and correctness assertions made by the called function. As a result, annotating the individual function signatures with the appropriate properties was not an onerous task and did not require specific knowledge of their implementation details.

4.4.2.3 Fragments

Using the sketch language semantics given in Section 3.5, Annote defines a basis set of fragment templates implementing common control-flow idioms. The semantics and structure of these templates was derived in large part from the idiom definitions developed by Ginsbach et al. [112], along with basic conditional control flow not relevant to their detection of idioms. A full listing of the template structures used, and the parameters they accept to instantiate a concrete fragment are listed in Table 4.1.

Table 4.1: Listing of fragment templates that can potentially be instantiated by Annote to populate the set of fragments used in sketches. In this table, the semantics of the templates are described informally as if they were C-like code. Their semantics are defined in practice by the implementation of the *compile* operation for each template (Section 3.5).

Template	**Parameters**	**Description**
`loop`	N : `value`	A loop structure similar to a C for-loop up to a fixed upper bound (where that bound is either an integral constant or a function parameter).
`regular_loop`	N : `value` T : `type` X : `param`	Similarly to `loop`, a for-loop to a fixed upper bound, but with the intent of looping specifically over (and acting on) an array X containing data of type T.
`zip_loop`	N : `value` T : `type` X : `param` S : `type` Y : `param`	Analogous to `regular_loop`, but specialised for the case where multiple arrays have identical size N. Shown here in the binary case, but is implemented by Annote for any number of arrays.
`if_else`	-	Unparameterised conditional statement; the condition and both arms of the statement are filled by the synthesis process.
`switch`	X : `param` C_0 : `int` C_1 : `int` ...	Switch statement parameterised on a set of constant values (i.e. X is checked against each C_i in turn). Implemented by Annote for any number of constant values.
`index`	T : `type` X : `param`	Use a synthesised value from the current scope to explicitly index into the array X.
`output`	T : `type` X : `param`	Write a synthesised value of type T to X or any pointer obtained by indexing into it.

1

$$\neg \mathit{size}(\_,N) \wedge \mathit{type}(N,\texttt{int}) \\ \Longrightarrow \texttt{loop}(N)$$

2

$$\mathit{size}(X,N) \wedge \mathit{type}(N,\texttt{int}) \wedge \\ \mathit{type}(X,T) \wedge \mathit{is\text{-}pointer}(T) \\ \Longrightarrow \texttt{regular\_loop}(N,T,X)$$

3

$$\mathit{size}(X,N) \wedge \mathit{type}(N,\texttt{int}) \wedge \\ \mathit{type}(X,T) \wedge \mathit{is\text{-}pointer}(T) \wedge \\ \mathit{size}(Y,N) \wedge \\ \mathit{type}(Y,S) \wedge \mathit{is\text{-}pointer}(S) \\ \Longrightarrow \texttt{zip\_loop}(N,T,X,S,Y)$$

4

$$\Longrightarrow \texttt{if\_else}()$$

5

$$\mathit{enum}(X,C_0,\ldots,C_n) \wedge \\ \mathit{type}(X,\texttt{int}) \wedge \mathit{type}(C_0,\texttt{int}) \wedge \\ \ldots \wedge \mathit{type}(C_n,\texttt{int}) \\ \Longrightarrow \texttt{switch}(X,C_0,\ldots,C_n)$$

6

$$\neg \mathit{size}(X,\_) \wedge \\ \mathit{type}(X,T) \wedge \mathit{is\text{-}pointer}(T) \\ \Longrightarrow \texttt{index}(X)$$

7

$$\mathit{output}(X) \wedge \\ \mathit{type}(X,T) \wedge \mathit{is\text{-}pointer}(T) \\ \Longrightarrow \texttt{output}(X)$$

8

$$\mathit{pack}(C,\_) \wedge \mathit{type}(C,\texttt{int}) \\ \Longrightarrow \texttt{loop}(C)$$

9

$$\mathit{indices}(X) \wedge \\ \mathit{type}(X,T) \wedge \mathit{is\text{-}pointer}(T) \wedge \\ \mathit{type}(Y,S) \wedge \mathit{is\text{-}pointer}(S) \\ \Longrightarrow \texttt{index}(Y)$$

10

$$\mathit{pack}(\_,X) \wedge \mathit{type}(X,T) \wedge \\ \mathit{is\text{-}pointer}(T) \\ \Longrightarrow \texttt{index}(X)$$

Figure 4.4: Full listing of the query rules used by Annote to instantiate fragments from function interface properties.

The set of fragment templates used is not deliberately specialised or biased towards a particular function or library; they are designed to reflect common code structures *as a human developer might have written*, and to be intuitively inferred based on property annotations.

4.4.2.4 Rules

The full set of rules used by Annote to instantiate fragments from its set of templates is given in Figure 4.4. These rules were developed in parallel with the sets of annotations

Algorithm 2 Combining fragments

```
function CombineFragments($frags, maxDepth$)
    $ps \leftarrow \emptyset$
    for $f$ in $fragments$ do
        $ps \leftarrow ps \cup$ CombinePartial($f, frags, maxDepth$)
    end for
    return $ps$
end function

function CombinePartial($partial, frags, depth$)
    $p \leftarrow \{partial\}$
    if $depth > 0$ then
        for $frag$ in $fragments$ do
            $new \leftarrow partial \circ frag$
            $p \leftarrow p \cup$ CombinePartial($new, frags, depth - 1$)
        end for
    end if
    return $p$
end function
```

and fragments it uses (Section 4.4.2.2 and table 4.1 respectively), based on "commonsense" interpretations of property annotations. For example, if documentation states that a pointer should point to at least n elements, and that it is also an output parameter, it is likely the case that each pointed-to element might be written to (combining rules 2 and 7 from Figure 4.4). This interpretation relies on the assumption common to this entire section: that the underlying code for a library function was originally written intuitively. That is, the code is not deliberately obfuscated or implemented to behave pathologically with respect to observational equivalence or synthesis techniques. It is easy to construct functions that are intractable to synthesise, but these functions do not frequently line up with useful application code written in practice.

4.4.2.5 Composition

After matching the rules in Figure 4.4, Annote has a set of individual fragments likely to appear in successfully synthesised solutions. Its next step is to produce compositions of these fragments corresponding to sketches of whole-program structure.

To do this, it uses a simple enumerative algorithm to produce all possible unique compositions of fragments up to a specified upper size bound; this algorithm is given

Algorithm 3 Dataflow generation

```
function FillDataflow($cfg, n$)
    $tree \leftarrow$ dominance tree of $cfg$
    $phis \leftarrow \emptyset$
    for each $block$ in inorder($tree$) do
        if $block$ has > 1 predecessors then
            $phis \leftarrow phis \cup \{$ new untyped $\phi$ node in $block$ $\}$
        end if
        for each $hole$ in $block$ do
            $live \leftarrow$ live SSA values at $hole$
            sort $live$ by proximity heuristic
            fill $hole$ by sampling geometrically from $live$
        end for
    end for
    for each $\phi$ in $phis$ do
        $live \leftarrow$ live SSA values at block with $\phi$
        choose incoming values to $\phi$ from $live$
    end for
end function
```

in Algorithm 2. While the number of generated sketches grows exponentially with the number of fragments available, the number of rules matched for any individual signature is small for every function in the evaluation dataset. Intuitively, it is unlikely that a function expresses every possible control-flow idiom available at the same time; instead it is likely to express a small subset.

4.4.3 Generating Dataflow

Given a sketch composed from several fragments (i.e. a partial program containing holes, as defined in Section 3.6.1), the final step in ANNOTE's synthesis process is to add data-flow instructions to the sketch, producing a candidate program.

Reifying a sketch with holes to an executable candidate program can be done using a generic algorithm that requires no specific knowledge of the fragments that make up the sketch. The algorithm (shown in Algorithm 3) walks the dominance tree of the sketch in-order, filling *holes* in the structure with stochastically sampled values. Additionally, because the sketch is in SSA form, ϕ nodes are inserted to handle looping or divergent control correctly.

At each node in the dominance tree, a set of "live" instructions is computed. The instructions in this set come from fragment definitions (for example, values loaded from memory in a loop iteration), as well as intermediate holes previously in the program. When a basic block in the sketch is traversed, the set of live values at each hole is sorted by a proximity heuristic: roughly, instructions are more likely to operate on more recently defined values than they are on "older" ones.

Holes are filled by sampling from a set of "recipes" that combine multiple instructions to produce a value. For example, if two values are both integers, then a valid way to combine them is to create an `add i32` instruction. Arguments are sampled from the sorted list of live values available at the current hole (weighting more recent values more highly), then a way of combining them is sampled from the set of valid combinations for those arguments. Validity is determined by a predicate for each recipe.

Available operations include integer and floating point arithmetic, calls to intrinsic mathematical functions, and a small number of other simple primitives such as conditional selects. Searching for a correct candidate program amounts to iteratively performing this instruction generation algorithm on each possible control flow structure, testing each resulting program for behavioural equivalence until a solution is found.

4.4.4 Verification

The final step taken by Annote is to test the reified solution program with a large number of input-output examples to determine whether or not it is correct. Because the synthesis problem posed in this chapter deals with "black-box" oracles, no *ground truth* specification is available for each synthesis problem. That is, given a candidate solution, there is no way to formally determine whether or not it is correct with respect to the library function in question.

The only available test for correctness in this context is *observational equivalence*: if a candidate exhibits identical behaviour to the oracle across a set of representative input examples, then they are observationally equivalent. Candidates reported as correct syntheses in this chapter are those that are observationally equivalent to the relevant reference function.

As the library functions targeted for synthesis by Annote act over a domain of floating-point numbers, the precise definition of "identical" behaviour is relaxed to accommodate cases where IEEE floating-point arithmetic diverges from ideal real-number semantics. For example, in floating point arithmetic, multiplication is not associative; it does not hold that $a*(b*c) = (a*b)*c$ in general. The order of operations chosen by a particular oracle is unlikely to be significant, and so it is useful to allow synthesised solutions to potentially select reordered expressions while preserving the underlying real-number calculation.

To achieve this, approximate floating point equality is used: if two numbers are separated by at most 10ULP or an absolute value of 0.00001, they are judged to be equal for the purposes of observational equivalence. No cases where this approximation permitted a fundamentally incorrect program to be judged correct were identified. These bounds can be configured depending on the demands of a particular scenario.

This test is unsound by definition, unfortunately; there is no way to establish formally that a candidate program will behave correctly on every possible input, as the number of possible input examples is intractably large. However, this unsoundness is common in property-based software testing and program synthesis, and is normally accepted to be an acceptable definition when stronger guarantees are not available [130]. Additionally, in the context of this chapter, the targeted oracles are performance-sensitive library functions originally written by a human developer. Such functions are unlikely to exhibit different behaviour on a sparse subset of their input domain, increasing the likelihood that observational equivalence on a large set of input examples is good evidence that a solution is in fact correct.

If the original source code for the targeted library functions was in fact available, then more sophisticated techniques to determine correctness can be deployed. For example, in Chapter 7, symbolic model checking with KLEE [131, 132] is deployed to verify solutions.

4.4.5 Summary

This section has presented the core synthesis algorithms used by Annote to learn programs that are observationally equivalent to library functions. By matching heuristic rules against simple-to-provide property annotations on a function's type signature, a set of program fragments is identified and composed into potential sketches. Then, values are sampled stochastically to fill the holes in generated sketches. This produces a reified program that can be executed and tested for observational equivalence against the library function oracle.

4.5 Experimental Setup

The evaluation of Annote in this chapter can be split into two distinct questions.

- Can program synthesis (i.e. Annote, as implemented in this chapter) make use of type signature annotations to discover programs behaviourally equivalent to real-world library functions?

Name	Kernels	Acceleration	LoC
NWChem	Dense	BLAS	1.2M
Abinit	Dense	BLAS,CUDA	900k
Pathsample	Sparse	Handcoded SpMV	40k
Darknet	Neural Network	CUDA	27k
Parboil	Linear Algebra	Handcoded MxM	187

Table 4.2: Application source code used for evaluation.

Library	Platform	Kernels
Intel MKL	Intel CPU	Dense Linear Algebra
cuBLAS	Nvidia GPU	Linear Algebra
cuDNN	Nvidia GPU	Neural Networks
cuSparse	Nvidia GPU	Sparse Linear Algebra
CLBlast	OpenCL Devices	Dense Linear Algebra

Table 4.3: Optimised libraries selected for evaluation.

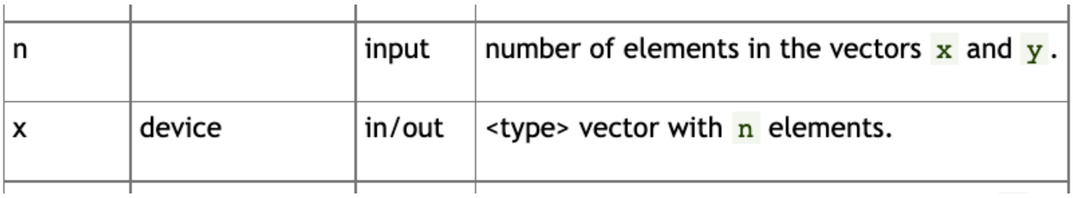
- Given such correctly synthesised implementations, can they be generalised to CAnDL constraint descriptions [12] and used to suggest valid refactorings that improve application performance?

In this section, a detailed methodology to answer these questions is presented. First, a collection of well-known libraries containing optimised versions of common computational bottlenecks is identified. The function signatures in these libraries are then annotated with the relevant interface properties listed in Section 4.4.2.2, and synthesis is attempted using Annote.

Then, a corresponding group of widely-used, performance-sensitive applications (that are likely to be able to make use of platform-specific optimised library routines) is identified. For functions that could be successfully synthesised by Annote, CAnDL constraints are generated following the method described in Section 3.3[6]. Each application's source code is then searched for matches of these constraints.

Finally, where successful constraint matches are found, the corresponding code refactoring is applied to the application (i.e. replacing original application code with calls to optimised library routines). The performance of each application on representative whole-program workloads, before and after applying these refactorings is measured and reported.

[6] Developed by Philip Ginsbach, a co-author of Collie et al. [2]



n		input	number of elements in the vectors `x` and `y`.
x	device	in/out	<type> vector with `n` elements.

Figure 4.5: An extract from the documentation of Nvidia's cuBLAS library, which provides optimised GPU implementations of linear algebra routines. This extract demonstrates (at least anecdotally), that documentation often provides the type of information used to annotate type signatures with properties of their interface.

4.5.1 Libraries

The libraries selected for evaluation are summarised in Table 4.3. They fall broadly into two categories: those that are optimised for a particular CPU architecture (Intel MKL) to achieve performance, and those that use the GPU (CUDA libraries, CLBlast). Each library targets a slightly different specific problem domain, but there are several common functions that are implemented in multiple libraries (for example, matrix-vector multiplication is a linear algebra operation contained in the BLAS standard, but it is also a fundamental building block of neural network architectures).

To prepare the libraries for synthesis, the subset of their functions compatible with Annote's interface (see Section 4.3) was identified. Informally, restricting the functions in this way selects for those most likely to present performance bottlenecks; features such as pointers to pointers are more likely to appear in setup functions or in stateful contexts such as memory allocation. Such cases are beyond the scope of this chapter.

For each function in the compatible subset, the corresponding library documentation was examined to identify which property annotations applied to it. For example, an excerpt from the Nvidia documentation for the function `cublas_daxpy` is shown in Figure 4.5. From this documentation, the relations $size(\mathtt{x},\mathtt{n})$ and $output(\mathtt{x})$ can be easily inferred; the property annotations used by Annote simply codify existing knowledge required of a human user of the library. Finally, a shared library loadable by Annote was compiled, re-exporting the relevant functions from each library.

4.5.2 Applications

To evaluate the potential for real-world performance improvements by migrating applications to new, optimised libraries, 5 representative example applications were selected (these are listed in Table 4.2). Each application could potentially make use of a different subset of the available libraries (e.g. some use sparse methods extensively, while others are bottlenecked on specific dense routines). Additionally, the applications vary in their *existing* use of acceleration methods: some are already locked in to a particular library (e.g. standard BLAS or CUDA), while others contain hand-coded library routines in their source. This variation serves to demonstrate the flexibility of the unified, inlining-based approach to migration.

Three of the selected applications (NWChem, Abinit, and Pathsample) are scientific simulation applications from different problem domains (molecular dynamics, density functional theory, and discrete path sampling respectively). Each of these represents a large code base, widely used by active research groups; NWChem contains in excess of 1M lines of code, and Abinit alone has been cited more than 6,000 times since 2002.

The scientific applications selected have different existing strategies for acceleration: Abinit must be linked to a BLAS implementation installed somewhere on the target system, while NWChem bundles an internal linear algebra library and Pathsample implements a small set of required operations by hand (including some sparse methods). Each application is distributed with several example workloads; two of these standard datasets per application (corresponding to different chemical scenarios, and exercising different sections of their code) were used to evaluate performance.

As well as scientific applications, machine learning and neural network workloads are a significant consumer of acceleration libraries. To represent this domain, the Darknet [123] framework was selected. Darknet is a widely used, open-source deep learning framework that can be used to instantiate implementations of different neural architectures (for example, it has been used recently to implement a number of highly cited, state-of-the-art computer models [133, 134, 135]). It offers two distinct implementations of its underlying neural network primitives, for the CPU and GPU (written in hand-optimised C and CUDA, respectively). To evaluate performance, Darknet implementations of three well known models for the ImageNet classification task were selected. These models exercise the underlying primitives in different ways, exhibiting different performance characteristics as a result.

Finally, to quantify the *best possible* performance achievable using the proposed library substitution approach, the matrix-multiplication benchmark (SGEMM) from Parboil was selected [136]. Because this benchmark contains a single linear algebra routine, and performs no additional work, it represents the limit of potential performance improvements. That is, Amdahl's law states that as the proportion of work p performed in the accelerated component approaches 1, the total speedup S_{total} approaches s, the component speedup:

$$S_{total} = \frac{1}{(1-p) + \frac{p}{s}} \tag{4.15}$$

4.5.3 Platform

The primary target platform for the experiments in this section was a dedicated, single-tenant server with a 24-core Intel Xeon E5-2620 processor, 16GB of RAM and an Nvidia Tesla K20 GPU. Applications were compiled at `-O3` using a source build of GCC 9.1.[7] Intel MKL version 2019.1.144 and version 8.0 of the CUDA libraries were used. To evaluate the performance impact of porting applications to a different platform, a secondary server with a 12-core AMD A10-7859k CPU and an integrated AMD Radeon R7 iGPU was used. For brevity, these machines are referred to as the Intel and AMD platforms, respectively.

4.5.4 Benchmarking Methodology

Each application was run on each available dataset in its "out of the box" configuration on the Intel platform to provide a performance baseline. This configuration represents the default performance achievable by each application without dedicated work to optimise the workload.

For Pathsample the baseline code is sequential, handwritten Fortran with no library calls. This is also the case for NWChem, which contains sequential implementations for a subset of the BLAS standard. For Abinit, the baseline configuration links to standard BLAS libraries installed on the host system. Darknet offers a sequential C implementation as well as a CUDA GPU one; the C version was selected as the baseline as the GPU version requires specific configuration.

The performance of each application was then measured after implementing the suggested code replacements for a single library at a time, on the Intel platform. In each case, this entailed using either Intel MKL 8.0 or a subset of the collected CUDA libraries.

Additionally, as Darknet offers a hand-optimised CPU OpenMP runtime, the relative performance impact of migrating its sequential C implementation to the GPU was evaluated using the CLBlast library.

4.6 Results

The results in this section are structured as follows. First, the overall performance improvements achievable through library migration are summarised across the applications and libraries selected for evaluation. Next, the number of library calls and candidate matches for API migration is shown, demonstrating the effectiveness of both Annote's synthesis methods. This is followed by an evaluation of the execution time

[7] The most recent version available at the time this work was orginally carried out.

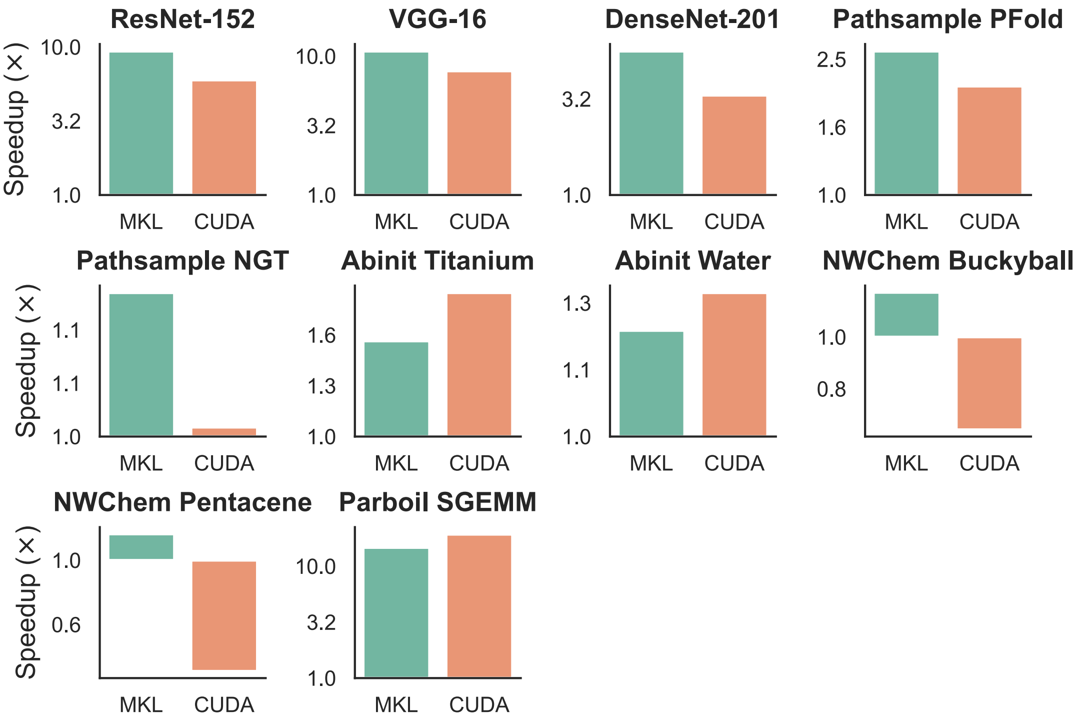


Figure 4.6: Performance achievable by adopting code replacements suggested by our tools, for both Intel MKL and Nvidia CUDA libraries across the set of benchmarks listed.

needed to synthesise equivalent programs for each library. Finally, the accuracy of the genetic graph matching algorithm is evaluated, and a discussion of the potential for unsound behaviour to arise when using these tools is presented.

4.6.1 Overall Results

A summary of the performance difference for each application's workloads on the Intel platform is shown in Figure 4.6. On scientific applications, the best implementation for each one achieved speedups of between 1.2 and 2.7× (note that this is the end-to-end performance of each application, rather than just isolated kernels). In Pathsample, the NGT workload spends less time in sparse matrix operations than the PFold workload; Amdahl's law means that inevitably PFold will benefit more from acceleration. MKL outperforms the Nvidia libraries by a small margin in both cases. This is due to the overhead of setting up the CPU-GPU interface, and transferring kernels across the shared bus. Memory tagging and a copy-on-write page caching scheme has been used in similar work to reduce this overhead [23]. Even if only Nvidia libraries were available, no slowdown was observed.

NWChem exhibits similar characteristics, with MKL significantly outperforming

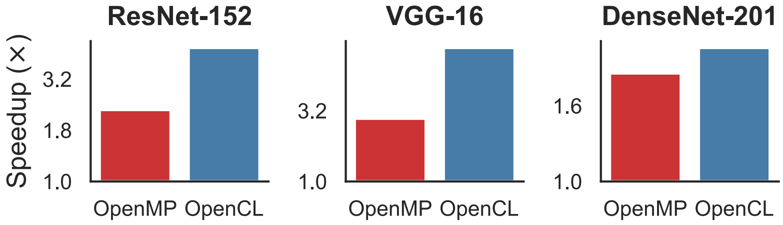


Figure 4.7: Performance results for neural network inference on an AMD device with no CUDA support.

the Nvidia libraries. Modest speedups are available for both configurations with an end-to-end speedup of 1.2× available. This is due to the increased heterogeneity in NWChem's workloads; far less of its computational time is spent in readily accelerable routines.

Abinit's behaviour differs; the Nvidia libraries outperform MKL, giving 1.2 to 1.9× speedup. This is due to the use of far larger homogeneous problem sizes, where the benefits of acceleration outweigh GPU communication overhead. Both MKL and CUDA are able to provide speedups on these workloads.

More significant improvements can be observed for the neural network workloads, as the amount of time spent in accelerator code sections increases further relative to the scientific applications. Improvements range from 5.5× for the smaller DenseNet-201 to 11× for the largest network, VGG-16. Like Pathsample and NWChem, all the networks achieve the greatest performance with MKL, though Nvidia libraries still give improvements (3.2× to 7.7×).

Finally, the Parboil SGEMM benchmark clearly illustrates the impact of Amdahl's law on this type of coarse-grained optimisation strategy. Here, there is just one kernel that can be readily accelerated; by doing so, speedups of 15× to 19× can be achieved. This represents, roughly, the best case performance improvement achievable with this kind of optimisation.

4.6.2 Porting to New Hardware

Within Darknet, the use of optimised GPU libraries is built into the code; CUDA and CPU implementations are mixed together using preprocessor directives and the build system. As CUDA is not available on AMD GPU platforms, porting Darknet to such a platform means targeting OpenCL based libraries such as CLBlast [137].

Using a similar methodology as for the Intel platform, the performance of the "out-of-the-box", CPU implementation of Darknet was compared against a hand-optimised parallel OpenMP version [138], and a version targeting CLBlast using suggested code

Table 4.4: Summary of locations in application code where matches of synthesised library functions could be identified. The left-hand side of the table gives the *true positive* number of locations as identified by manual examination of the code. Values reported are split by whether they corresponded to handwritten application code (C) or inlined library function calls (L). The right-hand side gives the number of these locations successfully matched by four different versions of the discovery algorithm: ❶ no generalisation, ❷ generalisation, ❸ generalisation with nested loop corrections, and ❹ false positive testing. See Section 4.6.5 for details.

		Abinit	**Pathsample**	**NWChem**	**Darknet**	**Parboil**	**Total**	**Discovered**			
								❶	❷	❸	❹
SPMV	C		2	2			4	0	0	4	4
	L						0				
GEMM	C		1	2	1	1	5	0	1	186	186
	L	180			1		181				
GEMV	C		1	2	1		4	0	0	51	51
	L	47					47				
GER	C		3	2			5	3	5	5	5
	L						0				
AXPY	C		7	2	1		10	28	31	31	31
	L	21					21				
AXPBY	C			27			27	0	29	29	29
	L	2					2				
SCAL	C		13	2	1		16	33	36	36	36
	L	20					20				
COPY	C		5	2	1		8	75	78	78	78
	L	70					70				
DOT	C		1	2	1		4	1	4	4	4
	L						0				
SOFTMAX	C				1		1	0	0	0	0
	L						0				
RELU	C				1		1	0	1	1	1
	L						0				

Table 4.5: False positive matches identified by the discovery algorithm. Each of these cases resulted from over-generalisation of constraint descriptions (e.g. incorrectly removing constraints enforcing data dependencies), and was resolved by performing input-output example-based testing of proposed code replacements.

	Abinit	**Pathsample**	**NWChem**	**Darknet**	**Parboil**	**Total**
SCAL			2	3		5
COPY			5	2		7
DOT				1		1

replacements. The results of this comparison are shown in Figure 4.7.

On all three networks, the suggested GPU implementation substantially outperforms the OpenMP implementation (which represents the best readily-available CPU performance on an AMD processor). Speedups from 2.4× (DenseNet-201) to 9× (VGG-16) were achieved. DenseNet-201 performs smaller matrix multiplications than the other networks, and so benefits less from GPU execution due to communication overheads. These results show that the approach proposed in this chapter allows for programmers to port applications to other platforms, without having to support multiple code bases for each possible implementation.

4.6.3 Library API usage

Across the libraries evaluated, ANNOTE was able to synthesise 11 different *classes* of function. Because the libraries share common functionality, many computations are implemented multiple times (this is, of course, what drives the ability to migrate between libraries). For the sake of simplicity when reporting the results in this section, functions performing the same computation are considered together in the same class. For example, `cublas_sgemm`, `cblas_sgemm` and `clblast::Gemm<float>` are all considered together under the GEMM class.

For each of the 11 function classes that could be synthesised by ANNOTE, Table 4.4 lists the number of instances in application code where the generalised constraint matching procedure identified a match with a library function.

Some of the applications examined make extensive use of library functions. For example, Abinit links against an installed standard BLAS library, and so all the instances detected in its code are from inlined library calls. Other applications bundle their own implementations; this code is detected rather than the corresponding call sites which results in a smaller overall number of matches. Table 4.4 distinguishes between these two cases (rows C and L for original code and inlined library functions, respectively).

4.6.4 Synthesis

The time taken by Annote to correctly synthesise each library program is acceptable for its usage model; every example could be synthesised in under 4 hours on a desktop-class machine, with examples that use shorter instruction sequences taking far less time. Synthesis time was not a primary goal of the work in this chapter, as learning the behaviour of a function for the refactoring workflow suggested is a one-off task. If synthesis time were to become a bottleneck, the literature suggests several approaches to improve performance [68, 139]; implementing these is beyond the scope of this chapter.

4.6.5 Matching

Table 4.4 shows the results obtained when searching for code satisfying the generated constraints for each function class. Four different versions of the constraints were tested: those generated from a single example constraint-based program, generalised versions from multiple programs, generalised with a post-processing step, and finally with dynamic testing of replacements.

The constraints generated from a single program fail to discover many examples. Only simple, inlined library calls are consistently matched by these constraints (Abinit in Table 4.4, for example) as the inlined code is identical to the code from which constraints are generated.

Subsequently, the graph matching algorithm described in Section 3.3 was applied to generalise constraints. These constraints are more successful; many instances that were not previously matched now are (e.g. in Darknet). Some instances such as gemm and spmv were not discovered by the generalised constraints. This was due to a consistent difference between Clang's code generator and the synthesiser for nested loops. A mechanical post-processing step fixed these constraints, allowing the corresponding examples to be detected properly (gemm, gemv, spmv columns in Table 4.4).

Although these constraints generalised well, some false positive matches occur due to over-generalisation (e.g. for scal, copy in NWChem and Darknet), because of missing data dependencies in code that interleaved other work with the learned function (for example, returning to Listing 4.1, the left hand code contains such interleaved work). These false positives did not occur for every synthesised library function, and only occurred for "C" matches (i.e. original application code, rather than inlined library functions) in NWChem and Darknet. Table 4.5 summarises these false positives.

To address over-generalisation, dynamic testing of suggested replacements was performed. When a suggestion is made, the surrounding code can be isolated and tested using IO examples (similarly to the testing process for entire functions used to determine correctness of synthesised candidates). If any difference in behaviour is observed, then the suggested replacement is invalid. Applying this additional check eliminated all false positives observed when using previous versions.

The only example not to be discovered in any of the test applications was SOFTMAX; it was implemented in the application code using a common numerical trick where the input data is shifted uniformly by its maximum value. ANNOTE's synthesis is not able to learn this approach as the resulting code is not "intuitive". Fortunately, it was not a significant contribution to execution time in the programs examined.

4.6.6 Soundness

There are a number of ways in which unsound behaviour can arise when using the synthesis, generalisation and replacement suggestion tools implemented in this chapter.

Random IO examples may not capture the full range of a function's behaviour. This is a limitation shared with many approaches to synthesis; relying on observational equivalence is a common assumption made by synthesisers when no formally verifiable specification is available. Additionally, the motivation for this chapter provides a useful heuristic that observational equivalence is an acceptable assumption; because the functions examined are performance-critical kernels, they are unlikely to produce different or irregular behaviour on a sparse subset of their inputs (as doing so would almost certainly impact performance).

ANNOTE may fail to synthesise a library function at all; not all functions have behaviour that can be captured by the fragments and heuristics used for synthesis in this chapter. If this is the case, the function is simply ignored. The refactoring and replacement workflow suggested in this chapter is able to produce useful performance improvements on a number of real-world applications, despite not being able to learn every individual library function.

False positives and negatives can occur when matching constraints. The results in this chapter demonstrate that the generated constraints generalised well to detect complex examples, and that false positives can in fact be readily eliminated by dynamic testing.

While these sources of unsound behaviour can and do affect the workflow in some cases, the actual effects are not critical to the practical application of the tools described in this chapter.

4.7 Related Work

The work in this chapter relates primarily to *sketch-based* program synthesis [48]. In particular, Annote draws on the idea that a program sketch does not have to be supplied up-front by a user, but can be identified by the synthesiser performing an initial search [62]. This chapter identifies a midpoint between user-supplied sketches and complex structural filtering: by encoding informally-known properties of an oracle interface, informative sketches can be obtained through a comparatively lightweight search process.

While Annote uses a set of input-output examples to specify the correctness of potential solutions, it differs from common programming-by-example (PBE) methodologies in that it assumes examples are plentiful, cheap to generate, and potentially uninformative. In contrast, other PBE implementations often presume particular properties of their inputs [80], or apply constraints to their generation [81] in order to maximise the information gained from each example.

4.8 Conclusion

Porting existing code to exploit accelerator libraries is a challenging problem for programmers. Understanding the behaviour of existing and new libraries requires significant work on the programmer's part.

This chapter presents two main contributions to help with this API evolution: a program synthesis technique that uses vendor-supplied type annotations to infer partial control flow structure for potential solutions, and a novel workflow for API migration based on approximate graph-matching from constraint descriptions. Using this approach produced significant improvements to the performance of existing, real-world scientific applications.

Chapter 5

Improved Synthesis using Learned Probabilistic Models[1]

Chapter 4 introduced a system (Annote) capable of synthesising equivalent implementations for library functions, based on their type signature and and manual annotations describing semantic properties of their interface. While this approach allowed for the behaviour of complex library functions to be correctly modelled, it was not fully automated in its approach (annotations must be provided by an expert user for each new library targeted).

Clearly, a more general solution is desirable. Ideally, new libraries would require no up-front manual intervention to be compatible with a synthesis-based solution as described previously. To this end, this chapter describes the motivation and design of a synthesiser (Presyn) that uses *probabilistic* models to predict the most likely structure of solutions, based on observations made of an existing corpus of examples.

Because library functions are diverse, covering a wide range of problem domains, the training and evaluation sets used to train Presyn are important. To do so, a representative set of examples from the synthesis literature and existing libraries is collected and normalised to a common specification format. Collecting this corpus is a non-trivial undertaking, and represents a substantial contribution to the field.

The usefulness of synthesis as a way of driving API migration problems of this kind is predicated on how many functions can actually be synthesised successfully. To evaluate Presyn with respect to this metric, Section 5.7 compiles an extensive set of synthesis benchmarks from existing literature and library source code. A methodology that allows for the fair preparation of synthesis examples and comparison of results for different synthesisers on this dataset is prepared. Then, in Section 5.8, four leading program synthesisers (as well as Annote) are evaluated against Presyn following this

[1] This chapter is based on published research in Collie et al. [5].

methodology.

5.1 Black-Box Oracle Guided Synthesis

As previously identified in Chapter 4, modelling and understanding the behaviour of software components is a key issue in software engineering [140]. While a formal model (for example, source code or a full specification) of such a component may be available, this is often not the case in practice. Components are frequently supplied as low-level binaries, network services or hardware [96]. In some cases, partial insights about a component's behaviour may be suggested by its interface, but such examples are in the minority. To develop tools capable of modelling *all* software components, a way of understanding these "black-box" components is necessary.

For the sake of wide applicability, this chapter makes the minimal possible set of assumptions of black-box component interfaces. The only requirements are that it exists in an executable form, and has a known type signature; a significant relaxation of constraints from Chapter 4. The concrete mechanism by which the component is executed is intentionally left abstract; while the tools implemented in this chapter use the C foreign function interface to load and call shared libraries, other mechanisms such as direct memory access or network requests could be substituted with no change to the underlying problem specification.

This set of assumptions instantiates an *oracle-guided* synthesis problem belonging to the most restricted class described by Jha and Seshia [50]. The aim of synthesis is to produce a program that behaves equivalently to a target oracle, which performs an unknown computation. No knowledge of the oracle's internal structure is available, but capturing its behaviour in the form of input-output pairs (IO examples) is inexpensive. Richer oracle-guided problem contexts allow the oracle to provide minimal counterexamples or formal verification of possible solutions; none of these are available for black-box oracles.

The problem of synthesis from a black-box oracle bears some similarity to programming by example, which has received considerable interest from industry [141]. Here, the aim is to synthesise a program from handwritten, user-provided examples of correct behaviour [80, 142]. Black-box synthesis cannot rely on any particular structure of the examples it consumes (for example, Leung et al. [143] require a user to provide increasingly specific disambiguating examples to guide synthesis).

5.2 Approach

The available information in the context of a black-box oracle-guided synthesis problem is limited: only the type signature and a collection of input-output examples are known. To decrease the size of the space of programs that must be searched through to find a correct solution, a relevant interpretation of either or both of these must be obtained.

Beyond their use in determining correctness for a potential solution, some previous work has aimed to introspect on the structure of input-output examples to determine likely solution structure. DeepCoder [79] uses a neural network to learn *structural* features of input-output examples in a functional language (for example, that a particular target oracle always returns a shorter list of values than it is passed as an argument). However, generalising this type of neural approach to environments where fewer structural properties are available has proved challenging, especially when examples are larger than a few elements in size [84].

However, Chapter 4 demonstrated that the type signature of a component can often be interpreted by a user to provide precise semantic information about the behaviour of that component. While the manual annotation process used in that chapter is not generally available for black-box oracles, the type signature is. If the underlying features of a type signature that map onto program structure could be learned and predicted, then some aspect of the annotation process and heuristic mapping could be replicated without user intervention.

This insight yields the approach developed in this chapter: to build an annotated ground-truth dataset of synthesis problems and their solutions, then to train a model that can *predict* a likely solution to a problem based on its type signature.

5.3 Models

Presyn uses two successive probabilistic models to predict program structure based on the type signature from the synthesis problem. This section introduces the features learned predicted by these models, and explains the training process and internal working of each one.

5.3.1 Features

Chapter 4 uses program *sketches* to describe partial structures that are likely to appear in the solution to a synthesis problem. Each correct solution corresponds to a composition of sketch fragments (using the sketch notation introduced in Section 3.5). For example, a problem containing a nested loop structure might use the composition `loop` ∘ `loop` for an appropriate fragment `loop`.

These compositions of fragments can be easily interpreted as a feature vector for the relevant synthesis problem: if the vector of fragments can be predicted accurately, then a great deal of the synthesis problem can be solved cheaply (as is demonstrated in Chapter 4).

Because each synthesis problem permits a different set of valid program fragments (for example, `loop(x)` is only valid when the problem signature contains a parameter `x`), the training and prediction steps for the two models deal with both *fragment classes* and *fragments* at different times. The distinction between the two will be made clear when appropriate.

5.3.1.1 Definitions

This section briefly states the formal definitions for the inputs to and outputs from the models described in this section. The synthesis interpretation of these will be given fully in later sections.

5.3.1.1.1 Inputs The input feature vector $\overline{\mathbf{x}}$ to both models is the type signature for the problem, encoded as a list of pairs:

$$\overline{\mathbf{x}} \triangleq [(t_0, p_0), \dots] \tag{5.1}$$

where each $t_i \in \mathbb{N}$ identifies the *base* type of the i^{th} parameter to the function (e.g. the base type of both `int` and `int**` is `int`), and each $p_i \in \{0, 1\}$ identifies whether the type is a pointer. This encoding is a simple mapping from the subset of the C type system defined in Chapter 4 to a format that can be consumed by a model without processing character-level information.

Parameter names are not used by this encoding; while the name of a parameter might encode semantic information (e.g. a parameter `int size` is marginally more likely to represent a loop bound than `int a`), to preserve the minimality of the problem context any names are discarded when encoding a signature. Without names, parameters are uniquely identified by their index. Where it aids clarity, names may be added back into examples.

5.3.1.1.2 Outputs Two output features $\mathbf{y}$ and p_m are defined; one for each of the models used by Presyn. The first is an unordered set of *fragment classes*:

$$\mathbf{y} \triangleq \{\mathtt{c_i}, \dots\} \tag{5.2}$$

where each $\mathtt{c_i}$ is a unique internal identifier for a fragment class.

The second is a probability mass function supported on the set of pairs of concrete fragments from the set F:

$$p_m : (F \times F) \rightarrow [0,1] \tag{5.3}$$

5.3.2 IID

The first step taken by Presyn is to predict which classes of fragment are likely to appear (as any possible instantiation) in a correctly synthesised solution.

Suppose $f_0, f_1, \ldots, f_n$ is an ordered sequence of fragments that when composed, produces a correct solution for a synthesis problem. The aim of Presyn's first model, IID, is to predict an initial set of fragments $\mathbf{F_0}$ that matches $\{f_0, f_1, \ldots, f_n\}$ as accurately as possible (without considering ordering). This model is given the name IID as each fragment in $\mathbf{F_0}$ is presumed to be equally likely to occur in a synthesised solution, independently of the presence of each other fragment (that is, they are independent and identically distributed). By way of comparison with Chapter 4, Annote performs a similar prediction task *deterministically*, using rules matched against property annotations.

The deterministic approach used by Annote, while easy to interpret and precise on the small set of performance-sensitive functions targeted for evaluation, is limited in two key ways. Firstly, and more generally, manual annotation of interface properties is required to consider a new function, and new heuristics are similarly required to add new fragment templates. Secondly, specifically motivating the development of IID is the issue that Annote may not select precisely the correct set of initial fragments: if its heuristics do not model the required semantics, then a solution can never be synthesised.

Presyn retains a single, limited aspect of Annote's deterministic approach to fragment selection, but does so through *observations* of input-output examples rather than through type signature annotations. IID provides a *probabilistic* prediction of the set $\{f_0, f_1, \ldots, f_n\}$. These two sets of fragments are written $\mathbf{F_S}$ and $\mathbf{F_P}$ below (for **S**emantic and **P**robabilistic, respectively).

Fragment Semantics While attempting to determine the initial fragment set using entirely deterministic methods does not scale well, some properties can be inferred cheaply by making observations of the behaviour of the target function, based on the input-output examples generated for testing. For example, if a memory region is written to in any of the observed examples, then that region represents an output reference parameter. If such parameters are present, then the set of fragments capable of performing output is collected as $\mathbf{F_S}$.

Classification Model As well as semantic knowledge, a classification model is used to determine inclusion in $\mathbf{F_0}$ for non-output fragments. To do so, a random forest model is trained to predict inclusion-exclusion classifications for each fragment class, given a function type signature as input. This trained model provides a decision function P that determines fragment inclusion in $\mathbf{F_0}$; producing such a model requires a small corpus of training data (type signatures and $\mathbf{F_0}$ for a set of successfully synthesised programs).

Writing $\mathbf{F_P}$ for the set of fragments satisfying the trained decision function P, the final prediction for a particular synthesis problem combines these two components:

$$\mathbf{F_0} \triangleq \mathbf{F_S} \cup \mathbf{F_P} \tag{5.4}$$

It is clear that it is safe for the prediction of $\mathbf{F_0}$ to over-approximate the true initial set: if additional irrelevant fragments are present, synthesis will be slower but will still eventually discover a correct solution. However, $\mathbf{F_0}$ may in fact be an *under*-approximation. If this is the case, then similarly to ANNOTE, correct solutions may never be considered. This problem is addressed by the second complementary model in PRESYN's synthesis process: Markov.

5.3.3 Markov

The approximate set of likely initial fragments $\mathbf{F_0}$ (as predicted by the IID model) represents the starting point for synthesis. However, for this set to be useful for synthesis, two issues must be resolved.

Firstly, the order in which the fragments are to be composed should be determined. ANNOTE is able to revert to an enumerative process here, as the number of potentially matching rules for a given synthesis problem is small, preventing intractable exponential increases in the size of the solution space. No such guarantee can be made in the case of PRESYN, so correctly ordering fragment compositions is an important part of the synthesis process.

Secondly, because IID can under-approximate in its prediction of $\mathbf{F_0}$, the remaining parts of the synthesis process should be robust to this. That is, fragments not included in $\mathbf{F_0}$ should be considered at some point, rather than being discarded entirely.

To predict fragment *orderings*, a different assumption on the structure of the same training data is required. PRESYN's second model, Markov, observes bigrams of fragment occurrences under a Markovian assumption. That is, the sequence of fragments forming a correctly synthesised solution is treated as if it had in fact been generated by a Markov model. The following definitions are used to define the underlying structure of such a model, so that the transition probabilities can then be learned.

First, auxiliary fragments f_{start}, f_{end} are defined. In the sketch language semantics defined in Section 3.5, both act as the identity under composition with any other fragment. Then, a weight function w is defined such that $w(f, f')$ represents the number of occurrences of the composition $f \circ f'$ in a particular training set.

Given these definitions, ordered sequences of fragments (representing compositions) can be sampled as follows. First, define:

$$s(f) \triangleq \sum_{f' \in \mathbf{F}} w(f, f') \tag{5.5}$$

Then, starting from the auxiliary fragment f_{start}, fragments can be sampled from the conditional probability distribution:

$$\mathbb{P}_w(f_n | f_{n-1}) = \frac{w(f_{n-1}, f_n)}{s(f_{n-1})} \tag{5.6}$$

That is, the fragment f_n is sampled after f_{n-1} with probability proportional to the ratio of observed compositions $f_{n-1} \circ f_n$ to all occurrences of f_{n-1}. This distribution resolves the first issue with IID identified above (ordering), but does not yet address the second (under-approximation). To do so, an augmented weight function w' is defined:

$$\begin{aligned} w'(f_i, f_j) &\triangleq w(f_i, f_j) \text{ if } f_j \in \mathbf{F_0} \\ &\triangleq 0 \text{ otherwise} \end{aligned} \tag{5.7}$$

The augmented weight function assigns non-zero weights only when the second fragment f_j is contained in the set predicted by IID (i.e. the "likely" fragments are assigned a weight, while "unlikely" ones are not). Then, define s' and $\mathbb{P}'_w$ analogously to s and $\mathbb{P}_w$:

$$s'(f) \triangleq \sum_{f' \in \mathbf{F}} w'(f, f') \tag{5.8}$$

$$\mathbb{P}'_w(f_n | f_{n-1}) = \frac{w'(f_{n-1}, f_n)}{s'(f_{n-1})} \tag{5.9}$$

These definitions reflect an additional prior on predicted sequences of fragment compositions: what is the probability that fragment f_{n-1} is followed by fragment f_n, given that f_n was classified as "likely" by the IID model?

Finally, a variable weight can be given to the classifications made by IID (i.e. to reflect the level of certainty that it has predicted the set $\mathbf{F_0}$ correctly). This parameter $b \in [0, 1]$ allows a final conditional probability to be defined:

$$\mathbb{P}(f_n | f_{n-1}) = b\mathbb{P}'_w(f_n | f_{n-1}) + (1 - b)\mathbb{P}_w(f_n | f_{n-1}) \tag{5.10}$$

Higher values of b weight Markov's predictions more heavily towards the "likely" fragments selected by IID (i.e. $\mathbb{P}'_w$ is given more weight). For $b = 1$, unlikely fragments are never selected, and for $b = 0$, IID is not considered at all. Through trial and error, $b = 0.8$ was selected for Presyn as implemented.

A sequence of fragments $f_0 \circ \cdots \circ f_n$ can be generated by sampling transitions from the Markov model induced by this distribution until f_{end} is sampled, or to a fixed maximum length. The sequence is then folded over the fragment composition operator $\circ$ to produce a sketch.

5.3.4 Training

Both models (IID and Markov) are trained using the same annotated dataset. For each synthesis problem in the dataset, its type signature and a sequence of fragments that leads to a correct solution are provided as training data. The method by which this training data is collected in practice is given in Section 5.6; this section describes the training process once a suitable corpus has been assembled.

IID: A random forest ensemble of 64 trees and a maximum decision depth of 5 is trained on the dataset, with internal splits at ≥ 2 nodes, and leaves containing ≥ 1 node. To validate the accuracy of the trained classifier, a 20% subset of the data is left out from the training set.

Markov: The weight functions w and w' can be trained by direct observation of the training dataset; no further processing other than prepending f_{start} and appending f_{end} to the fragment sequences is required. Once this is done, bigram sequences can be directly counted and used to build the weight functions.

5.4 Synthesis

Presyn retains the core low-level synthesis loop introduced by Annote in Chapter 4, using the program manipulation primitives from Section 3.6.1 to refine sketches into concrete programs. This core synthesis loop is summarised in Algorithm 4.

The novel improvements made by Presyn, as compared to Annote, relate to the automatic identification of plausible program *sketches* for a synthesis problem based only on its interface. Once such a sketch is identified, Presyn uses the same procedure as Annote to search for a correct instantiation of that sketch, and to test it for observational equivalence to the original synthesis oracle.

In brief summary, the process followed by Presyn (and presented in Algorithm 4) is as follows. First, the two models (IID and Markov) are trained on an appropriate

Algorithm 4 Presyn's core synthesis loop. This loop assumes that IID and Markov have been trained, and that the set of initial fragments $\mathbf{F_0}$ has been predicted and used to instantiate Markov. Values are sampled from the live set using a proximity-weighted heuristic.

```
function Synthesise(Markov, $examples$)
    $sketch$ ← output sampled from Markov
    loop
        for each $hole$ in $sketch.holes$ do
            recompute live values and dependencies
            $live$ ← proximity-sorted live values at $hole$
            $v$ ← sample geometrically from $live$
            if $hole.type$ is compatible with $v.type$ then
                Rauw-NT($hole, v$)
            end if
        end for
        $program$ ← $sketch.compile()$
        if $program$ satisfies all $examples$ then
            return $program$
        end if
    end loop
end function
```

representative dataset of synthesis problems; this is a one-off process and does not need to be repeated for each new problem. Then, the type signature for the current problem is used by IID to predict the set of initial fragments $\mathbf{F_0}$. This predicted set is combined with the set of global bigram observations w to produce the problem-specific Markov model.

Given the problem-specific Markov model, a sketch is produced by sampling fragments until f_{end} or a fixed size bound is reached. The symbolic holes in this sketch are filled iteratively using the same proximity and type prioritisation introduced in Section 4.4.3 to produce an executable program. This program is executed and tested for observational equivalence using the same input-output example generation methodology described in Section 4.4.4.

If a sketch does not yield a correct solution after a set number of candidates (for Presyn as implemented, 1000) have been generated from it, then it is discarded and a new sketch is sampled from Markov, restarting the process.

5.5 Comparing Synthesisers

Benchmarking program synthesisers against each other fairly while allowing for differences in specification and expression is a challenging problem [46, 47]. To fairly evaluate Presyn's performance, a set of comparable synthesisers must first be identified, and their relative strengths, weaknesses and biases characterised. This section describes how a set of 4 leading synthesisers (along with Annote, as implemented in Chapter 4) was identified, as well as an overview of their respective target languages and problem domains. Finally, an examination of how synthesis problems are prepared for each implementation is carried out.

5.5.1 Survey

To identify synthesisers against which Presyn could be evaluated, a survey of the recent program synthesis literature was carried out. Synthesis research papers were evaluated with respect to the following criteria:

Domain Fit For a synthesiser to be evaluated against Presyn, the programs it synthesises need to be in broadly similar target domains. That is, the programs should be *general-purpose* programs that potentially contain control flow (such as conditionals or loops).

This requirement rules out a wide class of domain-specific synthesisers such as Cosette [144] or Nonograms [54] that produce programs in a limited, often Turing-incomplete, target DSL. In these contexts, techniques such as SMT solvers can often be deployed to handle difficult exhaustive-search procedures; these solvers do not yet scale to modelling general control flow without extensive user input.

Additionally, synthesisers such as Helium [96] that synthesise programs in a general-purpose target language, but require significant assumptions to be made about the semantics of the target program are also ruled out by this criterion. For example, Helium requires that the programs it synthesises specifically represent image-processing kernels.

Leading Performance The synthesisers chosen should represent leading performance on their respective target domain; there is no sense in evaluating implementations whose performance has been substantially superseded on their own set of benchmarks.

Implementation Variety To ensure a broad comparison, different synthesis strategies should be used as far as possible by the implementations chosen for evaluation.

By doing so, relative advantages and shortcomings of Presyn's approach can be observed more clearly than they could be if it were benchmarked solely against synthesisers using only a single style of implementation.

Artefact Availability Evaluating the set of chosen synthesisers fairly against Presyn requires that each implementation provides a runnable artefact. Without this, only the original results quoted in each paper could be used for evaluation, preventing a fair cross-evaluation.

5.5.2 Implementations

Following the criteria above, a comprehensive review of the synthesis literature yielded the following 4 implementations to evaluate Presyn against:

SketchAdapt is a neural synthesiser that aims to strike a balance between *generative* and *search-based* approaches [84]. Where its neural models identify a program close to an existing one from its training dataset, it is able to directly generate program structure from an internal generative model; doing so is more efficient than performing exhaustive searches. However, its novelty lies in its ability to combine this approach with traditional search for examples that are less similar to the training set (and, indeed, to interpolate between the two, picking the correct point at which to abandon the generative approach). It represents the state-of-the-art synthesis performance on general programs achieved by a neural model.

Makespeare uses a genetic algorithm over an abstracted model of x86 assembly to synthesise programs containing loops [72]. Its novelty over previous approaches is a novel hill-climbing algorithm that delays the acceptance of intermediate programs; doing so provides an evolutionary advantage to programs that generalise successfully across a set of examples. It targets constrained assembly-like languages that can be translated to executable programs (x86 assembly, or the fictional tis-100 computer-game language). Synthesis problems are specified by providing the initial and desired final abstract machine states (i.e. registers and memory).

Simpl synthesises imperative programs written in a small C-like language [70]. This language permits looping control flow structures; the key innovation developed for Simpl is the use of static analysis techniques to prune the search space of candidate programs. Doing so ensures that dead-end candidates are not explored unnecessarily (for example, asserting that a looping program with an incorrect upper bound could *never* write to a location that has been modified between input and output).

λ^2 targets a polymorphic λ-calculus with recursive algebraic data types (for example, functional lists and trees) [77]. By constructing and attempting to refute type-correct hypotheses about the structure of candidate programs (e.g. can the `map` combinator in a given subexpression ever satisfy the problem's constraints?), λ^2 can quickly narrow down its search space. Synthesised programs are subject to a structural cost function that ensures they are the *minimal* correct program with respect to the particular set of input-output examples used.

Additionally, Annote (Chapter 4) is considered as a baseline implementation to evaluate Presyn against. By doing so, the advantage gained by using flexible probabilistic predictions over a set of hand-coded heuristics can be quantified.

5.5.3 Synthesizer Help

All of the synthesisers listed above can be loosely considered as black-box, oracle-driven synthesis, as each of them specifies *correctness* for a particular synthesis problem as observational equivalence over a set of input-output examples (rather than through formally-verified equivalence to a specification).

However, each synthesiser (including, to an extent, Presyn) requires additional *user help* to search for and construct solutions. This help is in the form of non-correctness specifications that provide hints or direction to the structure of a potential solution. These additional specifications cannot be used to determine whether or not a solution is correct, but do allow the search procedure to reach that solution more efficiently. Additionally, it is worth distinguishing between help at the dataset level and help at the problem level. All synthesisers encode some kind of heuristic about the structure of solutions into their search process; these help to guide search but are different from (and easier to provide than) help provided for *individual* problems.

To fairly evaluate synthesisers against each other on a shared dataset, it is important to clearly state the scope of this additional help for each synthesiser; doing so allows for their underlying assumptions and models to be made clear. In Section 5.5, a methodology for comparing these synthesisers both with and without the additional assistance they make use of is presented.

The additional help used by each synthesiser evaluated in this chapter is as follows:

Annote uses a library of hand-coded heuristic rules and program sketches at the dataset level, and requires a set of semantic property annotations to be provided by the user for each new synthesis problem.

Presyn requires a model pre-training step (to instantiate IID and Markov on a set of representative problems), but does not require any additional user assistance for new synthesis problems.

SketchAdapt also requires a model training step before synthesis begins (with far more data required than for Presyn, owing to its use of neural network models). It does not require any additional assistance for new problems.

Makespeare requires that a particular set of its abstract machine's registers be instantiated with relevant values (for example, that a loop upper bound should be placed in one register). Additionally, the *structure* of the input-output examples used to specify a problem must be carefully chosen to encourage the genetic algorithm to converge on a fully general solution.

Simpl relies on the user providing (by hand) a partial program sketch for each new problem, as well as a set of relevant integer constants that appear in computation.

λ^2 uses a standard library of functions and combinators (essentially acting as program sketches in the functional context). Additionally, when supplying input-output examples for a new problem, the user is required to explicitly cover the recursive base-case for that problem.

5.6 Evaluation Dataset

Evaluating program synthesisers against each other fairly is a long-standing goal of the community. However, the wide spectrum of different target languages, synthesis styles, and user assistance given makes it extremely difficult to do so in practice. Different implicit assumptions, search procedures, and problem inputs can mean that an easy problem for one synthesiser might be pathologically hard for another. This challenge has not escaped the community; attempts to compare even similar synthesisers on common tasks are fraught with subtle problems. For example, one recent paper [46] observes:

> A more salient observation is the difficulty of comparing these methods due to drastically different intended applications, despite the common goal of program synthesis. *(Pantridge et al. [46], p. 1)*

> The comparison presented in this paper highlights the difficulty of using benchmark problems to compare IPS [Inductive Program Synthesis] methods . . . the most conclusive finding that has come out of this comparison is that not all

Table 5.1: Groups of synthesis benchmark problems

Group	N	Description
makespeare	11	Problems that require loops to manipulate arrays of integers in place. The full set of benchmarks from [72] are used, modulo those adapted from [70].
simpl-int	15	Arithmetic manipulations of integer values, requiring loops and data-dependent control flow; the full set of integer benchmarks from [70] are used.
simpl-array	12	Problems stated over integer arrays, with different styles required (e.g. pairwise iteration, reductions and elementwise computation); the full set of array benchmarks from [70] are used.
λ^2	8	Singly-nested integer linked-list manipulation problems from [77], restated for other synthesisers as array problems.
SketchAdapt	10	A series of generated list problems, taken as representative samples from the 500 program evaluation file presented by [84].
string	16	The C standard library's string processing functions [145]. Impure functions such as `strtok` are removed.
mathfu	15	Vector-scalar and vector-vector mathematical functions from the Mathfu [146] library.
blas	4	Matrix-vector linear algebra functions from the BLAS [147] standard as synthesised in [1], disregarding functions such as `spmv` for which extensive assistance was required.
dsp	21	Vector- and matrix-based signal processing functions from the TI signal processing library [148], adapted for platform portability and removing functions with requirements for extensive constant data such as filter taps.

> IPS systems can be applied to the same problems. This makes comparison extremely difficult. *(Pantridge et al. [46], p. 7)*

This section presents a methodology for assembling a set of benchmark problems that can be used to fairly, consistently and reproducibly evaluate the program synthesisers listed in Section 5.5. While the methodology presented is (at least partially) specific to the synthesisers listed previously, it is hoped that the broad goals and techniques used to prepare the common set of benchmarks can be generalised to more general applications in the future.

First, a common language subset is established such that all the synthesisers under consideration are able to interpret every collected benchmark problem. Then, the methods by which these problems should be obtained are considered (should they be curated by hand, or generated randomly somehow?). With that in mind, a concrete set of examples is described and listed in detail. Finally, the procedure required to elaborate this list into equivalent inputs for each synthesiser is described.

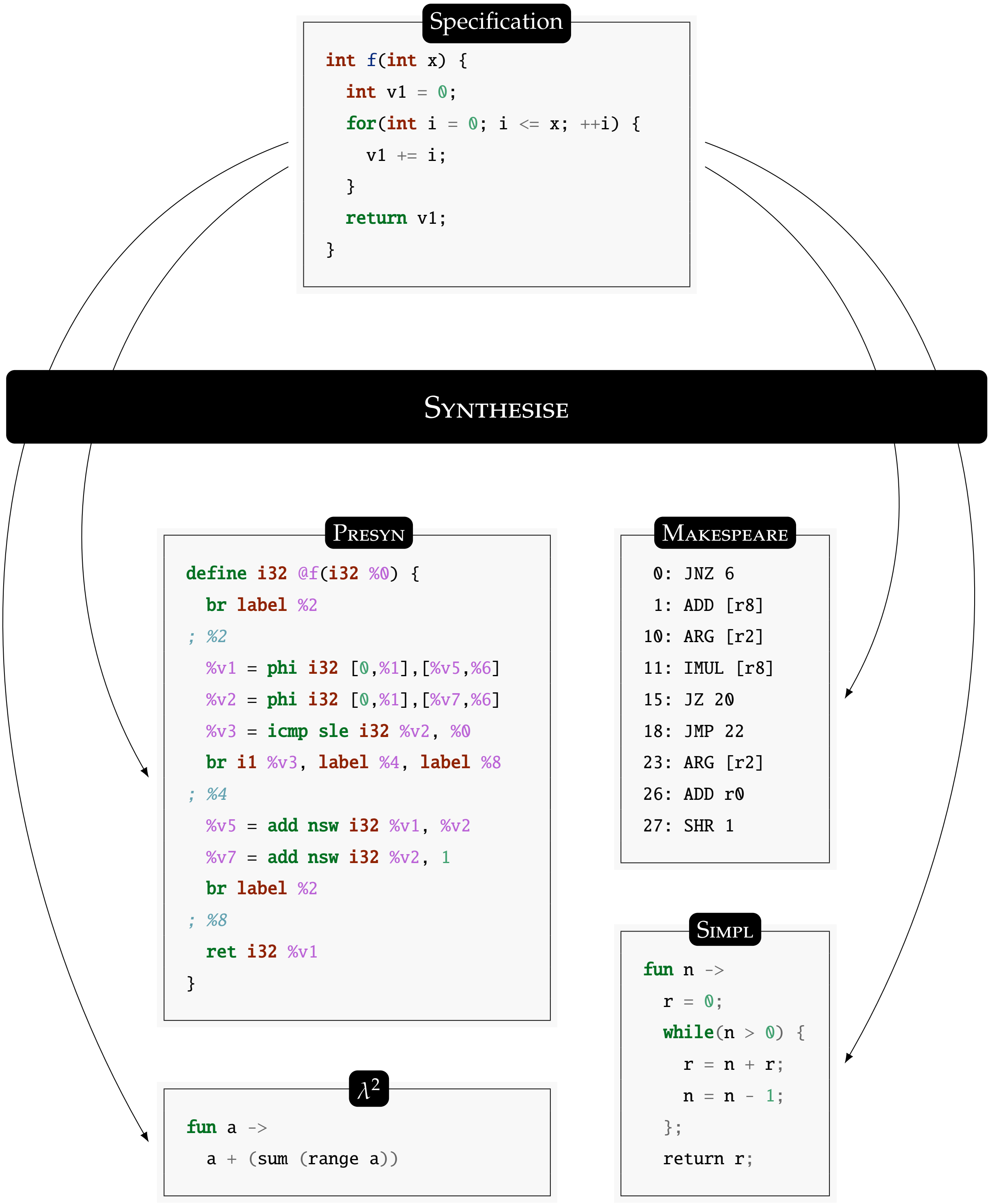
5.6.1 A Common Language

As stated above by Pantridge et al. [46], not all synthesisers can be applied to the same set of problems. As a result, for fair cross-evaluation of synthesisers, it is necessary to identify the subset of their features that are shared and can be meaningfully compared.

For the synthesisers evaluated in this chapter (Presyn, Annote, SketchAdapt, Makespeare, Simpl and λ^2), the shared set of features can be stated concisely as *numerical programs operating over scalars, lists, or arrays of values, where each value is either integral or floating-point*.

This definition intentionally allows for some difference of expression between synthesisers. For example, λ^2 implements lists of values as a recursive algebraic data type, while Makespeare operates over a region of memory in an abstract machine. The intent here is to allow for comparisons where two synthesisers are able to produce the same *abstract* computation under their own domain interpretation. For example, Figure 5.1 shows a solution produced by several synthesisers to the same synthesis problem.

While restricting the set of synthesis features under comparison does entail a disadvantage for some implementations (e.g. λ^2 is able to synthesise programs operating on more complex types such as trees, and SketchAdapt can be parameterised by custom DSL operations) at the expense of others, it provides a consistent way to compare results across all implementations. Making this assumption explicit means that the synthesis results on this set of benchmarks (see Section 5.8) can be fairly interpreted.



```
int f(int x) {
  int v1 = 0;
  for(int i = 0; i <= x; ++i) {
    v1 += i;
  }
  return v1;
}
```

```
define i32 @f(i32 %0) {
  br label %2
; %2
  %v1 = phi i32 [0,%1],[%v5,%6]
  %v2 = phi i32 [0,%1],[%v7,%6]
  %v3 = icmp sle i32 %v2, %0
  br i1 %v3, label %4, label %8
; %4
  %v5 = add nsw i32 %v1, %v2
  %v7 = add nsw i32 %v2, 1
  br label %2
; %8
  ret i32 %v1
}
```

```
 0: JNZ 6
 1: ADD [r8]
10: ARG [r2]
11: IMUL [r8]
15: JZ 20
18: JMP 22
23: ARG [r2]
26: ADD r0
27: SHR 1
```

```
fun a ->
  a + (sum (range a))
```

```
fun n ->
  r = 0;
  while(n > 0) {
    r = n + r;
    n = n - 1;
  };
  return r;
```

Figure 5.1: From a common problem specification (prepared as C code as described in Section 5.6.3.3), different synthesisers each produce a solution in their own output language. The specification at the top of the figure is taken from the Simpl [70] benchmark set, and represents the computation $\sum_{i=0}^{N} i$ (i.e. the sum of the range $[0, N]$). Not shown in this figure is the intermediate generation of implementation-specific inputs based on the common specification (see Figure 5.2). For brevity, outputs from Annote and SketchAdapt are not shown.

5.6.2 Methodology

Two broad approaches to collecting a dataset of synthesis problems are viable: random syntax-guided generation of examples using the sketch language described in Section 3.5, or manually sourcing and annotating examples from existing sources.

5.6.2.1 Random Generation

Using randomly or semi-randomly generated programs as a synthesis benchmark dataset is appealing from the perspective of automation; existing work has demonstrated that models for complex prediction tasks can be trained effectively using random examples. For example, Cummins et al. [149] achieve state-of-the-art performance on a well-studied classification task (determining whether an OpenCL kernel should be scheduled on the CPU or GPU for optimal performance) by synthesising large numbers of OpenCL programs using a recurrent neural network. It is plausible that a similar process could be used to generate useful, representative synthesis problems for cross-evaluation.

For such randomly generated examples to be useful, it is important for them to (in aggregate) demonstrate sufficient variability, and to cover the space of synthesiser features effectively. The set of benchmark examples produced by random generation should be sufficiently interpretable that this property is practical to verify.

5.6.2.2 Manual Sourcing

Alternatively, benchmark examples can be collected by a process of manual curation: a human expert decides what examples are representative of the relevant problem domain, and collects these examples into a dataset. Doing so clearly requires more *marginal* work per example collected (though the up-front work to produce a valid random generation methodology may be less), but benefits from increased interpretability.

Collecting examples in this way requires that such example benchmark problems do in fact exist, and can be readily identified and collected. For some problems, this may not be the case (for example, some of the libraries used to evaluate Annote are closed-source). Additionally, while ensuring that the underlying generation methodology for random examples is free of bias is important, it is also so for an expert curator: choosing examples manually can equally introduce bias into the dataset.

5.6.2.3 Conclusion

For the evaluation of Presyn described in this section, manual collection of synthesis benchmark examples was identified as the most suitable methodology. The most important reason for doing so was to match the original motivation for this thesis: learning functions that behave compatibly with *existing* library functions. No compelling method for randomly generating reliably executable library code was identified, and even state-of-the-art automated collection of real-world library code cannot yet make that code executable (as opposed to simply compilable) [150]. Additionally, curating examples allows for a more direct comparison with other program synthesisers: inspection of their respective evaluation datasets allows for Presyn's dataset to cover the same ground more easily.

5.6.3 Sourcing Examples

This section describes how a large set of representative synthesis benchmark examples was collected to evaluate Presyn against the other synthesisers listed in Section 5.7. Two primary sources for these examples were considered: the existing benchmark datasets used for existing synthesisers, as well as functions from existing, well-known software libraries (similarly to Chapter 4, but without the specific goal of identifying performance bottlenecks).

5.6.3.1 Benchmarks

Each of the synthesisers selected to evaluate Presyn against has a set of benchmark problems on which they were originally evaluated. Typically, these problems are in large part taken from previous work, which is useful for cross-evaluation and comparison of results across papers. Unfortunately, the reuse of benchmarks is more *vertical* than *horizontal*. That is, implementations tend to reuse benchmarks from previous similar implementations that they supersede or improve on; it is far less common to adopt benchmarks from conceptually different synthesisers or competing parallel implementations.[2]

To address this issue, the set of benchmarks used to evaluate Presyn is partially derived from each competing synthesiser's benchmarks. From these sets, duplicate problems were removed (for example, Makespeare is already partially evaluated on a subset of Simpl's benchmarks), as were problems inexpressible in the common synthesis language described above.

The result of this process is a set of 56 unique problems (listed and described in the top half of Table 5.1), each of which was originally used as a benchmarking example for a competing synthesiser.

[2] There are, of course, a number of places where this is not the case. For example, the SyGuS research project aims to standardise competition among syntax-guided synthesisers [66]. However, "in the wild" it is more often the case that new techniques or implementations are tested on a fresh set of benchmarks.

5.6.3.2 Black-Box Components

To provide an expanded set of benchmarks that support the primary goal of this thesis (modelling and understanding the behaviour of black-box components through synthesis), the initial set of synthesis benchmark problems was augmented with additional problems derived from real-world software libraries.

In Chapter 4, ANNOTE is used to synthesise implementations of 11 functions shared between a set of optimised mathematical libraries such as Intel MKL [28] or Nvidia cuBLAS [124]. The evaluation in that chapter focuses on 11 specific performance bottlenecks. To broaden the evaluation of PRESYN's synthesis abilities, a larger set of similar library functions was collected (from the Mathfu mathematical library [146], as well as BLAS functions [147] and the TI signal processing library [148]). Additionally, given its popularity as a synthesis evaluation domain [81, 142, 151], string processing functions from the C standard library [145] were also included.

After de-duplication and filtering for compatibility, these libraries also yielded 56 unique synthesis problems (listed by group in the bottom half of Table 5.1). Each of these represents a real library function used in production application code.

5.6.3.3 Obtaining Code

For each of the 112 unique problems collected, a reference implementation that can be used to collect input-output examples is required. PRESYN uses the same interface as ANNOTE to do so (dynamically loading symbols from a shared library). For the benchmarks corresponding to existing library functions, preparing a reference implementation simply requires a wrapper function that calls into the existing library.[3]

For the benchmarks derived from existing synthesiser evaluations (excluding SKETCHADAPT), no such existing code was available. To create reference implementations for these functions, the English descriptions from their original papers were used to create implementations by hand. For example, the paper describing SIMPL contains descriptions such as:

> 7. Given n and m, return $\sum_{i=n}^{m} i$.
>
> 8. Given n and m, return $\prod_{i=n}^{m} i$.
>
> 20. Multiply two arrays of same length into one array.
>
> *(So and Oh [70], p. 377)*

[3] For the TI DSP library, the existing library distribution is platform-specific. Reference C code from the library documentation was collected manually and compiled into a library to present the same scenario as the other sources.

This process of re-implementation was simple, and no descriptions proved difficult to interpret. All of the reference implementations were independently verified by Jackson Woodruff, a co-author of Collie et al. [5] to ensure that they matched the intent of the original paper. For SketchAdapt, instead of English descriptions, training examples written in an internal DSL were translated into C to produce reference implementations.

Many of the example problems (as originally written) entail floating-point operations. However, some of the synthesisers under evaluation (Simpl, Makespeare) only target integer operations. This limitation is somewhat artificial (it is easier to demonstrate synthesis concepts over a domain with only one primitive type); to compensate, integer versions of these problems are added to the dataset in parallel, representing the same abstract computation.

At this point, every example in the collected dataset can be dynamically loaded and executed by a test harness to produce input-output examples. A manifest file lists the names and type signatures of each example so that they can be correctly loaded and called.

5.6.4 Model Training

In Section 5.3.4, the training procedure for the models IID and Markov was described abstractly (i.e. without reference to a particular dataset). This section explains how a concrete training dataset was produced from the set of evaluation problems described previously,

The example problems obtained from the sources listed above require annotation; the original programs are not intended for inter-operation with Presyn, and so are not annotated with the set of program fragments used to synthesise them. To finalise the training set, these annotations must be added by hand.

In most cases, the fragments required to synthesise a particular example were obvious from that example's code; the high-level control flow represented by fragments almost always maps well onto the original structure of the code.

An additional construction step could be performed to ensure that the added annotations were in fact correct. To do so, an interactive tool was developed that allows an expert user to *act* as if they were the synthesiser. Instead of fragments being predicted by two probabilistic models, and instructions enumerated by filling typed holes, the user selects the correct choices from the set considered by the synthesiser. Because this process is somewhat labour-intensive, only those examples where the correct annotation was ambiguous or unclear were annotated using this procedure.

5.7 Experimental Setup

Sections 5.6 and 5.7 give details of a set of competing synthesisers and synthesis benchmarks, respectively. This section describes the experimental methodology by which these synthesisers can be fairly compared against each other on that set of benchmarks.

5.7.1 Problem Preparation

The evaluation dataset described previously comprises a single shared library with a dynamically-loadable symbol for each function. A test harness loads these symbols, randomly generates input values, and observes the corresponding outputs.

For each synthesiser, the generated input-output values must then be converted into the correct input format (Presyn and Annote are integrated directly with the test harness and do not require additional preparation). A brief description of each of these is:

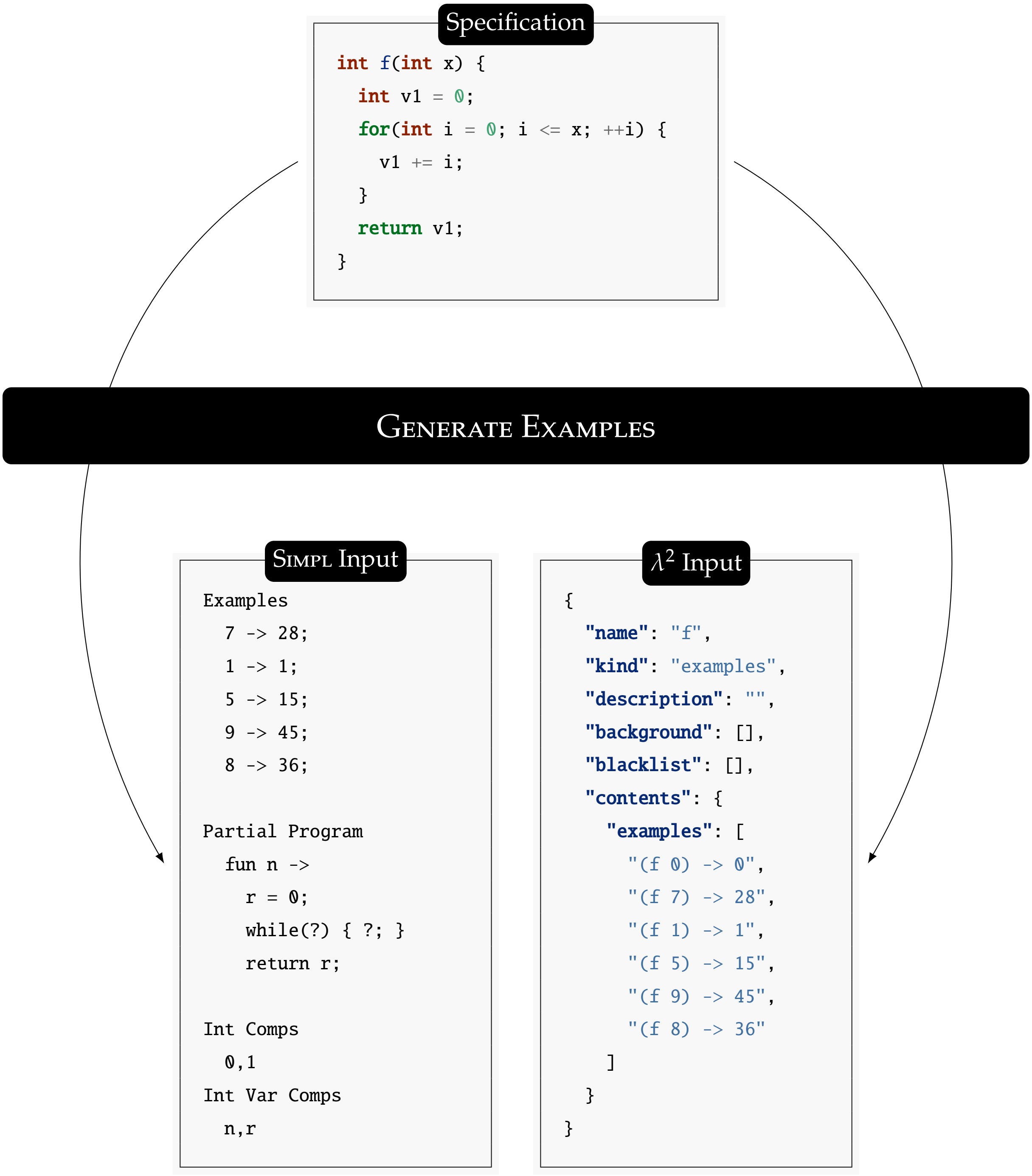
SketchAdapt Serialised, binary, tool-specific Python data structures representing the problem's type signature, inputs and outputs. These structures are generated using code adapted from the original SketchAdapt implementation.

Makespeare Textual format with register and memory states for the x86-like abstract machine targeted by Makespeare. Both the initial and desired final states are encoded in the same way.

Simpl Custom textual format containing a partial program sketch (parsed by Simpl to begin synthesis), a problem type signature, a set of input-output examples, and a set of integer constants to consider in synthesis.

λ^2 JSON files containing lists of input-output examples along with problem metadata. Because λ^2 targets a functional language, the problem type signature is inferred from the examples.

For example's sake, Figure 5.2 shows a problem specification as prepared for both Simpl and λ^2 (SketchAdapt does not use a textual format, and Makespeare's specifications are too verbose to include directly). A script was written for each synthesiser to convert a shared set input-output examples into a problem specification, such that each synthesiser is given as close to the same set of inputs as possible.



```
int f(int x) {
  int v1 = 0;
  for(int i = 0; i <= x; ++i) {
    v1 += i;
  }
  return v1;
}
```

```
Examples
  7 -> 28;
  1 -> 1;
  5 -> 15;
  9 -> 45;
  8 -> 36;

Partial Program
  fun n ->
    r = 0;
    while(?) { ?; }
    return r;

Int Comps
  0,1
Int Var Comps
  n,r
```

```
{
  "name": "f",
  "kind": "examples",
  "description": "",
  "background": [],
  "blacklist": [],
  "contents": {
    "examples": [
      "(f 0) -> 0",
      "(f 7) -> 28",
      "(f 1) -> 1",
      "(f 5) -> 15",
      "(f 9) -> 45",
      "(f 8) -> 36"
    ]
  }
}
```

Figure 5.2: Example synthesiser inputs for Simpl (right) and λ^2 (left). These inouts are generated automatically from a common reference specification (see Section 5.6.3.3). Note that here, both inputs have implementation-specific *help* added: Simpl receives a partial program sketch, while λ^2 has the `"(f 0) -> 0"` recursive base case explicitly instantiated. Inputs for Presyn and Annote are not shown as they have no explicit textual representation, inputs for Makespeare are omitted for brevity, and those for SketchAdapt because they are in binary rather than text format.

5.7.2 Varying Synthesiser Help

As discussed in Section 5.5.3, each synthesiser can make use of a different level of *per-problem* help (i.e. specific direction for that problem, rather than dataset-level training). To evaluate the effect of this help on synthesis performance, two parallel sets of experiments were performed.

First, each synthesiser was evaluated using help equivalent to its original presentation (that is, conditions as similar as possible to its original evaluation). Then, each one is evaluated using the same per-problem help as Presyn, while retaining their training phases if appropriate: only the type signature for each problem is given as help.

Section 5.5.3 describes the information given to each synthesiser; the specific instantiation for each problem aims to match the original work as closely as possible.

Each implementation makes varying use of input-output examples: Simpl and λ^2 use static analysis techniques that become prohibitively expensive for large sets of examples, while Makespeare *requires* large sets to converge on a solution. Synthesisers were each given an appropriate subset of a "master" set of examples (such that the first few examples, given to all implementations, are identical). Makespeare, Annote and Presyn are all given 2,200 examples, while SketchAdapt, Simpl and λ^2 are given only 10.

5.7.3 Experiments

For each of the 112 benchmark examples, the following experimental procedure was followed:

- Dynamically load the symbol (from the shared benchmark library) corresponding to the current synthesis problem.
- Generate 2,200 random input examples, and run the reference implementation on each of them to obtain a corresponding set of 2,200 outputs.
- Run each synthesiser's preparation script on the input-output pairs to produce an appropriate problem description.
- Pass the problem descriptions to the appropriate synthesisers in turn to attempt synthesis.
- If a result is produced, then synthesis succeeds. Otherwise, time each synthesiser out at 12 hours and record a failed attempt.

For implementations that can make use of additional per-problem help, this procedure is repeated with and without the additional help provided.

5.7.4 Model Training

Every benchmark function in the evaluation dataset is annotated with the fragment correspondences that allow Presyn's internal probabilistic models (IID and Markov) to be trained. To prevent the possibility of either over-fitting, or an unfair advantage from considering the entire dataset in advance, these models are trained on a randomly sampled 20% subset of the evaluation data. All synthesis results quoted in the following section are obtained using this training split.

5.8 Results

This section presents the experimental results obtained by following the experimental procedures described in the previous section. First, the number of synthesis problems solved successfully by each implementation is examined, along with the time taken to do so. Then, the impact of the two probabilistic models (IID and Markov) used by Presyn is examined by way of a partial ablation study. Finally, these trained models are inspected to gather insights into the structure of the evaluation dataset.

5.8.1 Coverage

The primary evaluation criteria for Presyn against other synthesiseers is the number of programs in the evaluation dataset it is able to synthesise. Figure 5.3 shows this proportion for two scenarios: firstly, where the synthesisers are given per-problem help as originally specified in their evaluation; secondly, where this help is not given and a 1 hour timeout on synthesis is applied.

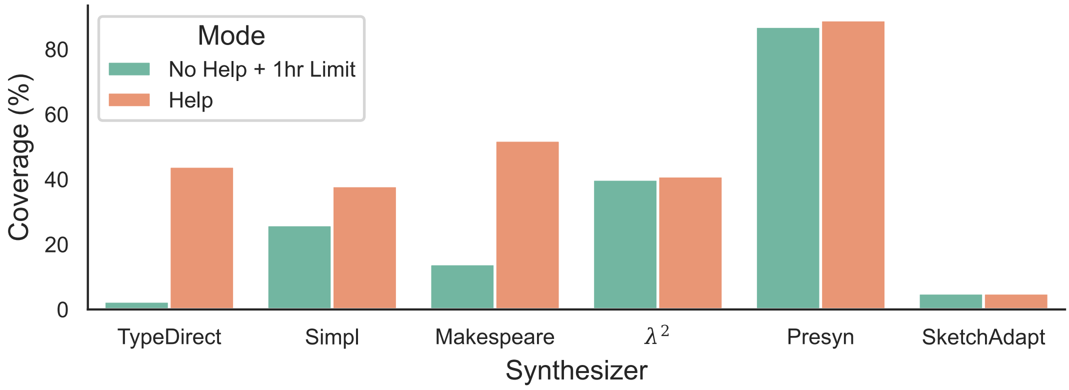


Figure 5.3: Proportion of the synthesis benchmark set synthesised by each implementation under favourable conditions (see section Section 5.5.3), and when restricted by time limits and reduced help.

Table 5.2: Proportion of each group of synthesis benchmarks synthesised by each synthesiser under favourable conditions (see Section 5.5.3)

Group	Annote	Makespeare	Simpl	λ^2	SketchAdapt	Presyn
makespeare	0.20	0.64	0.09	0.45	0.00	0.55
simpl-int	0.00	0.80	0.73	0.20	0.00	0.93
simpl-array	0.58	0.75	0.58	0.58	0.08	0.92
λ^2	0.43	1.00	0.38	0.75	0.00	1.00
SketchAdapt	0.20	0.50	0.00	0.10	0.10	0.50
Mean	**0.26**	**0.73**	**0.39**	**0.39**	**0.04**	**0.79**
string	0.00	0.23	0.13	0.56	0.00	0.75
mathfu	1.00	0.60	0.67	0.47	0.13	1.00
blas	0.75	0.00	0.00	0.00	0.00	1.00
dsp	0.90	0.26	0.40	0.38	0.00	1.00
Mean	**0.92**	**0.33**	**0.38**	**0.48**	**0.04**	**0.93**
Mean	**0.53**	**0.54**	**0.39**	**0.43**	**0.04**	**0.86**

In Table 5.2, the synthesis results for each implementation under favourable conditions are broken down by their original source (synthesis benchmarks and real-world libraries. Presyn is able to successfully synthesise more functions across the set of synthesis benchmarks than each of the other implementations, even when they are given appropriate assistance and unlimited execution time: 89% of the functions evaluated, while the next-best performing (Makespeare) synthesises only 65%. On real-world code, Presyn synthesises 93% vs. 63% for Annote.

Interestingly, it is not the case that each synthesiser performs best on its own benchmarks or that each set of benchmarks is best synthesised with the corresponding implementation; this is likely due to the differences in setup between the experiments in this chapter and the original work. Nonetheless, it indicates that synthesis is by nature a fragile problem to evaluate experimentally.

5.8.1.1 Synthesiser Help

Makespeare required a large number of input-output examples to generalise correctly. Its results were influenced less by the help (passing specific information such as array lengths as input register values) provided, as they were by the time required by synthesis. Relative to the other implementations, Makespeare required a long time to produce successful syntheses (over 12 hours in some cases). For computationally intensive genetic techniques like Makespeare, the time budget available to freely explore the search space is in some sense an additional source of help.

For Simpl, the most valuable source of additional help was the sketch provided to demonstrate the correct program structure. While Simpl is able to explore the search space without this help, not providing it both slows synthesis and reduces the number of successfully synthesised examples.

Broadly, λ^2 is successful without substantial additional help being provided. For a small number of problems, explicitly providing an input-output example encoding a relevant recursive base case produced a correct synthesis where one had not previously been achievable.

SketchAdapt suffered from poor performance on general-purpose problems; it was only successful on trivial examples outside its own evaluation domain. In fact, the experimental procedure found that it was unable to reproduce its own results once the specific input values were changed. As it has the least mature implementation, this is perhaps not unexpected. Additionally, SketchAdapt is in some sense further out of its comfort zone than any of the other synthesisers evaluated here. It is intended to perform well on restricted problem domains where the generalisations required are much smaller, and the dataset covers a more significant fraction of the program space. Examples of this can be seen within the original paper, where Nye et al. [84] are able to demonstrate high accuracy on a number of different subdomains.

When synthesis time is limited or less help is provided for a synthesis problem, Presyn exhibits an even greater advantage over other implementations. Both λ^2 and Simpl exhibit degraded synthesis when assistance is not given (not shown in Figure 5.3 is that successful syntheses took up to 300× longer to discover in these cases). Makespeare's use of a genetic algorithm means that it relies on being able to spend a long time searching a space of programs, and struggles when a timeout is imposed.

5.8.2 Synthesis Time and Validity

The amount of time spent in synthesis by each scheme varied considerably. Presyn, λ^2 and Simpl all showed mean synthesis times (for successful cases) of less than 120 s. SketchAdapt required longer, with a mean synthesis time of 914 s. Because of its reliance on genetic search, Makespeare used a mean of 4522 s per synthesised program, with some taking up to 3× this long before being timed out.

The size of programs generated by Presyn varied from 40 to 110 lines of LLVM IR. Without a formal specification, synthesised programs can only be *tested* for observational equivalence (i.e. they cannot be formally verified as correct). For every Presyn synthesised program, further random and boundary value inputs (outside the initial

Table 5.3: Jaccard coefficient for predictions of the initial fragment set $\mathbf{F_0}$ across each set of synthesis problems.

Group	Acc($\mathbf{F_0}$)
makespeare	0.77
simpl-int	0.85
simpl-array	0.87
λ^2	0.72
SketchAdapt	0.72
string	0.72
mathfu	0.90
blas	0.95
dsp	0.77
Mean	**0.81**

set used for synthesis) were generated to determine if outputs matched those from the target black-box function. In all cases the synthesised program was found to be behaviourally equivalent.

5.8.3 Impact of Probabilistic Models

The two probabilistic models used by Presyn (IID and Markov) both serve to *accelerate* the synthesis process by reducing the size of the relevant search space; if fewer candidate programs are considered, then a correct one can be identified more quickly. This section evaluates how effective these models are at doing so.

First, the accuracy of the predictions made by IID is evaluated. While in theory Presyn could proceed with synthesis without first predicting $\mathbf{F_0}$, the size of the resulting sketch space is so large as to be completely impractical. Because of this, IID's accuracy is measured directly against ground truth data for each synthesis problem to quantify the extent to which it over- or under-approximates.

Then, the impact of Markov is evaluated using a small ablation experiment; the number of candidates evaluated by Presyn before a successful solution is measured both with and without Markov (i.e. applying a uniformly distributed model after IID's predictions).

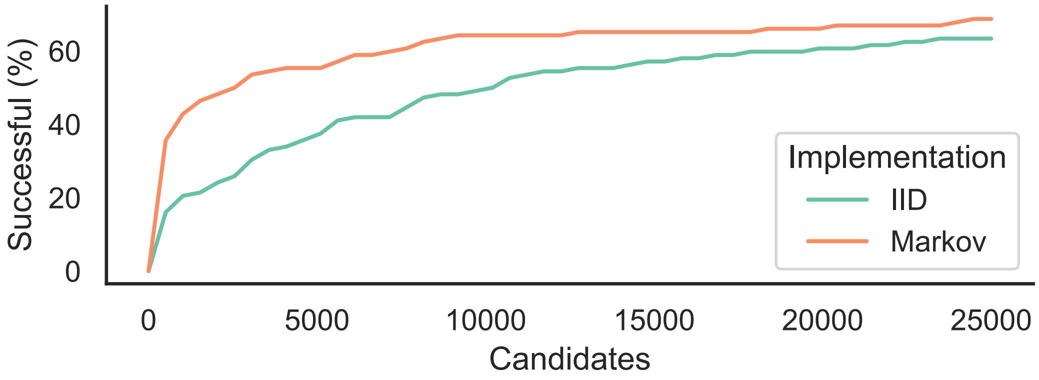


Figure 5.4: Functions synthesised vs. candidates evaluated using the IID and Markov models of synthesis.

5.8.3.1 IID

Because IID can both over- and under-approximate its prediction of $\mathbf{F_0}$ (i.e. it can incorrectly include an irrelevant fragment, or it can omit a relevant one). The most commonly used similarity metric for scenarios such as this is the Jaccard coefficient [152], which measures the *overlap* between two sets. For two sets A and B, the Jaccard coefficient is defined as:

$$J(A,B) \triangleq \frac{|A \cap B|}{|A \cup B|} \tag{5.11}$$

For any two such sets, $0 \leq J(A,B) \leq 1$. To evaluate IID, the ground truth set of fragments $\mathbf{F_t}$ must be known; this is the case for the evaluation dataset, as each example has already been annotated with a set of fragments in order to train IID and Markov. A particular predicted set of fragments $\mathbf{F_0}$ can therefore be assigned an accuracy score:

$$\mathrm{Acc}(\mathbf{F_0}) \triangleq \frac{|\mathbf{F_0} \cap \mathbf{F_t}|}{|\mathbf{F_0} \cup \mathbf{F_t}|} \tag{5.12}$$

Table 5.3 shows the prediction accuracy achieved by IID on each set of benchmark problems. It does not significantly over- or underapproximate; across the entire dataset, an accuracy of 81% represents approximately one prediction error per example.

5.8.3.2 Markov

To evaluate the impact of Markov on PRESYN's synthesis, its performance before and after training Markov was evaluated. For both experiments, the same synthesis procedure was followed, but with a uniformly distributed Markov chain substituted for the trained Markov model in the former case. The distribution of successful syntheses

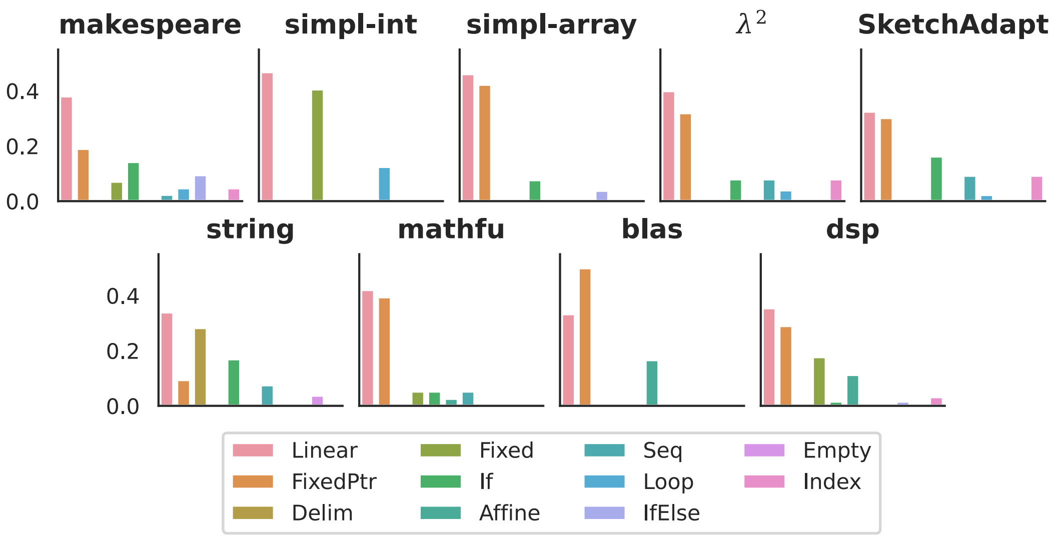


Figure 5.5: Relative frequency of each fragment type in each group of benchmarks.

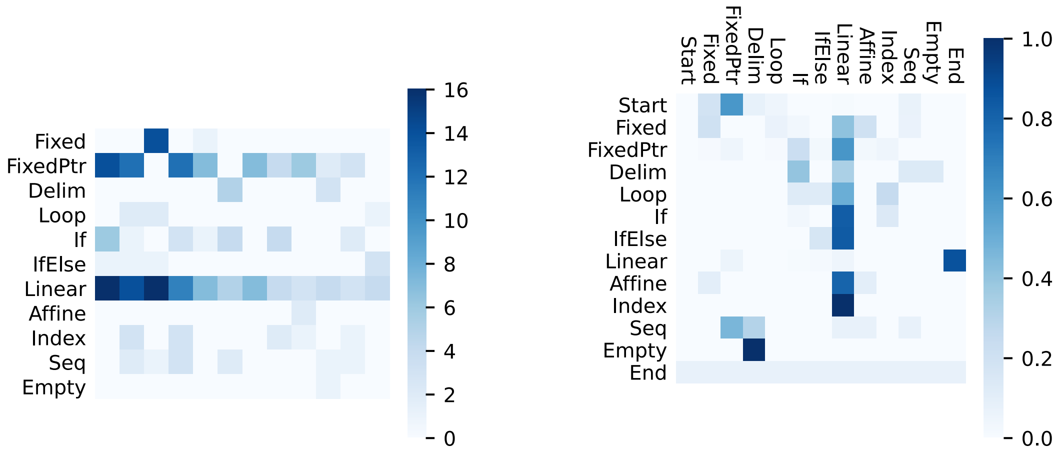
against candidate programs evaluated was recorded in each case, as a measure of how *quickly* successful results can be accumulated.

The results of this experiment are shown in Figure 5.4. It is clear that Markov significantly accelerates the synthesis process; it is able to synthesise 60% of the programs in the dataset using fewer than half as many candidate programs as the uniform model. The distribution of successes is long-tailed, and the final $\sim 20\%$ of the dataset Presyn can synthesise is not shown in this figure. In the long tail of programs, the difference between the two models is smaller as the complexity of synthesis is dominated by the search for long sequences of instructions.

5.8.4 Insights into Program Structure

As well as outperforming competing implementations on a wide range of synthesis problems, Presyn provides interesting statistical insights into the structure of the programs it synthesises through its use of probabilistic models. In this section, these models (IID and Markov) are *updated* after synthesis with the newly observed structure of each solution. The updated models are not used for synthesis; the results shown in previous sections still reflect the original 20% test-train split of the dataset.

In Figure 5.6, three different insights into the structure of synthesised programs are shown. First, in Figure 5.5, the relative frequency of each type of fragment across benchmark groups is given. Linear blocks of code are common across all the benchmarks, as every program is required some kind of computation. In terms of control flow, the easiest synthesis benchmark suites (simpl-int, simpl-array, mathfu and λ^2) are those with largely homogeneous control flow across their benchmarks, while the more



(a) IID distribution of each fragment type, organised by unique type signatures. From left to right, each column represents a type signature. For example, the leftmost column shows the distribution of fragments across functions with the signature `int (int, int*)`.

(b) Markov model transition probabilities for each pair of fragment types; each column's entry is the relative likelihood of that fragment following the fragment identified by the row's label.

Figure 5.6: Insights into the underlying distributions yielded by Presyn's training process.

challenging ones (makespeare, string, SketchAdapt) have far more variation. These results suggest that at least an approximate notion of difficulty for a set of synthesis benchmarks is the heterogeneity in code structure required to solve the problems in that set. Other intuitive structure that can be observed is the ubiquity of *nested* loops in the BLAS matrix-vector problems.

Figure 5.6a shows the distribution of fragments that appear in solutions with each of the 12 most common type signatures in the dataset (from left to right, each column represents a type signature; e.g. `int (int, int*)` in the leftmost column). From this, it is apparent that the most common type signatures dominate the synthesis dataset, with a long tail of less common ones. Part of the reason for this is that signatures are not equal over reordering; if parameters are given in a different order (e.g. `int, float*` rather than `float*, int`), then it is probable that different semantics are intended. Additionally, it is clear that the use of fragments by each signature is consistent (even more so for the most common signatures). This consistency explains the initial success of IID and Markov in synthesising the first, simplest part of the dataset seen in Figure 5.4.

Finally, Figure 5.6b shows the transition probabilities extracted from the fully-trained version of the Markov model. The consistently darker column in the center indicates that compositions are most likely to follow most fragments with a linear region of code. Under the composition semantics given in Section 3.5, this typically represents a control-flow structure being instantiated with a nested computation. This means that Markov is self-limiting, and favours generating smaller programs: most fragments are most likely to be followed by a linear fragment, which is itself likely to end the synthesis.

5.9 Related Work

This chapter deals with the *acceleration* of search-based program synthesis through the application of learned probabilistic models. Similar techniques have been used to improve search through syntactic models [67]; where that work uses a generic formulation of synthesis problems, Presyn develops the domain-specific ideas from Annote to reduce the quantity of training data required. By working within a particular synthesis domain (functions similar to those appearing in common libraries), simpler models can perform favourably against more complex ones. Recent work in neural program synthesis focuses on training larger models with multiple modalities in order to capture this domain specialisation [87, 153, 154].

As well as synthesis, this chapter deals with the systematic cross-evaluation of synthesisers; a well-known problem in the program synthesis literature [46]. While existing work has attempted to assemble aggregate synthesis benchmark suites [47], the evaluation in this chapter considers more varied sources (library functions as well as existing benchmarks), and presents a more detailed analysis of how the *results* produced by different synthesisers can be compared to one another.

5.10 Conclusion

This chapter has addressed the novel problem of *black box* program synthesis, where problem specifications are based on the observed behaviour of an existing component (rather than a human description). The synthesiser developed to attack this problem, Presyn, achieves better performance across a wide range of synthesis benchmarks (composed of both new and existing problems) than five other competing synthesisers, including Annote as introduced in Chapter 4.

Evaluating Presyn required the collection of a large, state-of-the-art set of synthesis benchmarks that partially subsumes those used to evaluate the competing synthesisers. Additionally, a carefully considered comparison methodology was developed to allow for a fair evaluation across different target languages and styles of synthesis. As well as strong synthesis performance, the simple probabilistic models used to implement Presyn provide interesting insights into the structure and difficulty of synthesis problems.

Chapter 6

From Synthesis to API Migration[1]

Chapter 4 presents the design of a program synthesiser (Annote); it aims to synthesise programs that are behaviourally equivalent to functions from an existing software library. By doing so, regions of application code that exhibit the same behaviour can be searched for and replaced with a better implementation. The evaluation of Annote in Chapter 4, and the results it achieves are limited by its requirement for manual annotation of each new synthesis task.

While Chapter 5 improves the *synthesis* aspect of the workflow in Chapter 4, this chapter addresses an orthogonal question: how successful is the *code refactoring* component? To do so, the approach originally prototyped in Chapter 4 is re-examined in the context of *API migration*, a well-known problem in software engineering research.

Beginning with the work done in Chapter 5, an expanded dataset of synthesis tasks derived from software libraries is collected. This dataset builds on the one used to evaluate Presyn by deliberately introducing potential redundancies between libraries (and therefore opportunities for API migration). Presyn is able to successfully generalise to the unseen functions in this expanded dataset using the same training and synthesis procedures defined in Chapter 5.

A novel migration workflow (M^3) that uses inlining of synthesised programs to identify *partial* matches of library behaviour and application code is defined. By doing so, a large number of potential API migrations in real-world applications can be correctly identified and reported.

6.1 API Migration

Libraries are a fundamental feature of software development. They allow the sharing of common code, separation of concerns and a reduction in overall development

[1] This chapter is based on published research in Collie et al. [4].

time. However, libraries are not static: they continually evolve to provide increased functionality, security and performance. Unfortunately, upgrading software to match library evolution is a significant engineering challenge for large code bases.

Given the wide-scale nature of the problem, there is much prior work in the area under various headings (e.g. library upgrade, API evolution or library migration). Work in these areas aims to answer the same question: when (and how) can a program using API X be transformed to one that uses API Y, while preserving its behaviour? This is a difficult problem even when X and Y have compatible or similar interfaces. It becomes more challenging if their behaviours do not match, and requires a complex understanding of the contextual code at library call sites.

There are several approaches to this migration problem: if examples exist of previous successful migrations, then these examples can be used to derive mapping rules [106]. Doing so requires that a full history of the application's source code is available, annotated with the libraries in use at each commit. Neural models have been used successfully to predict properties of programs based on learned vector-space embeddings [105]. However, these approaches require large training sets and are imprecise with respect to program semantics. A more precise (but less automatic) approach is to use expert knowledge to encode known migration patterns [155, 156].

All these prior approaches require some knowledge of the API. This chapter tackles the challenging task of API migration *without* any prior knowledge of the source or target libraries. Here, no access to the library's source code is available, nor to a corpus of example usages of the library. While this scenario may seem draconian, it is often the case in practice [157].

There are several reasons why access to a library's source code may not be available in practice. For licensing or intellectual property reasons, libraries may be deliberately closed-source [158] (i.e. to protect proprietary implementation details). Even setting aside licensing issues, it is often the case that libraries are distributed through package management software as binaries for convenience [159]. It is even possible that there is *no* underlying source code for a library if it represents an interface to specialised hardware or a network service [160]. This chapter proposes a novel approach which automatically learns pattern-based semantic migrations, but without up-front expert knowledge.

6.1.1 M^3: Model, Match and Migrate

The key to the approach in this chapter shares a common heritage with the ideas first elaborated in Chapter 4: deriving a model for library functions that is itself executable code, then using that model to drive refactorings of user code.

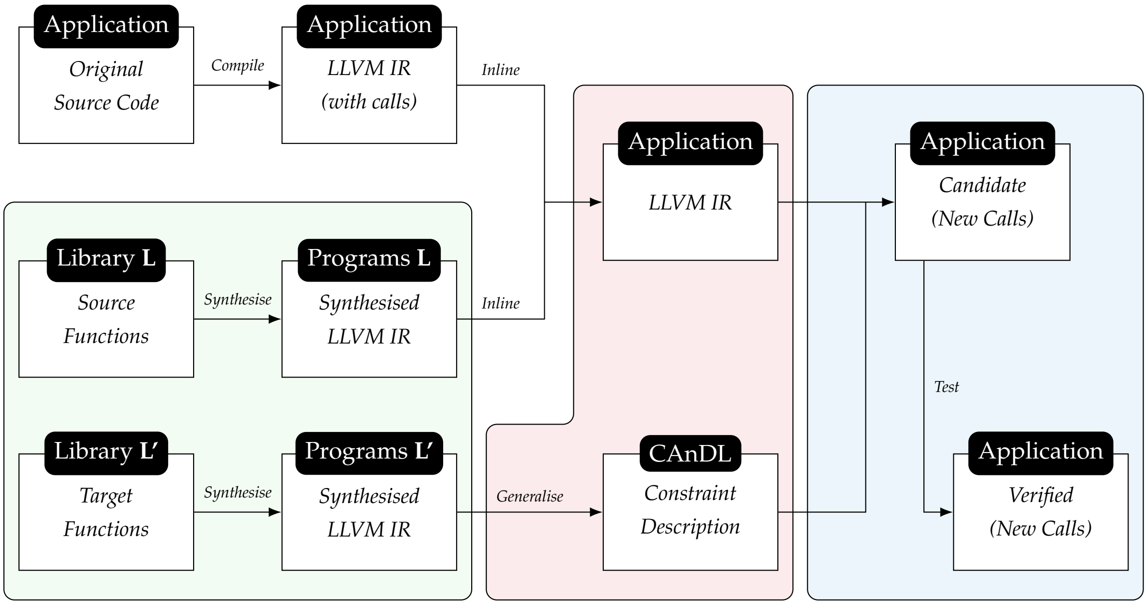


Figure 6.1: A summary of the M^3 workflow. Models for library functions are synthesised. Source functions are inlined while synthesised target functions are generalised into constraint descriptions, which are then used to search compiled user code for potential migrations. From left to right, the three coloured regions highlight the different phases of M^3: Model in green , Match in red , and Migrate in blue .

Deriving migrations from the behaviour of code, rather than from external information (e.g. commit logs or language models) is called *semantics-based* migration in this chapter. Given a specification for a library function (type signature, function name, library binary containing its implementation), M^3 attempts to automatically *Model* its behaviour using program synthesis and checks correctness with respect to automatically generated input-output examples. It inlines the learned program models, then uses compiler-based constraint analysis to *Match* regions of application code with compatible libraries. Finally, these regions can be *Migrated* by replacing application code with library calls.

A useful feature of this approach is that as well as library migration, it allows the refactoring of library-free user code to use libraries. This is because the synthesised models are themselves code, and are inlined and analysed together with application code. Complex refactorings that integrate contextual code around an API call are enabled by this approach.

This approach, while having the benefit of not requiring library vendors to release their source code, relies on the ability to synthesise programs in a reasonable time. To do so, it builds on the probabilistic, sketch-based program synthesiser implemented in Chapter 5 (Presyn). The full approach is evaluated across 7 libraries, synthesising

94 functions, and matching them to over 7,000 old library calls across 10 applications with up to 1MLoC. More than 2,000 of these calls could be successfully migrated to another library.

6.2 Overview

This section describes, at a high level, the M^3 workflow. M^3 is a synthesis-based approach to API migration that allows for *semantics-based* migrations to be discovered, taking into account the contextual behaviour of application code as well as the functionality of the libraries in question.

6.2.1 M^3 Workflow

Figure 6.1 shows an overview of the data flow through the M^3 workflow. Each of the three distinct phases (Model, Match and Migrate) is highlighted, as is the contribution of a traditional compiler.

Before the M^3 workflow begins, an unmodified compiler is used to compile application source code to intermediate representation (as implemented here, LLVM IR [8]). Doing so allows the subsequent phases of M^3 to programmatically manipulate the compiled code before it is finally compiled. In Figure 6.1, regions that are not highlighted are implemented using the unmodified compiler.

The first step (**MODEL**, green highlight in Figure 6.1) is to synthesise executable models for library functions. While the implementation described in this chapter uses PRESYN, the probabilistic sketch-based synthesiser described in Chapter 5, any similar approach could be used in practice. Regardless of the methodology used to obtain them, the key output of the Model step is to obtain executable descriptions of functions from two distinct libraries. In this section, these libraries are referred to as the *source* and *target* libraries; the API migration task addressed by M^3 is to replace calls to source library functions with calls to equivalent target library functions. For brevity, the source and target libraries may equivalently be labelled as **L** and **L'**.

The second phase, (**MATCH**, red highlight) uses the synthesised descriptions of source and target library functions in different ways. First, any calls to synthesised *source* library functions are inlined into the application source code at each call site. Doing so produces code that is library-free as far as possible. That is, it appears *as if* the entire application, including library functionality, had been implemented by hand. The behaviour of the library function and the *context* in which it appears are unified; migrations that require splitting, merging or moving functionality can be discovered and performed.

Next, the synthesised code for target library functions is generalised to a constraint-based description language (the process by which this is achieved is due to Philip Ginsbach's contributions to Collie et al. [2]; see Section 3.3 for an overview and full attribution). This description language permits the use of a constraint solver to efficiently search for regions of application code that match the structure of the synthesised library function.

Finally, after running the CAnDL search process (Section 3.3), a set of matches are identified. These matches show where application source code exhibits structure compatible with synthesised *target* library functions. The final stage of the M^3 workflow (**MIGRATE**, blue highlight) is to identify which of these matches can in fact be migrated correctly to the target library.

To do so, basic integration testing using random examples is carried out on the new (post-migration) code. This helps to eliminate false positive matches. At this stage, the migration can be performed automatically, although in practice the user would be asked to confirm the migration (as is usual with API migration tools). Full-application integration testing can also be carried out to verify the correctness of the implemented migrations.

6.2.2 Example

To demonstrate the types of migration that M^3 offers, it is helpful to consider a running example derived from real application code. Specifically, this example is a set of representative fragments taken from the Common Weakness Enumeration (CWE) database [161]. This database contains examples of programming patterns and API misuses likely to result in safety or security issues in application code.

The example considered in this section relates to the handling of null-terminated (C) strings in application code. For example, if the standard `strncpy` function is used to copy a string, it is not guaranteed that the destination buffer will contain a null terminator. This can lead to buffer over-reads, and so alternative functions are often provided to perform a safer, terminated copy (for example, `strlcpy` on BSD, `StringCchCopy` on Windows or other application-specific implementations). In Figure 6.2, implementations of `strncpy` and `strlcpy` are shown together for comparison; the explicit addition of a null terminator can be seen at line 8.

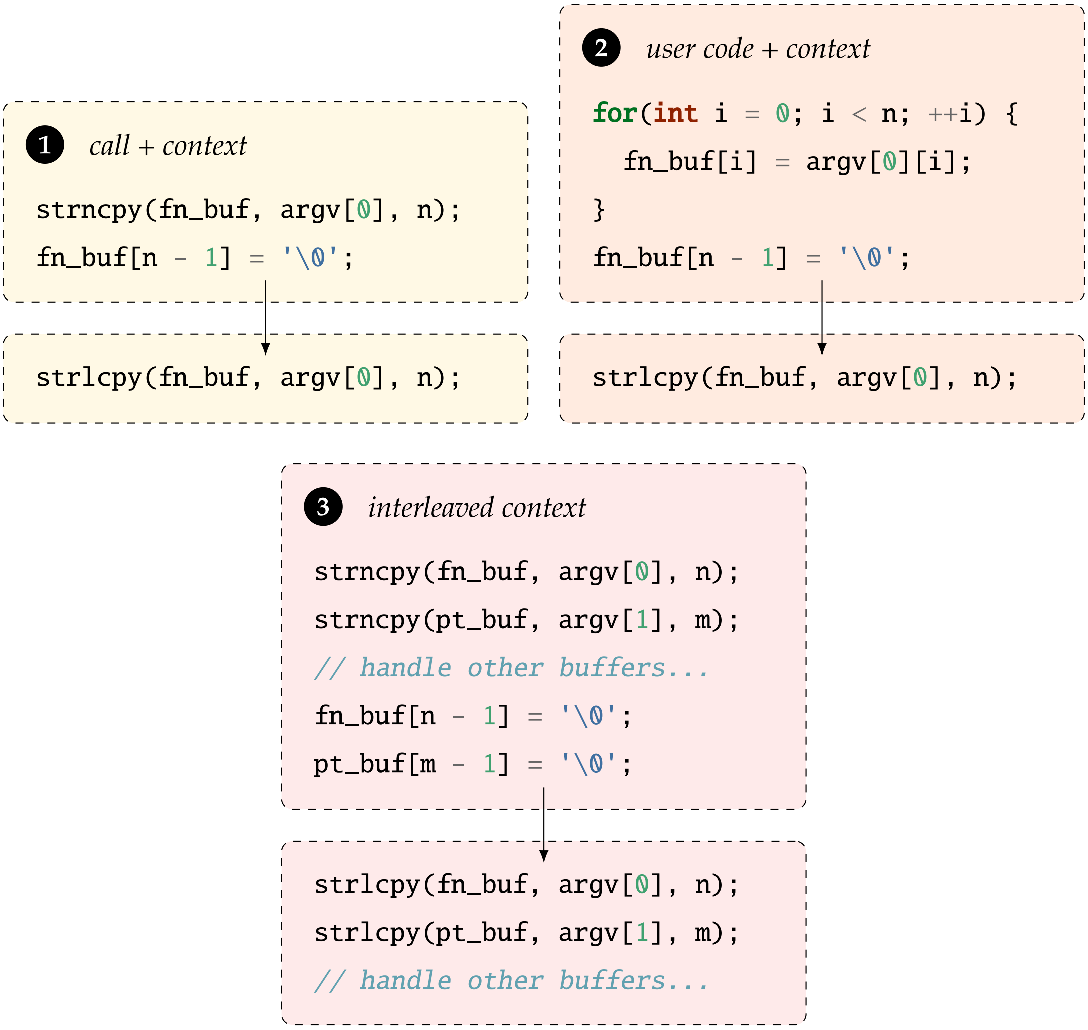
CWE-126 identifies a common pattern where `strncpy` is used in combination with a manually added null terminator, and suggests that such instances should be replaced with calls to the "safer" string copying routines. This is an instance of a useful, real-world API migration task.

Figure 6.3 summarises the three code patterns identified in CWE-126 that are

```c
char *
strncpy(char * dst, const char * src, size_t maxlen) {
    const size_t srclen = strnlen(src, maxlen);
    if (srclen < maxlen) {
        memcpy(dst, src, srclen);
        memset(dst+srclen, 0, maxlen - srclen);
    } else {
        memcpy(dst, src, maxlen);
    }
    return dst;
}
```

```c
size_t
strlcpy(char * dst, const char * src, size_t maxlen) {
    const size_t srclen = strlen(src);
    if (srclen + 1 < maxlen) {
        memcpy(dst, src, srclen + 1);
    } else if (maxlen != 0) {
        memcpy(dst, src, maxlen - 1);
        dst[maxlen-1] = '\0';
    }
    return srclen;
}
```

Figure 6.2: Standard library implementations of `strncpy` and `strncpy`; both are taken from the source code of the Darwin XNU operating system kernel under the terms of the Apple Public Source License Version 2.0.



❶ *call + context*

```
strncpy(fn_buf, argv[0], n);
fn_buf[n - 1] = '\0';
```

```
strlcpy(fn_buf, argv[0], n);
```

❷ *user code + context*

```
for(int i = 0; i < n; ++i) {
  fn_buf[i] = argv[0][i];
}
fn_buf[n - 1] = '\0';
```

```
strlcpy(fn_buf, argv[0], n);
```

❸ *interleaved context*

```
strncpy(fn_buf, argv[0], n);
strncpy(pt_buf, argv[1], m);
// handle other buffers...
fn_buf[n - 1] = '\0';
pt_buf[m - 1] = '\0';
```

```
strlcpy(fn_buf, argv[0], n);
strlcpy(pt_buf, argv[1], m);
// handle other buffers...
```

Figure 6.3: Example of three contexts in which M^3 is able to perform contextual API migrations using only the behaviour of the source and target functions.

suggested as targets for refactoring. The first case ❶ is the simplest. A call to `strncpy` is made, and the destination buffer immediately has a null-terminator set. The relevant CWE entry suggests that this code should be refactored to a single call to `strlcpy`. This migration could be performed using tools such as Refaster [155], but would require an expert to encode it manually.

The second case ❷ highlights the utility of M^3: after performing inlining, the code that explicitly calls `strncpy` is no different to code that performs an explicit loop. Both of these patterns exist in real code, and can be migrated equivalently using M^3. Because many different syntaxes can represent the same underlying semantics, writing source-code based tools that discover loops in this way is a hard problem [12]; M^3's compiler integration and IR-level search allows it to handle loops and other control flow statements homogeneously.

Table 6.1: Corpora used to evaluate M^3.

(a) Application source code for which migrations were tested.

Software	**Description**	**LoC**
`ffmpeg`	Media processing	1,061,655
`texinfo`	Typesetting	76,755
`xrdp`	Remote access protocol	75,921
`coreutils`	Utilities	66,355
`gems`	Graphics helpers	46,619
`darknet`	Deep learning	21,299
`caffepresso`	Deep learning	14,602
`nanvix`	Operating system	11,226
`etr`	Game	2,399
`androidfs`	Filesystem	1,840

(b) Library APIs for which synthesised implementations were learned and used to drive migration.

Library	**Description**
`string.h`	C standard library string handling
`StrSafe.h`	Safety-focused C string handling
`glm`	Graphics functions
`mathfu`	Mathematical functions
`BLAS`	Linear algebra
`Ti DSP`	DSP Kernels
`ARM DSP`	DSP Kernels

Finally, the third case ❸ shows a complex migration where calls to `strncpy` are interleaved with their respective terminations. By operating at the IR level, M^3 is able to identify that no dependencies exist between the calls, and so the migration is possible. In general, source code-based tools, even with expert knowledge, are less able to make this determination.

Unifying these different forms of migration without requiring up-front expert knowledge or library source code is the key advantage of M^3's approach to semantic migrations.

6.3 Implementation

This section summarises the implementation of the M^3 workflow, including the inputs used by each of its three phases (Model, Match and Migrate). Inputs and outputs from different stages of the process are distinguished; M^3 is run for each new *library* of interest, as well as for each *application* targeted for API migrations. Because much of M^3 comprises a composition of other tools, places where new components are contributed are highlighted.

6.3.1 Model

To synthesise executable models of library functions, Model uses Presyn as implemented previously in Chapter 5. Because the dataset of library functions used to train Presyn in Section 5.6 already contains library functions relevant to the migration task attempted by M^3, the same training data was used to implement Model (i.e. a separate training dataset specifically for Model was not collected).

The Model phase is only run during the per-library phase of M^3. Synthesis using Presyn, while practical, is expensive relative to compilation and searching for matches to CAnDL constraints; it is therefore necessary to cache the outputs of synthesis and constraint generalisation.

Model consumes the following inputs for each fresh library to which M^3 is applied:

- An instance of Presyn, pre-trained on an appropriate, relevant dataset following the specification in Section 5.6.
- A type signature for each function in the library that should be modelled (i.e. those for which potential migrations would be useful).
- C foreign function interface (FFI) compatible callable interfaces for each library function with a supplied type signature.

From these, it produces several ($n \geq 2$, as multiple programs are required to generalise constraint descriptions as described in Section 3.3) synthesised implementations for a subset of the supplied library functions (i.e. those that can be successfully synthesised). Multiple reported instances are required to facilitate later stages of the workflow.

6.3.2 Match

The Match phase is concerned with both the per-library and per-application stages of M^3. At the per-library stage, it receives as input the multiple instances of synthesised library functions generated by Model. From these, it uses a version of the graph-based constraint generalisation described in Section 3.3 to produce CAnDL constraints for the synthesised library function.

Several new heuristic post-processing steps are applied by Match to the application code and the generated constraints to improve their accuracy. These steps were specified through a process of manual experimentation, and do not make changes to the functionality of either the code or the constraints. Rather, they act to normalise the two representations and reduce artifacts from specific compilation steps:

- The application code is compiled initially with a set of LLVM optimiser flags that match (as closely as possible) the shared idioms created by fragment compilation routines. For example, the `mem2reg` pass is run (among other optimisations) because fragments do not generate pessimal memory-spilling code when compiled. This normalisation does not rely on the specific semantics of any one fragment, but does require that they collectively generate "intuitive" LLVM IR.

- For each constraint generated, a corresponding graph is extracted (treating atomic constraints as nodes, and references as edges). Then, the largest connected component of this graph is extracted and returned as the "true" component; doing so forces *locality* in the constraint search. CAnDL allows for matches to cover disconnected (potentially distant) regions of code, but library functions as synthesised are very unlikely to do so.

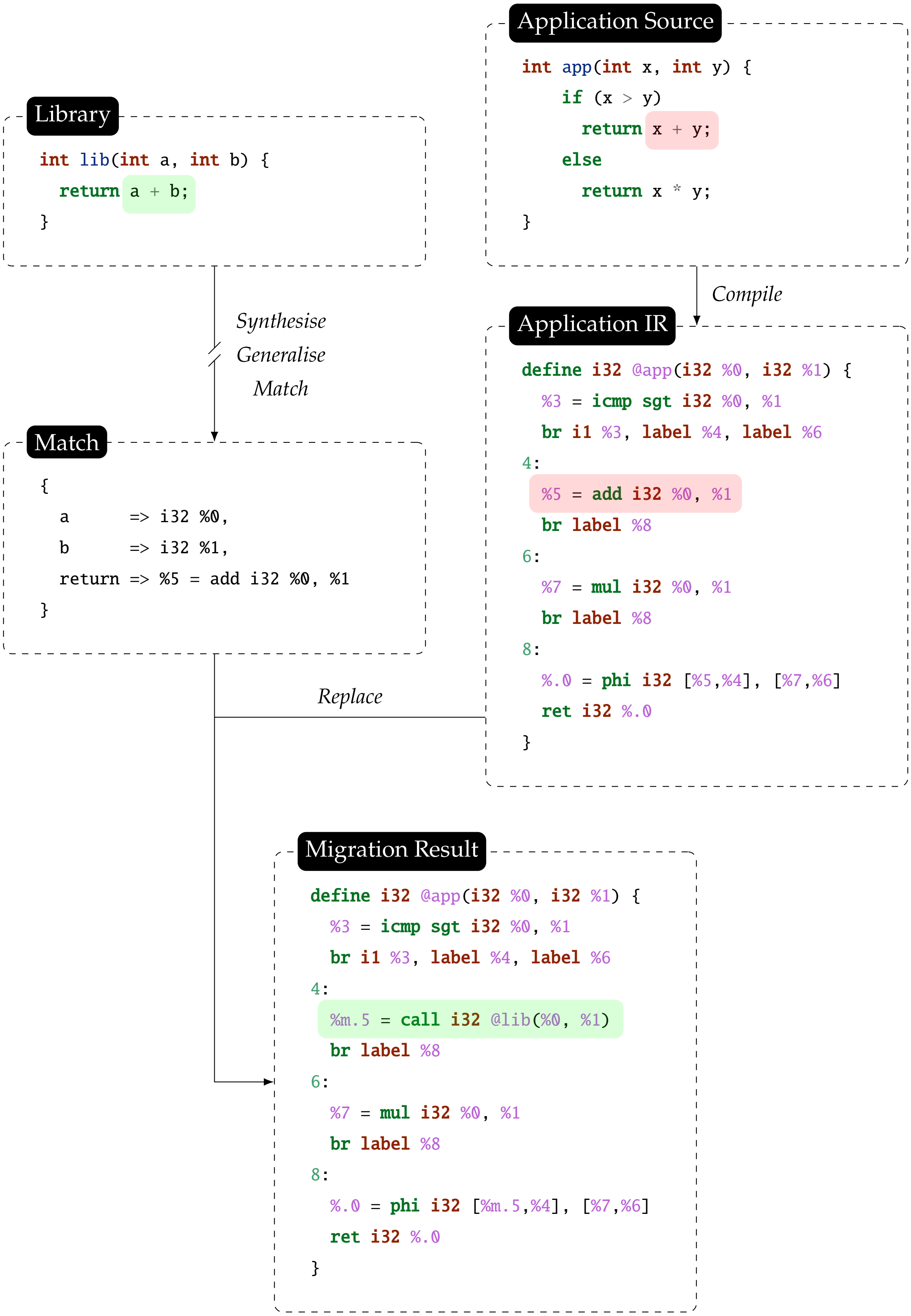
- Some generated patterns are lifted to equivalent, more general ones. For example, a pattern matching `add i32 %a, %b` would be lifted (by commutativity) to one matching `add i32 %b, %a` as well.

After these normalisations are applied, and the CAnDL solver has been run on the application code, Match produces a set of results. Each result encodes the satisfying values for a particular constraint (i.e. a mapping from variable identifiers in the generated constraint to concrete LLVM IR values in the application).

6.3.3 Migrate

The final phase, Migrate, is run only at the per-application stage of applying M^3. It takes as input a match result produced by the previous stage (a constraint variable to IR value mapping), and attempts to *replace* the matched values with a library call. To do so, several checks are necessary:

- The constraints generated must form a connected component on the constraint graph; the matched region is validated to ensure that this is indeed the case on



```
int app(int x, int y) {
    if (x > y)
      return x + y;
    else
      return x * y;
}
```

```
int lib(int a, int b) {
  return a + b;
}
```

```
define i32 @app(i32 %0, i32 %1) {
  %3 = icmp sgt i32 %0, %1
  br i1 %3, label %4, label %6
4:
  %5 = add i32 %0, %1
  br label %8
6:
  %7 = mul i32 %0, %1
  br label %8
8:
  %.0 = phi i32 [%5,%4], [%7,%6]
  ret i32 %.0
}
```

```
{
  a      => i32 %0,
  b      => i32 %1,
  return => %5 = add i32 %0, %1
}
```

```
define i32 @app(i32 %0, i32 %1) {
  %3 = icmp sgt i32 %0, %1
  br i1 %3, label %4, label %6
4:
  %m.5 = call i32 @lib(%0, %1)
  br label %8
6:
  %7 = mul i32 %0, %1
  br label %8
8:
  %.0 = phi i32 [%m.5,%4], [%7,%6]
  ret i32 %.0
}
```

Figure 6.4: Demonstration of how Migrate substitutes calls to library functions for matched IR values in application code.

the application IR.

- There can be no dependencies carried between values that were matched and those that were not; if this is the case, then the application is necessarily performing *additional* computation not modelled by the library function. In some cases, migrating to the new library function would be valid but result in redundant computations being performed.

If all these checks are successful, the values mapped by Match to the library function's *arguments* and *return value* are identified. Then, the migration can be performed. An example of this is shown in Figure 6.4; first, a new call instruction is inserted after the instruction matched to the library return (`%m.5` in the figure). The call takes as arguments the appropriate values from the match result (`a` and `b` in the Match Result box). Then, all references to the matched instruction (i.e. `%5`) are replaced with references to the new call. Finally, the standard LLVM dead code elimination pass is used to remove the previously matched application code.

The migrated application can then be tested. When the enclosing function for a migration is compatible with the behavioural equivalence checking mechanisms used by M^3 during synthesis, an integration test can be applied to the entire function. If this is not the case, the user is responsible for validating the migration; this responsibility is shared with the vast majority of API migration tools and is not a significant burden on the usability of M^3.

6.4 Experimental Design

This section describes an experimental methodology by which the M^3 workflow can be evaluated. First, the ability of the synthesis component used (in this case, Presyn) to generalise to new library functions is evaluated. Then, the correctness of the synthesised programs is examined; observational equivalence is a useful definition in "pure" synthesis contexts, but must be more closely examined when carrying out API migrations. Finally, the respective accuracies of the Match and Migrate phases must be measured: how many instances of target library functions can be discovered in application source code, and how many of these can be successfully migrated.

These evaluation criteria are broken down into four questions as follows:

Feasibility and effectiveness of the Model phase: Can Presyn's probabilistic program synthesis be used effectively to learn the behaviour of black-box library functions?

Correctness of synthesised programs: Do the programs synthesised by Presyn behave the same as the target program over a particular set of inputs (that is, are they observationally equivalent)?

Presyn uses randomly generated inputs to verify this correctness in practice. To assess the adequacy of these random inputs (in terms of how well they exercise different application behaviours in practice), the branch coverage achieved across the evaluation dataset is measured.

Accuracy of Match phase: Given synthesised implementations for library functions, can compatible instances in application code be accurately discovered? In this research question, the ability of the Match phase to accurately discover inlined implementations of the *same* synthesised library functions in application code is measured.

Accuracy of Migrate phase: Given instances of user code that match the constraints generated from a library function, can API migrations be correctly implemented? This research question investigates ability and accuracy of the Migrate phase in matching and migrating implementations in application code to *different* library functions.

The remainder of this section first describes the collation of an evaluation dataset for synthesis and API migration in practice, then details the experimental procedure by which each of the above research questions can be answered.

6.4.1 Evaluation Corpora

To fairly evaluate M^3, two separate datasets are required. First, a set of libraries (and therefore library functions) is required. Because these functions will be used to identify potential API migrations, there should be some redundancy across libraries; that is, if two libraries both implement function F, then both copies should be retained in the dataset. This contrasts with the approach taken to evaluate Presyn in Chapter 5, where the aim was to measure the number of orthogonal, *unique* functions synthesisable by each implementation. As well as directly equivalent functions, the libraries selected should exhibit instances of *contextual* API migrations that are not direct replacements.

Once these libraries are selected, the second required dataset is a set of applications against which the potential for API migrations can be evaluated. These applications should make calls to some of the libraries selected previously, and should additionally demonstrate potential contextual migrations.

6.4.1.1 Libraries

To evaluate M^3, 7 distinct libraries were selected as targets for migration. These relate, broadly, to two problem domains (so that the number of potentially demonstrable migrations can be maximised): string processing and mathematical operations. Similar problem domains are commonly targeted for migration by other related work in API migration, though often with different tooling and source language contexts [CITE]. Additionally, integrating with the tools used in M^3's implementation (i.e. Presyn, CAnDL and LLVM IR) meant that the libraries selected were limited to those with APIs compatible with the C foreign function interface (FFI). The libraries selected were:

String Processing The starting point for selecting string processing libraries was the C standard library implementation of `<string.h>`. As identified in Section 6.2.2, the standard functions available for manipulating C strings are easily misused, with such usage errors potentially leading to critical security issues.

However, usage of standard C functions is endemic, even to C programs written newly today. Migrating potentially unsafe usages to widely-available, less readily misused libraries is therefore a valuable API migration task. To evaluate how well M^3 is able to carry this task out, the StrSafe library was identified as a candidate for replacement [162].

There are many implementations that implement essentially the same set of improved string manipulations as StrSafe; repeating the evaluation on these libraries would not provide any *new* information, and so StrSafe was selected as the sole candidate for migration.

Mathematical There exists a far broader range of implementations within the scope of mathematical libraries; many general-purpose libraries partially overlap in terms of their supported problem domain. To evaluate M^3's ability to identify these partial overlaps, 5 such libraries were identified. These were the GLM support library for graphical applications, Mathfu (a general-purpose library), a subset of the BLAS standard interface, and two platform-specific signal processing libraries distributed by ARM and Ti respectively.

Discovering migrations between these libraries is likely to provide useful insight into cross-platform migrations, as well as those related to performance (as in Chapter 4).

A more detailed summary of the 7 libraries selected, and their respective functionality is listed in Table 6.1b. Similarly to the evaluation of Presyn in Section 5.7, a shared library interface and collection of type signatures was prepared as the synthesiser input; doing so requires no knowledge of a library function's interface than its signature and the ability to make calls to it.

6.4.1.2 Applications

To evaluate the Match and Migrate components of M^3, 9 widely-used, real-world applications were identified. These applications target different problem domains and are unrelated to each other. A summary of each of the applications selected is given in Table 6.1a.

These applications were selected by manually searching GitHub and similar online repositories[2] for code that matched the following criteria:

- The most important criterion for selection was a build system that permitted easy interposition of the M^3 compiler toolchain. As implemented in this chapter, this requirement ruled out applications not written in C or C++, although with some additional engineering work any language with an LLVM frontend could be integrated.
- When selecting applications, large codebases in active real-world user were prioritised. Projects under active development, or those for which significant distribution and usage could be identified were preferred.
- Additionally, applications were selected so as to maximise the diversity of implementation and target domain in the dataset. Duplicating domain-specific idioms or paradigms within the dataset is less likely to provide useful insights into the abilities of M^3.
- No pre-selection of applications based on knowledge of their source code was performed; no familiarity with any of the 9 applications was taken advantage of when constructing the dataset.

6.5 Results

This section describes the results M^3 was able to achieve with respect to the 4 research questions posed in Section 6.4. Presyn is able to generalise from the results in Chapter 5 to new, unseen libraries. The programs it synthesises are shown to be observationally equivalent to the reference implementations, and the use of randomly generated input-output examples is validated by testing branch coverage. The Match

[2] Using `https://searchcode.com/`

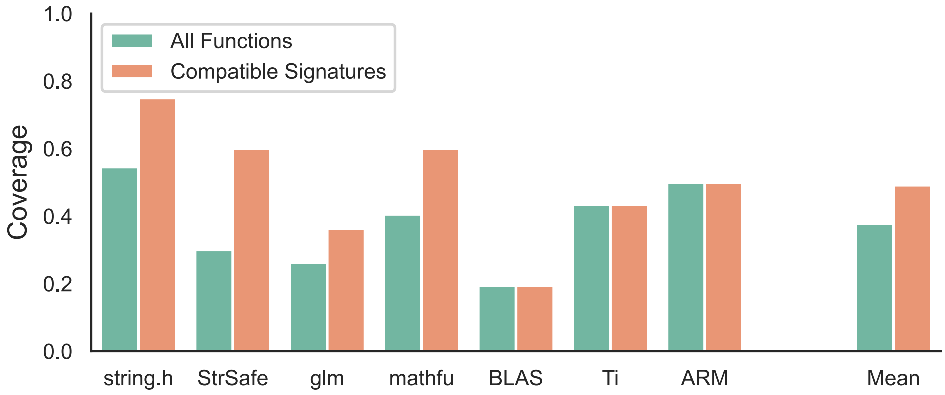


Figure 6.5: Proportion of each library's API that Presyn was able to successfully synthesise, across all functions in the library as well as those with compatible type signatures.

and Migrate phases of M^3 are able to effectively and accurately discover instances of generalised constraints in application code, that can subsequently be migrated to new target libraries.

6.5.1 Feasibility and Effectiveness of Model

The first research question posed in Section 6.4 asks whether program synthesis is a feasible approach to modelling the behaviour of a library. To do so, first Presyn's ability to generalise to the correct synthesis of new, unseen libraries beyond its original evaluation and training set is evaluated. To do so, the proportion of functions in each library API across the evaluation dataset is measured. For synthesis using Presyn to be a useful component of M^3 in practice, the time taken to successfully synthesise each example must be practical.

6.5.1.1 Synthesis Coverage

Figure 6.5 shows the proportion of each library's API that could be successfully synthesised by Presyn. Across the entire evaluation dataset, it is able to synthesise nearly 40% of functions. This figure includes functions that cannot be loaded by Presyn because their signatures are not compatible for engineering reasons (for example, if they require pointers-to-pointers, or opaque data types). Discounting these functions, Presyn is able to successfully synthesise approximately half of the loadable functions. This represents a significant proportion of each library's behaviour; even in the worst performing case (BLAS), it is able to synthesise nearly 20% of all functions in the library. Performance on the BLAS library is limited by the high complexity of many of its constituent functions (e.g. solving systems of equations on packed matrix structures).

For each synthesis failure in the set of evaluation functions, the reference function was examined by hand to determine why it could not be synthesised. In some cases (e.g. `strtok` from `string.h`), the function demonstrated stateful behaviour. Modelling this type of function is an open problem in program synthesis, with recent work addressing limited contexts such as heap manipulation [163]. Presyn's synthesis methodology presumes that target functions are idempotent, and so does not support stateful functions. Doing so is interesting future work. A small number of functions (e.g. `ssyr2k` from `blas`) exhibit unusual control flow idioms not expressible using Presyn's set of fragments. However, the majority of failures are timeouts resulting from long required sequences of instructions in target functions.

The coverage achieved by Presyn on libraries it was not originally trained and evaluated on in Chapter 5 is consistent with the original training set. For example, the StrSafe library does not implement improved versions of the simplest (i.e. least error-prone) functions in the `string.h` library; across the set of similar functions, the coverage achieved by Presyn is almost identical.[3] A similar effect can be observed on the signal processing libraries.

These results indicate that Presyn is able to synthesise a substantial proportion of each library's API. Taking into account the similarity of results between the new and previously-evaluated libraries, as well as Presyn's overall performance in Section 5.8, it can be concluded that these results approach the best performance achievable by fully black-box synthesisers at the time of writing. Achieving further coverage would clearly benefit the overall success and applicability of M^3, but is likely to be a challenging goal on the most complex of library functions under the black-box assumption. Relaxing this assumption (i.e. providing additional information to a synthesiser beyond the black-box model) is likely to provide a benefit to synthesis performance; one such set of relaxations is examined in Chapter 7.

6.5.1.2 Difficulty

For the Model component of M^3 (driven by Presyn) to be useful in practice, it must be able to achieve successful syntheses in a practical time. The usage model of M^3 suggests that this time can be greater than the time taken for compilation; synthesis of each library function is a one-off task that is not repeated when a new application is compiled. Nonetheless, the time taken should still be within the scope of a traditional

[3] This similarity is determined manually, and is intended as a qualitative demonstration of the similar coverage achieved over both library APIs. For example, the functions `strncpy` and `StringCchCopyN` functions are not directly equivalent, but are similar in their synthesis difficulty and intent.

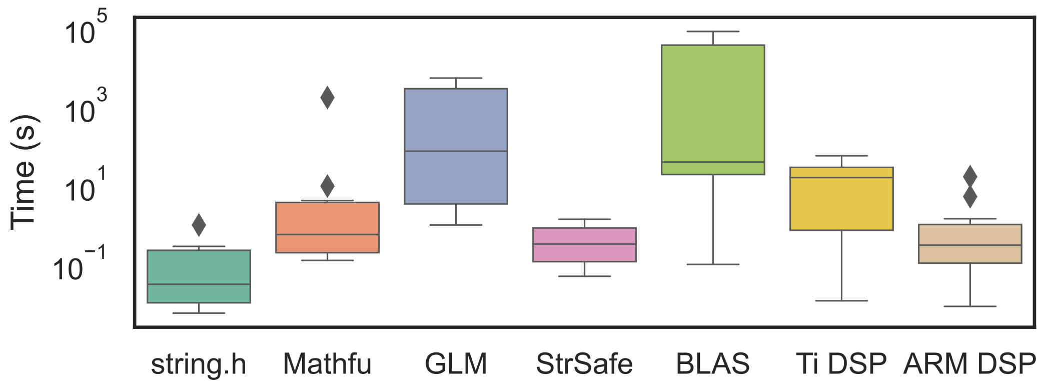


Figure 6.6: Distribution of synthesis times for each library API.

development workflow. To reflect this, a 4 hour timeout was used when synthesising these functions.

Figure 6.6 shows the distribution of synthesis times across each of the libraries evaluated. The distribution is long-tailed; the median function across the entire dataset can be synthesised in less than two minutes, with the most challenging ones taking over three hours. These results indicate that the one-off Model phase of M^3 can be practically integrated with a typical development workflow.

6.5.2 Correctness of Synthesis

Synthesised functions are tested for correctness using *observational equivalence*; because M^3 is designed for black-box library interfaces, there is no better way to determine correctness in practice. However, as the underlying code for each library function is available as an artifact of the evaluation process, the adequacy of this method of testing can be verified by measuring the *branch coverage* achieved on each synthesised function.

6.5.2.1 Observational Equivalence

Presyn tests synthesised solutions for correctness using randomly generated input examples; these examples are collected from the reference implementation and used to obtain true output values. Input examples are generated by uniformly sampling values in the range of the input data types. To provide a stronger guarantee that the reported solutions are in fact correct, two further strategies were employed for each reported solution:

Manual Evaluation As well as testing using random IO examples, each synthesised solution was examined by hand, using the author's knowledge of the intended

```c
int tricky(int x, int y) {
  if(x == 0x0BADC0DE) {
    return y;
  }
  return x + y;
}
```

```c
int wrong(int x, int y) {
  // Incorrect when
  //   x == 0x0BADC0DE!
  return x + y;
}
```

Figure 6.7: The right-hand side shows an example C function that demonstrates divergent behaviour on a sparse subset of its input domain (in this case, a single "magic number"). Detecting such cases is a challenging problem for black-box synthesis methods; shown on the left is the solution likely to be returned from a synthesiser using observational equivalence to decide correctness.

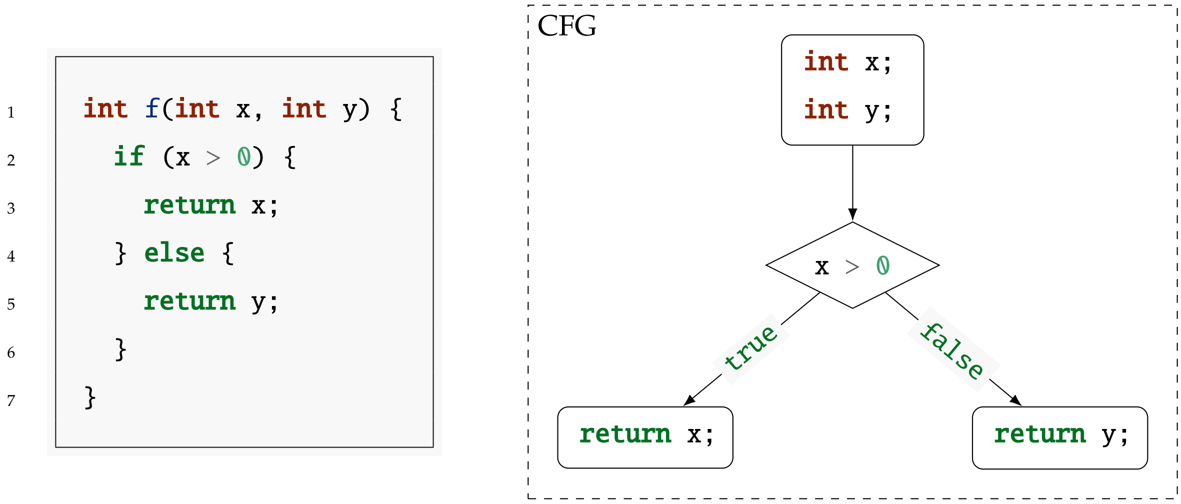
behaviour of that example. Only one program was judged to be incorrect: the `memmove` function from `string.h`. If the memory regions passed as arguments aliased (i.e. they overlapped), the synthesised implementation would exhibit incorrect behaviour. The testing methodology used by Presyn did not generate aliased memory; generating a set of such inputs manually forced Presyn to correctly synthesise `memmove`.

Boundary Value Testing Additionally, each synthesised candidate was tested using boundary and outside range values for inputs. In every case, the synthesised candidate conformed to the expected behaviour on these inputs.

6.5.3 Code Coverage

The primary limitation of observational equivalence over random inputs is when the reference implementation exhibits divergent behaviour on a sparse subset of its possible inputs. For example, the function on the left of Figure 6.7 returns `x + y` for every `x` and `y` except when `x == 0x0BADC0DE`. If the set of generated input examples does not include this special-cased value, then a synthesiser is likely to report an incorrect solution (i.e. the code on the right hand side of Figure 6.7).

Resolving these cases is a somewhat philosophical issue in synthesis. For implementations where input examples are supplied by the user, the implicit assumption is that they will cover the entire space of potential behaviours. However, in the black-box scenario assumed in this chapter, the input examples are *gathered* from the reference implementation by the synthesiser. If there are special cases that are not collected



```
int f(int x, int y) {
  if (x > 0) {
    return x;
  } else {
    return y;
  }
}
```

Figure 6.8: A C function shown with its control flow graph, illustrating the "distinct branches" element of measuring branch coverage. For any single value of `x` passed to the function, a branch coverage of 0.5 will be achieved (as the two branches are mutually exclusive).

during this process, it is a failure of *synthesis* rather than a failure of *specification*.

To understand how effective Presyn is at capturing all possible behaviours of the library functions it targets, the adequacy of the randomly generated inputs it uses is measured. That is, how effective are random inputs in exercising behaviours of the synthesised implementations. To do this, the *branch coverage* achieved on each solution reported as successful is measured.

Branch coverage is a commonly used metric for code and test coverage experiments as it is easy to measure and report, and is likely to be sufficiently precise for functions without complex conditionals.[4] Branch coverage is defined as the proportion of *distinct* conditional branch results taken across a set of program executions. For example, the CFG shown on the left-hand side of Figure 6.8 contains two conditional branch results that could be taken. Each individual execution will result in a branch coverage of exactly 0.5, but if multiple values of `x` are passed such that both branches are taken on some executions, it is possible to achieve coverage of 1.

Figure 6.9 shows the branch coverage achieved across the full set of library functions evaluated, as a function of the number of random input examples used. With as few as 10 distinct inputs, more than 98% of the branch conditions in the corpus of synthesised programs are exercised. Typically, at most around 30 random inputs are needed to provide 100% branch coverage for any synthesised candidate. The numerical libraries evaluated most often contain loops as their primary control flow; branch coverage is

[4] In such cases, splitting of boolean subexpressions, or more stringent metrics such as path coverage could be applied.

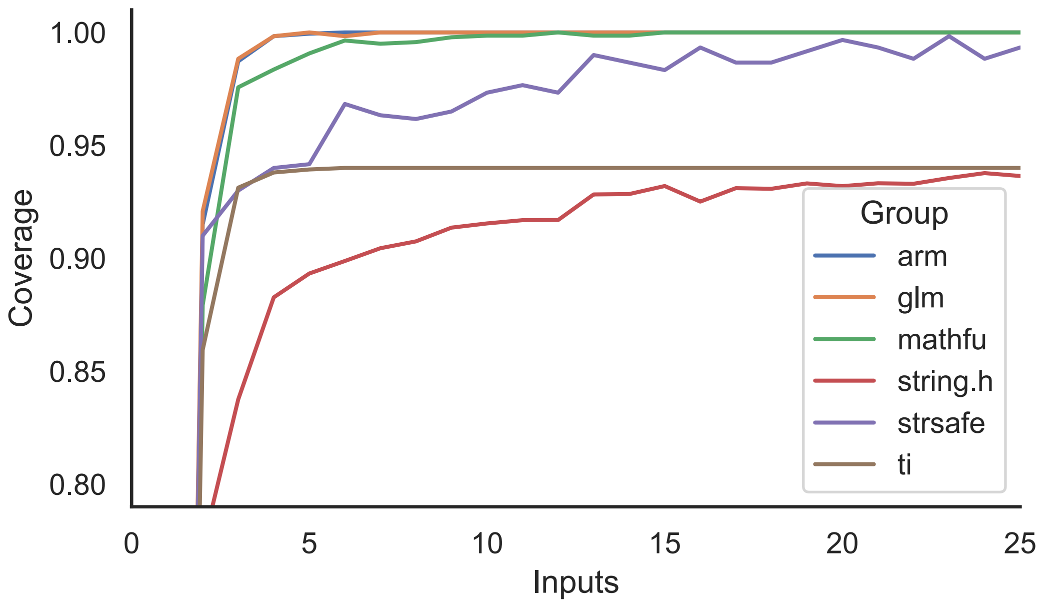


Figure 6.9: Corpus branch coverage achieved using randomly generated inputs. Coverage values are reported as the mean of three separate runs (using different randomly generated inputs on each run). For the `string.h` and `ti` libraries, full coverage was not achieved because of a check for overlapping memory in the `memmove` function (or an analogue).

less difficult to achieve over looping code than over conditionals. These results provide confidence that the synthesised candidates behave equivalently to the target program with respect to inputs that exercise the complete control flow in the candidates.

6.5.3.1 Inside the Black Box

In many cases, the underlying source code for the libraries in the evaluation set was available in some form, making it possible to directly compare synthesised programs to the original code by "looking inside" the black box. These programs were compiled to LLVM IR and used as input to the Match and Migrate phases as if they had in fact been synthesised by Presyn. No meaningful divergence in results could be observed by doing so; similar per-library branch coverage was achieved, and the compiled IR for synthesised and handwritten implementations was almost identical in most cases. No behavioural differences were observed.

Table 6.2: The number of call sites where synthesised functions were inlined in each application, along with the proportion of these that could be successfully rediscovered using Match.

Application	**Inlined Calls (L→L)**		**User Code (C→L′)**
	#Instances	Matched (%)	#Instances
`ffmpeg`	4,976	100%	24
`texinfo`	586	100%	1
`xrdp`	686	100%	0
`coreutils`	623	100%	16
`gems`	46	100%	61
`darknet`	128	100%	13
`caffepresso`	189	100%	0
`nanvix`	0	-	16
`etr`	4	100%	45
`androidfs`	0	-	2
Total	**7,238**		**178**

6.5.4 Accuracy of Match

This section evaluates the success of Match. To do so, its "rediscovery" success rate on synthesised library functions is examined, as well as its ability to discover user code matching these functions. A common notation shared with Migrate is established to describe these results.

6.5.4.1 Rediscovery

For the Match phase of M^3 to be useful in practice, an important litmus test is whether it is able to recover all inlined instances of a library function from the constraints generated from that library function. If this is not the case, then information is lost during the Match phase, and it may not be possible to reverse the inlining process to recover the original application code.

To assess whether Match satisfies this property in practice, every successfully synthesised function was inlined at each of its call sites across the set of evaluation applications. Then, the generated CAnDL constraints for that function were used to search for matches in the application code; Table 6.2 shows the results of doing so (the number of library call sites that were inlined, along with the proportion of these that were rediscovered successfully). As shorthand, instances of inlined library code that

```c
void copy_cpu(int N, float *X, int INCX, float *Y, int INCY)
{
    int i;
    for(i = 0; i < N; ++i) Y[i*INCY] = X[i*INCX];
}
```

Figure 6.10: Re-implementation of the standard BLAS function `SCOPY` discovered in the Darknet [123] machine learning framework.

could be matched against the *same* original function are written **L**→**L** (i.e. **L**ibrary to **L**ibrary matches).

These results show that Match is able to successfully recover every instance of inlined library calls in the original code. This is because the same, identical code is inlined at each site, and because inlining does not change the structure of the code from which the constraint description was generated.

6.5.4.2 Application Code

As well as being able to successfully identify inlined calls, Match is able to identify locations in the application code where equivalent functionality to a library function is implemented. For example, for portability the Darknet machine learning framework re-implements some functions from the BLAS standard, one of which is shown in Figure 6.10. M^3 is able to match the body of this function against the previously synthesised and generated constraints for `SCOPY`. This ability is not limited to "clean" function body matches; Match is able to identify locations within larger contexts of application code where similar matches occur.

This type of match is notated as **C**→**L**′ (i.e. user **C**ode to new **L**ibrary), and the number of such instances discovered in each application is shown in Table 6.2. A manual search for further instances not discovered by Match based on these results was performed. A combination of several techniques was used to perform this search: handwritten CAnDL constraints for significantly abstracted versions of each function were used to guide an initial search, as well as textual similarity and heuristic exploration of the code.

For example, where a re-implementation of one string-processing function was discovered, a search for similar re-implementations that were not discovered by Match was performed. For a region to be classified as a re-implementation, it was required

Table 6.3: Migration opportunities discovered in each application, broken down by the category of the source context (source library calls **L** or user code **C**).

Application	**Migrations**	**Category**		
		L→L′	**C→L′**	**L+C→L′**
`ffmpeg`	655	629	24	2
`texinfo`	431	413	1	17
`xrdp`	274	269	0	5
`coreutils`	649	633	16	0
`gems`	107	46	61	0
`darknet`	40	7	13	20
`caffepresso`	24	24	0	0
`nanvix`	16	0	16	0
`etr`	49	4	45	0
`androidfs`	2	0	2	0
Total	**2,247**	**2,025**	**178**	**44**

that on well-formed inputs (i.e. not accounting for "exceptional" control flow), the region performs the same task as the original function.

No further instances of this kind were identified by this search, confirming with reasonable certainty that there were no false negatives from Match (though no technique can verify this formally). The constraints generated by Match were specific enough that none of the application code matches represented false positives.

Running the CAnDL solver takes additional time during compilation; approximately the same as compilation itself for each pattern to be searched for [12]. This time is not a bottleneck when using M^3 practically.

6.5.5 Accuracy of Migrate

Finally, to assess the usefulness of M^3 as a *migration* tool, the number of matches that can be translated to a different library (rather than the **L→L** matches where uses of the original library are recovered) must be evaluated.

To do so, the number of cases where the generated constraints for each library function matched application code that was *not originally* a call to that function was counted. This quantifies the number of possible migrations enabled by M^3. Table 6.3 gives the total number of migrations found in each application, as well as a breakdown into three categories:

- Replacement of a source function with a semantically identical target function from a different library (**L→L′**).
- Identification and replacement of redundant application code that could be better expressed as a target library function call (**C→L′**).
- Replacement of code that *combines* a source library call and handwritten code with a target function (**L+C→L′**).

The most common migrations were **L→L′**, where two libraries implemented the same function (for example, delimited string copying or a vector dot product). Some functions did not produce migration opportunities, even though they could be inlined and matched. `memcpy` is an example of this due to its ubiquity; applications like `ffmpeg` and `xrdp` that frequently perform buffer copies show far fewer migrations than inlined matches because no migrations were available for `memcpy`.

Note that the category **C→L′** corresponds exactly to the number of matches to user code (**C→L′**) quoted in Table 6.2. This is because any matching instance of a function in application code represents a migration opportunity; if original, handwritten application code matches the generated constraints for a library function, there is by definition an available migration.

These results demonstrate that M^3 is able to successfully identify distinct classes of migration (other tools are often limited to one of these classes only, and **L+C→L′** migrations generally require expert knowledge to express). The migrations identified are useful and would be difficult to identify with existing tools.

6.5.6 Threats to Validity

The results in this section show that M^3 is able to identify and perform a large number of useful migrations using real-world applications and libraries, in contexts not well served by existing tools. The primary threats to *internal validity* are:

1. The fragment vocabulary used by PRESYN and Model is a limiting factor; the variety of programs that can be synthesised depends on this vocabulary. However, this is a limitation shared by all sketching program synthesisers, and can be extended easily without changing the overall M^3 approach.

2. The CAnDL constraint generation used by Match and Migrate is not formally verified; it relies on testing with different library functions to check constraints always match their source programs.

3. The correctness of synthesised implementations against their respective reference implementations is checked using randomly generated test inputs, with

additional reassurance from branch coverage tests. Proving total correctness is challenging even if the original reference program were available; doing so amounts to program equivalence checking, an open research question for imperative programs as considered in this thesis [164]. If the original reference is not available, then observational equivalence represents the best possible correctness specification.

The main threat to *external validity* lies in the subject libraries chosen and the restriction to two problem domains: string processing and mathematical operations. These domains have also been targeted by other migration tools and are therefore used to facilitate comparison. PRESYN and Model are not fundamentally restricted to these domains Extending their vocabulary of fragments to include more expressive computations will allow them to scale synthesis to more complex APIs and functions.

6.6 Related Work

The work in this chapter offers improvements over two common approaches to API migration: automatic library-to-library migrations, and complex semantic rewriting of application code.

Automated approaches to library migration often use large corpora of migration examples, from which statistical inferences about the likely occurrence of further migrations can be drawn. For example, pan [165] use textual similarity metrics to identify API calls that appear in similar application contexts. Recent work has aimed to decrease the degree of prior data required [166], but in the context targeted by M^3, such data may not exist at all.

More complex migrations often require expert users to encode their desired patterns manually. For example, tools like ReFaster [155] and NoBrainer [156] (for Java and C/C++ respectively), generate migration patterns from paired source code examples. These patterns can be *applied* automatically, but require an expert user to encode the pattern initially. Similarly, the initial work on the CAnDL language and migration engine [12] (upon which M^3 is built) can automatically discover migration opportunities given a manually-encoded description of the relevant code or libraries. M^3 uses program synthesis to learn these descriptions, and provides a method for automatically applying migrations. By doing so, it combines automated application with semantic analysis without requiring large quantities of training data. Other tools that aim to perform complex migrations (such as Meditor [167] or EdSynth [168]) either require more information, or migrate at the syntactic level, thereby sacrificing precision.

6.7 Conclusion

This chapter has proposed a novel API migration tool that enables complex migrations in real-world code. This tool, M^3, uses the behaviour of library functions to synthesise executable models, and discover migrations without expert knowledge, change logs, or access to the library's source code.

This approach was successfully applied to 7 large, widely-used libraries, successfully synthesising nearly 40% of their functions. From these, M^3 was able to discover over 7,000 matching instances in 10 well-known C/C++ applications, many of which represented missed opportunities for library usage or optimisation. Using constraint-based search for API migration allows for the *semantics* of the code in question to be accounted for, rather than just the contexts in which they appear; this results in more precise migrations.

Chapter 7

Grey-Box Information for Synthesis[1]

In Chapters 5 and 6, a system for performing *black-box* oracle-guided program synthesis was described and evaluated. By restricting the contextual information available to guide synthesis, a minimal viable problem statement for the underlying synthesis task could be obtained and analysed. However, Chapter 4 demonstrated that sources of information beyond type signatures, when available, can provide effective priors on the synthesis procedure. The aim of this chapter is to bridge the gap between these two viewpoints: what information can be used in practice to improve black-box synthesis, while retaining as much automation as possible?

The key observation made in this chapter is that the target synthesis oracles that motivate the whole of this thesis are *real components*. Traditionally, oracle-guided synthesis treats the oracles in question as fully abstract objects that produce input-output examples when requested. However, in the context of this thesis, the oracles in question are the actual library functions. When an input-output example is generated, the library function is executed on that input.

This chapter presents a method for *grey box* program synthesis. Compared to black-box synthesis, where the internal structure of the oracle is entirely concealed, a grey box context is one where dynamic observations of component behaviour can be made as the component is executed on each different input. These observations cannot specify correctness as input-output examples can, but do restrict the space of possible solutions to a given synthesis problem.

The design and implementation of Haze, a program synthesiser designed to integrate grey-box information is presented. This design is then evaluated on a substantially expanded set of synthesis benchmarks, showing improved performance and

[1] This chapter is based on published research in Collie and O'Boyle [6].

scalability against Presyn, as well as against more specialised binary lifting tools.

7.1 Introduction

Existing automated software engineering techniques often rely on component specifications being made available; for example, to perform tasks such as migration between two APIs. Unfortunately, it is often the case in practice that these specifications are unavailable. Frequently, components are often only available in a low-level form [169, 170], or are implemented as proprietary libraries or specialised hardware [111].

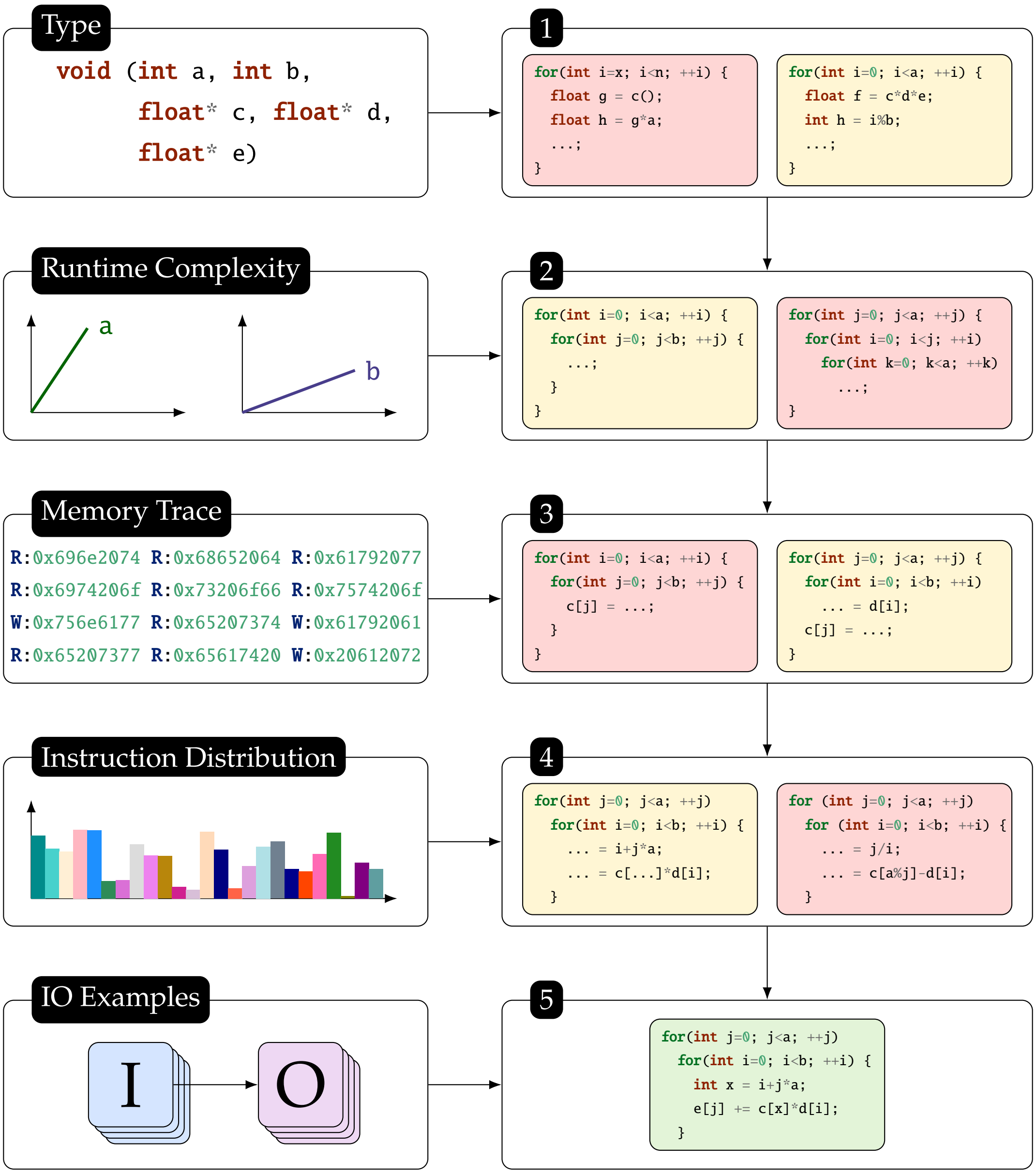
This has led to a large body of work aimed at recovering high level code from components, to allow them to be specified and used to drive software evolution. At the heart of these approaches is the idea of *lifting*, where high-level semantic information lost during the implementation of a component is recovered or reconstructed [171].

However, existing lifting schemes are often ill-suited to recovering real-world programs. Some are overly *specific*, limited to targeting just one application domain (e.g. image processing [96] or finite string automata [111]). Others, aiming at greater generality, are too *weak*, restricting the lifted programs to a few lines of loop-free code [66, 84] or Turing-incomplete DSLs [79]. Still others push the problem to the developer, either requiring *external* access to the binary [100]; a full specification and a oracle that can provide counterexamples [51, 54]; or relying on user-provided annotations to help the recovery [2].

Ideally, a more general lifting scheme would be available, that is able to automatically recover complex real-world specifications (programs) from components. It should require minimal external specification, human assistance or static binary information.

This chapter presents a new approach to lifting an unknown component using *grey-box* program synthesis. Unlike other synthesis approaches, which construct a program solely based on a static specification [172, 173], it exploits, wherever possible, dynamic observations of component behaviour. In particular, it is able to use the execution time, memory traces and performance counters of a component to guide the synthesis of an equivalent program. By doing so, more complex components can be lifted and used to drive software evolution.

The remainder of this chapter is structured as follows. First, in Section 7.2, an overview of the problem scenario and proposed solution is given with reference to a worked example. Then, Section 7.3 details the implementation of Haze, a grey-box program synthesiser designed to attack this problem. Section 7.4 and Section 7.5 present a set of synthesis benchmark problems, and compare Haze's performance on them against a set of related synthesisers and lifters.



```
void (int a, int b,
      float* c, float* d,
      float* e)
```

```
for(int i=x; i<n; ++i) {
  float g = c();
  float h = g*a;
  ...;
}
```

```
for(int i=0; i<a; ++i) {
  float f = c*d*e;
  int h = i%b;
  ...;
}
```

```
for(int i=0; i<a; ++i) {
  for(int j=0; j<b; ++j) {
    ...;
  }
}
```

```
for(int j=0; j<a; ++j) {
  for(int i=0; i<j; ++i)
    for(int k=0; k<a; ++k)
      ...;
}
```

```
R:0x696e2074 R:0x68652064 R:0x61792077
R:0x6974206f R:0x73206f66 R:0x7574206f
W:0x756e6177 R:0x65207374 W:0x61792061
R:0x65207377 R:0x65617420 W:0x20612072
```

```
for(int i=0; i<a; ++i) {
  for(int j=0; j<b; ++j) {
    c[j] = ...;
  }
}
```

```
for(int j=0; j<a; ++j) {
  for(int i=0; i<b; ++i)
    ... = d[i];
  c[j] = ...;
}
```

```
for(int j=0; j<a; ++j)
  for(int i=0; i<b; ++i) {
    ... = i+j*a;
    ... = c[...]*d[i];
  }
```

```
for (int j=0; j<a; ++j)
  for (int i=0; i<b; ++i) {
    ... = j/i;
    ... = c[a%j]-d[i];
  }
```

```
for(int j=0; j<a; ++j)
  for(int i=0; i<b; ++i) {
    int x = i+j*a;
    e[j] += c[x]*d[i];
  }
```

Figure 7.1: The left column shows types of information used for disambiguating synthesis candidates, and the right hand column shows a subset of the space of potential candidates at each step. As more information is obtained, the space is narrowed, with candidates coloured pink discarded due to their incompatibility with newly available grey-box information. Yellow candidates are compatible and pass onto the next stage. The green box shows the successful synthesised candidate that matches the input/output behaviour of the component (dense matrix-vector multiplication).

7.2 Overview

The aim of this chapter is to generalise the black-box approaches to program synthesis developed in Chapters 4 and 5 to include "grey-box" information exposed by actual executions of a component. Because the shared eventual goal of these chapters is to enable the porting and re-targeting of application code using such components, the approach taken in this chapter focuses on the synthesis of components for which there would be a likely performance benefit from re-targeting. These are typically regular, data-centric functions that represent a performance bottleneck within a system.

In the remainder of this section, a high-level example is used to explain the ideas that underpin grey-box synthesis as it is implemented in this chapter, and a comparison is made against other common strategies for program synthesis from the literature.

7.2.1 Example

Consider a scenario (as proposed similarly in Chapters 4 and 5 where an application makes a call to an external library, but a newer, faster implementation for the same task is available from a different library. Migrating the application from the old library to the new is a useful automated software engineering task. However, the source code for either library is not available to help specify and verify the migration.

By using grey-box information gathered from executions of both libraries, synthesis can be used effectively to construct programs equivalent to the relevant library functions. This information is considered incrementally (from least to most device-specific), depending on what can practically be obtained from the particular oracle. Figure 7.1 summarises the gathering of this information for an representative component; reference to it is made throughout this section.

Type Signature The most important type of information is the type signature of the underlying component. While it is not obtained in the same way as the other types (the type signature is static and available up-front, rather than as the result of observing dynamic component executions). Having the type signature available allows a synthesiser to interpret the structure of data being passed into and out of the component; this in turn allows input examples to be constructed automatically for the component. However, it does little to narrow the search space of potential solutions. At stage ❶, only egregiously incorrect candidates can be ruled out, and many possible programs with the correct signature remain in consideration.

Runtime When executing a component to collect input-output (IO) examples to verify observational equivalence against candidate synthesised programs, the elapsed runtime of the component can be easily observed at the same time. The component must be executed anyway to observe its behaviour, and so observing its runtime adds very little overhead to a synthesis workflow.

Runtime complexity information gives a partial insight into how the internal component implementation uses the parameters it is passed. For example, at stage 2 it is observed that the component's execution time scales linearly in the value of both parameters `a` and `b` (i.e. it is both $\in \Theta(a)$ and $\in \Theta(b)$).

This observation allows the space of potential solutions to be narrowed significantly; the candidate in the pink box at stage 2 can be discarded as it has complexity $\Theta(a^3)$, while the yellow candidate has complexity $\Theta(ab)$ and can therefore be retained.

Memory Traces For many types of component it is possible to observe a trace of the memory addresses accessed during its execution. The mechanism by which this observation can be carried out depends on the precise implementation of the component oracle. For example, architecture-specific binary tracing could be used when the component is an executable or library, or a JTAG interface when it is implemented in hardware.

If such traces are indeed available for a component oracle, the pattern of memory addresses accessed during each execution can be used to refine the space of potential candidates. For example, the traces at stage 3 can all be observed to contain patterns of accesses to the arrays `c` and `d`. This rules out a substantial proportion of the control-flow structures considered previously: the candidate in the pink box can be safely discarded as it makes no reference to array `d`.

Performance Counters Given the partial program structure above, the final step in the synthesis process is to work out the actual computation being performed by the component (i.e. instantiate a program sketch into a complete program). Many synthesisers perform an expensive enumerative search at this point if they cannot discharge the process to a formal method such as an SMT solver. Reducing the number of programs considered during this exponential search is an important mechanism by which the entire synthesis process can be optimised.

If the component conceptually executes sequences of instructions, the distribution of the type of instruction is an important constraint on what programs should be considered in this enumerative search process. For example, at stage 4, it is observed that add and multiply instructions are approximately equal in frequency. All other types of instruction occur infrequently or not at all, and can therefore be discarded. From this observation, the candidate in the yellow box is therefore far more likely to be correct than the one in the pink box (which performs several operations that the component does not, such as divisions and modulos).

Input-Output Examples Finally, the input-output examples gathered from the component can be used to authoritatively identify a correct solution through observational equivalence. At stage 5, the collected examples show that the correct solution is the code in the green box (a general matrix-vector multiplication).

7.2.2 Synthesis for Lifting

The grey-box approach to synthesis presented in this chapter can be compared to other common techniques used for lifting programs from specifications. Typically, existing approaches are either black- or white-box (i.e. component implementations unavailable or available, respectively), and are often multi-modal. Grey-box synthesis combines aspects of all of these styles.

White-box Synthesis In classical synthesis, an external formal specification is provided which can be used to direct the construction of programs via, for example, counterexample-guided search (CEGIS) [51, 54]. Typically, this type of synthesis is used when the aim is to lift a high-level specification from a low-level implementation, and *both* levels of abstraction permit verification or interpretation through formal tools such as SMT solvers.

These systems can be characterised these as *white-box* schemes, because extrinsic specifications are provided for solutions, and the structure of candidate solutions can be introspected on to enable formal verification (that is, all the internal details of both the component and solution are known).

Black-box Synthesis An alternative approach is *black-box* synthesis, often referred to as *programming by example* (PBE) in some problem domains. Here, specifications are simply input-output examples that provide pointwise constraints on solution behaviour. The internal structure of the target component is not known or available, and so formal verification of candidates against the component is not possible. This means that observational equivalence is often the strongest possible correctness guarantee. There exists a substantial, often machine learning-based body of work in this area [77, 80, 141, 142].

Unfortunately, from a systems perspective, purely example-driven, black-box synthesis is ill-suited to the complexity and specifications found in real-world problems, leading to the proliferation of domain-specialised lifters that require substantial assumptions to be made of the behaviour of their target components [96].

Multi-Modal Specifications Combining multiple specifications for a single synthesis problem has been explored in software engineering tasks (such as API migration [165]). By combining different types of specification, the solution search space can be constrained in several different directions at once; this may not be possible when considering only one modality.

Much of the recent work on multi-modal synthesis has focused on neural techniques, which excel at reducing diverse modalities into homogeneous internal representations. For example, the use of natural language descriptions alongside IO examples has recently proved fruitful in neural program synthesis [84, 92]. However, such approaches are typically weak (the programs are restricted to small DSLs), specialised (string processing and regular expressions respectively), and require external assistance in the form of human-provided text descriptions.

Because grey-box synthesis assumes *partial* availability of a component's structure through behavioural observations, but does not assume that formal verification tools and workflows are practical, it represents a conceptual middle-ground between black- and white-box synthesis. This is an intuitive space to occupy: typically, black-box scenarios do not arise as a result of deliberate obfuscation on the part of the component vendor. Rather, they are simply the result of it being *easier* not to provide full specifications for components. Using the partial information left behind as a byproduct of this process is a pragmatic design choice intended to take advantage of as much contextual information as possible.

Similarly, grey-box synthesis adopts the most important facet of multi-modal synthesis: the use of multiple distinct types of specification for a problem, with some of the types providing "non correctness" information that does not fully specify a solution. The distinct element of grey-box synthesis from typical multi-modal systems is that these specifications are obtained from observations of component behaviour, rather than from external sources.

7.3 HAZE: Grey-Box Synthesis

This section describes the implementation of HAZE, a synthesiser designed to demonstrate the grey-box synthesis methodologies introduced previously. It is structured as follows: first, in Section 7.3.1, an overview of HAZE's core synthesis algorithm and workflow is given, along with comparisons to relevant aspects of ANNOTE and PRESYN.

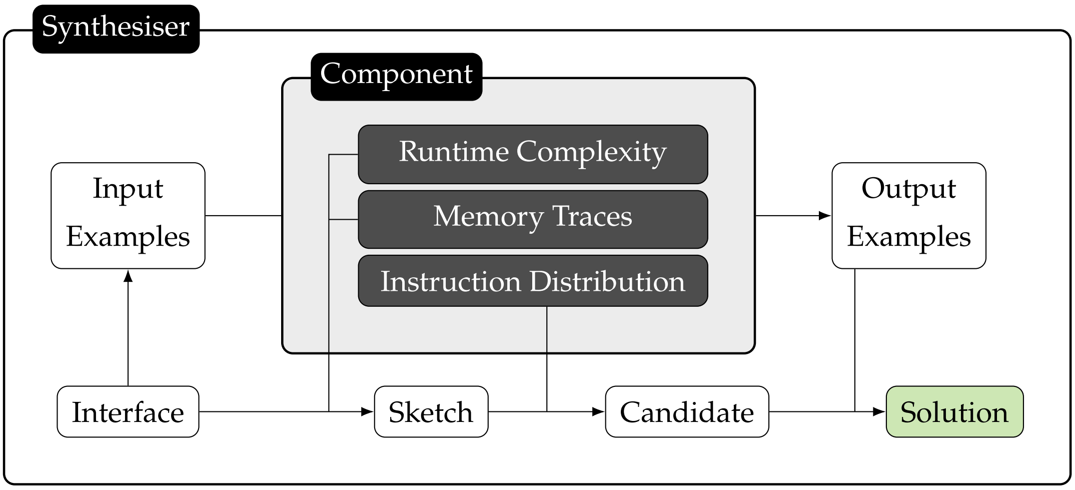


Figure 7.2: Full system diagram showing how information flows through the stages of our synthesiser.

Then, in Section 7.3.2, each of the sources of grey-box information consumed by Haze is introduced, along with a detailed description of how that information is used to narrow the search space of candidate programs. Finally, Section 7.3.3 analyses the correctness and safety of Haze's synthesis procedures.

7.3.1 Overview

Figure 7.2 shows an overview of Haze's approach. From a component's interface (oracle access to call the component, and a type signature), inputs can be generated and used call the component. Doing so returns corresponding outputs, which are used to form input-output examples for later observational equivalence testing. By varying inputs in a structured manner, it is possible to observe how they affect execution time; the execution time is used to inform the construction of a sketch (in the common sketch language described in Section 3.5). If available, memory traces and performance counters can be used to further refine the sketch and provide constraints on the potential data-flow in candidated programs.

Based on these constraints, candidates are iteratively sampled from the space of possible programs, until one is identified that matches the input-output examples generated previously from the component. Many such examples can be generated to ensure that observational equivalence is a sufficiently strong correctness criterion. In Section 7.3.3, the methodology used to generate sensible inputs is examined, as well as the use of model checking to verify formally that programs are correctly synthesised.

Algorithm 5 summarises the core synthesis algorithm used by Haze, analogously to Algorithm 4 for Presyn. The remainder of this section explains the steps of this

Algorithm 5 Haze's core synthesis loop, assuming that all possible grey-box information is available for the component oracle in question. The acquisition of runtime complexity and instruction distribution information is performed at the same time as input-output examples are generated for the function, and for brevity is not included in this figure.

```
function Synthesise($target$)
    $sketches \leftarrow$ {all sketches satisfying runtime complexity of $target$}
    $traces \leftarrow$ {memory traces from executions of $target$}
    $scores \leftarrow \emptyset$
    for each $sketch$ in $sketches$ do
        $proxy \leftarrow sketch$ extended with dummy memory accesses
        $sketchTraces \leftarrow$ traces from executing $proxy$
        $scores[sketch] \leftarrow \text{mean}_{s \in sketchTraces}(\text{max}_{t \in traces}(\text{Align}(s,t)))$
    end for
    $ranked \leftarrow \text{sort}(scores)$
    loop
        $sketch \leftarrow$ geometric sample from $ranked$
        for each $hole$ in $sketch.holes$ do
            recompute live values and dependencies
            $op \leftarrow$ opcode sampled proportionately to $target$
            $live \leftarrow$ sorted live values at $hole$
            $v_1, \ldots, v_n \leftarrow$ samples from $live$
            $v \leftarrow op(v_1, v_2, \ldots, v_n)$
            Rauw-NT($hole, v$)
        end for
        $program \leftarrow sketch.compile()$
        if $program$ satisfies all $examples$ then
            return $program$
        end if
    end loop
end function
```

algorithm and associated implementation in detail.

7.3.2 Implementing Haze

This section explains in detail the synthesis algorithm listed in Algorithm 5, working through the grey-box information and synthesis steps taken by Haze in the order they are used.

7.3.2.1 Type Signature

The first source of information used by Haze is the type signature of the target function. Having access to the type signature allows safe, structured interpretation of the inputs passed to or outputs received from the target component. Without a type signature, understanding the structure of input-output examples would be intractable.

The assumption that access to the target component's type signature is available is shared with Annote and Presyn, and is not an onerous additional requirement given that a callable oracle must somehow be implemented for the target component.

7.3.2.2 Generating Inputs

Haze, like Annote and Presyn, uses observational equivalence over a set of randomly generated input-output examples to specify correctness for candidate synthesised programs. For those implementations, demonstrating that the candidate and target function are equivalent is the only role of the input examples; branch coverage experiments (in Section 6.5.3) show that randomly-generated examples do so adequately for typical synthesis targets.

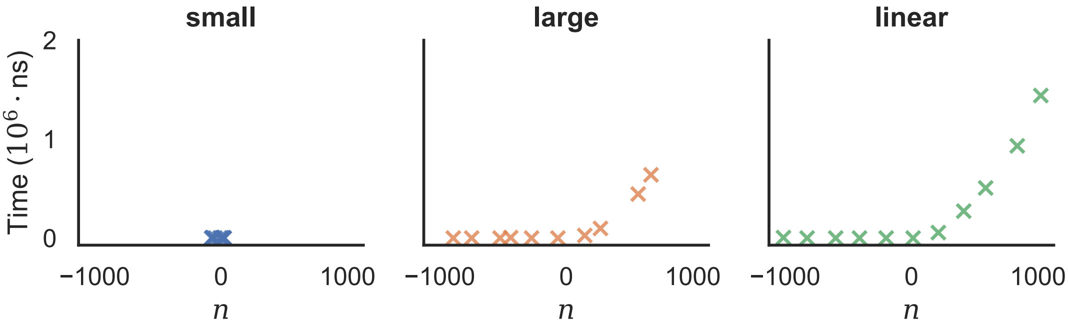
However, Haze requires a broader guarantee from the input examples it generates. Rather than only demonstrating observational equivalence, the input examples must be able to induce a broad range of dynamic behaviours from the target component; these behaviours are not captured by input-output examples and so are not verified by code-coverage experiments.

As an illustration of the problem, consider the *execution time* of the target component. This is one of the sources of grey-box information used by Haze for synthesis; its exact interpretation and use is expanded on in the following section. It is intuitive that if the inputs generated for a component are insufficiently broad, then it may not be possible to fully observe variations in its execution time.

Figure 7.3b demonstrates this effect for a simple synthesis problem taken from the Simpl benchmark suite [70]. The execution time of this example, on randomly-generated inputs from three different schemes is plotted. For the "small" scheme used

```
int triangle_sum(int n) {
  int r = 0;
  for (int i = 1; i < n; ++i)
    for (int m = 1; m < i; ++m)
      r += m;
  return r;
}
```

(a) Example synthesis problem, with time complexity of $O(n^2)$ in its single parameter n [70].



(b) Execution times of `triangle_sum` plotted against 10 different input values of n, for three input generation strategies: small ($[-64, 64]$) and large ($[-1024, 1024]$) uniform random, and linearly spaced.

Figure 7.3: Example showing the benefit of a partially-deterministic, interpretable input generation strategy when considering dynamic behaviour of synthesis oracles.

by Presyn (and verified to be adequate with respect to branch coverage), no inputs large enough to confidently observe a variation in execution time are generated; for $n \in [-64, 64]$, the execution time is within measurement error and appears approximately constant.

One possible solution to this problem is to expand the sampling range; again in Figure 7.3b, the "large" regime uses the range $[-1024, 1024]$, which is large enough to show clear increases in execution time. However, in this case the sampled values are clustered around $n = 0$, with only two exceeding $n = 300$. As a result of this nondeterminism, a large number of input values need to be sampled to consistently and confidently observe the quadratic complexity of `triangle_sum` (Figure 7.3a).

The solution adopted by Haze, as implemented in this chapter, is to combine a randomly-generated scheme with a deterministic scheme that generates linearly spaced

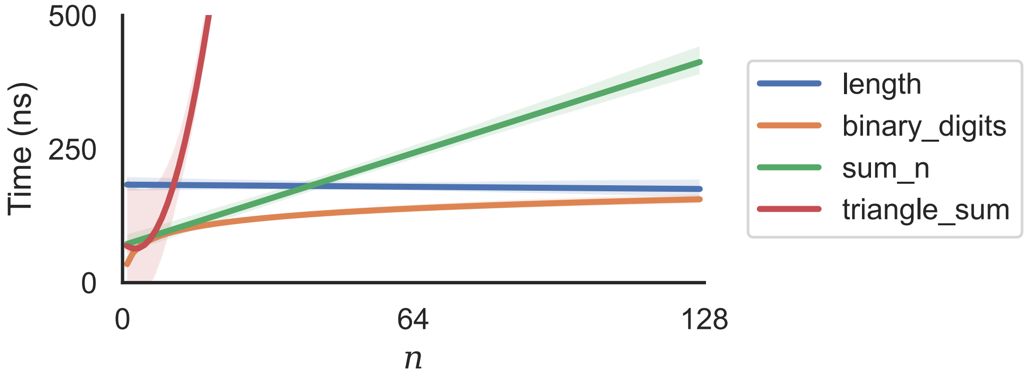


Figure 7.4: Comparison of fitted runtime complexity models for four functions in the dataset used to evaluate Haze. Using the linear sampling regime, complexity in a single parameter can be reliably inferred (shaded regions show 95% confidence intervals).

data points. Doing so produces the clearest, most consistent results in Figure 7.3b. While it is possible that this combined scheme does not fully capture the potential range of behaviours in some other scenarios, it was judged to be adequate with respect to the potential behavioural variations exhibited by Haze. For additional sources of grey-box information in future work, additional structure in the input data may be required to fully exercise the dynamic behaviour of components.

7.3.2.3 Performance Models

There is a strong, intuitive correlation between a program's control flow structure and its runtime performance characteristics; if particular control flow structures are present in a program, then it is likely that they imply a portion of the runtime complexity class of that program. For example, if a loop of the form `for(int i=0; i<N; ++i)` is present, then that program is likely to have complexity of the form $O(\mathtt{N} \cdot \ldots)$.

Automatically computing a program's runtime complexity in terms of its individual parameters is a well-established research problem; being able to do so enables a number of useful tasks. For example, by modelling runtime complexity, Calotoiu et al. [174] predict scalability issues in high-performance workloads that would not become evident before deployment to a production environment.

Haze uses the observed runtime performance of a target component to search for combinations of fragments that are likely to produce that runtime complexity. To do so, a new collection of fragment templates derived from (but substantially extending) the library used by Presyn was constructed, using the sketch language semantics introduced in Section 3.5. Each fragment in this collection was annotated with its runtime complexity, and the fragment compilation semantics were extended with rules for how their *complexities* compose.

To formally describe the runtime performance of a synthesis target, a notational framework similar to the Performance Model Normal Form (PMNF) of Calotoiu et al. [174, 175] is used. This normal form provides a parameterised equation that describes the runtime performance of a program in terms of each of its scalar inputs $x_1, \ldots, x_m$ individually:

$$f(x_i) = c_0 \cdot c_1 x_i^{p_i} \cdot c_2 \log_2^{q_i}(x_i) \tag{7.1}$$

HAZE fits a model of this form for a synthesis target by first recording its runtime at different values of each scalar parameter (using the combined linear-random input generation strategy described in the previous section). Then, exponents $p_i, q_i \in \mathbb{Q}$ and constant factors c_0, c_1, c_2 are regressed against the observed performance to produce the best-fitting performance model for each input parameter. Finally, the models are grouped into broader categories for the purposes of sketch search.

7.3.2.3.1 Sketches The library of fragment templates used to drive HAZE's search for program sketches is based on intuitive, common patterns likely to occur in programs (and that can be mapped readily onto their runtime complexity in terms of template parameters). This library was based on prior work to assemble a set of common *idiom descriptions* written in the IDL DSL [112]. In this context, an idiom represents a full computation or class of computations (e.g. "dot product" or "histogram"). The description of each idiom is built compositionally from smaller components (for example, a loop is specified in terms of the description of a single-entry single-exit region with some additional properties); these smaller components are broadly the source of the fragment templates used by HAZE.

From the default library of idioms developed by Ginsbach et al. [112], the following criteria were used to select ones appropriate for HAZE. First, they should be expressed at a similar level of abstraction to sketches in comparable implementations [61, 70]; the aim of HAZE is not to compose entire high-level algorithms. Secondly, they should be readily translatable to a concrete fragment definition (some descriptions cannot easily be). Finally, and optionally, the translated code should depend on a free variable to produce a template rather than a fragment (for example, the loop bound in a for-loop).

Then, each IDL description was translated into a fragment implementation by hand (i.e. for each description, a code generation routine was implemented that produces code satisfying the original IDL description). Each implementation was verified against the original description, and an appropriate complexity class was associated with each

Table 7.1: List of fragment types used by Haze. These can be divided into three broad groups: fragments that perform iteration, fragments that instantiate specific computations over live IR values, and fragments that create non-iterative control flow structures. Fragments marked with (†) are *not* constant-time operations in reality, but their runtime complexity does not depend on sketch parameters; for the purposes of search, they are labelled as $O(1)$ for consistency of implementation. Those marked with ($\star$) do not apply multiplication of complexities under composition.

Group	**Fragment**	**Complexity**		**Description**
Iteration	`bounded(n)`	$O(n)$		These fragments represent traditional C-style loops, and are differentiated largely by their termination conditions. Some (e.g. `collect(n)` include specialised dataflow code to act over arrays, while others are simply control flow with fragment holes.
	`collect(n)`	$O(n)$		
	`delim()`	$O(1)$	(†)	
	`divided(n)`	$O(log(n))$		
	`fixed(n)`	$O(n)$		
	`loop(n)`	$O(n)$		
	`static()`	$O(1)$		
	`triangular(m,n)`	$O(mn)$		
	`until()`	$O(1)$	(†)	
Dataflow	`affine()`	$O(1)$		Computations identified as likely to appear in the body of nested control flow structures.
	`index()`	$O(1)$		
	`indirect()`	$O(1)$		
Control	`if()`	$O(1)$		Non-iterative control flow, as well as mechanisms to plumb several fragments together intuitively, and specific data-flow placeholders.
	`if_else()`	max(_,_)	($\star$)	
	`block()`	$O(1)$		
	`empty()`	$O(1)$		
	`seq()`	_+_	($\star$)	

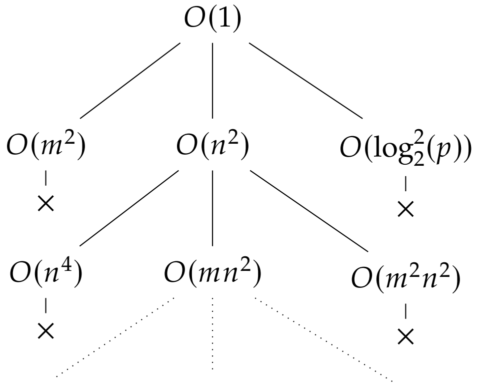


Figure 7.5: Partial search tree for a sketch with overall complexity $O(mn^3\log_2(p))$ in parameters m,n,p. Each labeled node in the tree represents the accumulated complexity; paths through the tree are ruled out as their associated complexity becomes unviable.

one by inspection (in terms of its free variables). Table 7.1 lists the full set of fragments collected.

The set of fragments used by HAZE to begin synthesis represents a potential source of bias, and implicitly encodes assumptions about the structure of the space of programs targeted. While not fully general, the set of collected fragments in Table 7.1 can describe the majority of performance-sensitive functions for which migration to an optimised implementation is a desirable goal. Additionally, as the set of fragments is derived from an existing set of idiomatic patterns, the potential for bias towards a particular set of synthesis problems is reduced.

7.3.2.3.2 Tree Search A set of possible program sketches can be constructed from complexity-labelled fragments using a simple tree search procedure. Initially, the fragments are partitioned into two groups: those that have $O(1)$ complexity in terms of the function's parameters, and those that have greater than $O(1)$ complexity.

For each parameter's performance model, a set of fragments that *could* contribute to that model are identified. For example, a sketch with complexity $O(n^2)$ could appear if the overall complexity in n was $O(n^3)$, but not if it were only $O(n)$ (for example). Then, compositions of the sketches for the full set of parameters are enumerated, pruning compositions that exceed the total required complexity (see Figure 7.5 for an illustration of this).

7.3.2.4 Memory Traces

For many components, it is possible to obtain a trace of the memory addresses they access during their computation. While this is a more specific notion than execution time (and therefore is less likely to be available for every possible type of component oracle), it is well-defined for many of the most common types. For example, an executing library or binary can have its execution instrumented with a dynamic binary instrumentation framework such as Pin [176], or a hardware device can be interrogated using a standard JTAG interface.

Haze uses memory traces to *rank* the sketches produced previously by how likely they each are to produce a correct solution. No information is discarded by Haze at this point; if a sketch is identified as a possible candidate previously, then it will remain a candidate after this ranking. As a result, the memory tracing step is optional, allowing for components for which it cannot be implemented to be analysed.

7.3.2.4.1 Producing Traces First, define a memory trace $\overline{T}$ to be an ordered sequence of pairs:

$$\overline{T} \triangleq ((m_0, a_0), \dots (m_n, a_n)) \tag{7.2}$$

where each m_i indicates the type of access made (read or write), and each a_i indicates the address of that access. To minimise the burden of supplying this information (and therefore to maximise the number of potential component oracles that can be used with this stage), neither the specific value read or written, nor the size of each access are recorded.

The concrete memory addresses in a trace will be different even between executions of the same program, and so the traces need to be normalised to a common representation that can be compared across executions. First, a set of base addresses in the trace are identified, which all accesses will be expressed relative to. To do so, addresses that appear at the beginning of contiguous sequences of accesses are identified; these addresses are *base* addresses. Then, the closest matching base address is subtracted from each access to produce a (base, offset) pair, such that no offset is negative.

The assumptions made here are similar to, but weaker than those made by related work in the area of memory trace analysis [177]; where that work requires the issuing instruction for each memory operation to be identified, the eventual alignment and ranking of traces induced by Haze is an approximate procedure and requires only similarity (rather than full equivalence) between the sketch and reference traces.

Next, code is inserted into each sketch that imitates memory accesses at appropriate points. To do so, a read instruction is added whenever a memory address becomes live in the sketch code (i.e. when the sketch computes an offset to a pointer), and a write before the point that address is no longer live. Heuristically, this represents a simple usage pattern where values are loaded from memory, operated on by the (as yet unsynthesised) sketch body, then written back to memory. Each sketch is executed several times with this instrumentation, selecting randomly chosen assignments for branch conditions. This produces a set of memory traces for each sketch that represent a non-exhaustive subset of its possible behaviours.

7.3.2.4.2 Scoring & Ranking Traces The Gotoh sequence alignment algorithm [178] is used to identify potentially conserved regions between a sketch and program trace. This algorithm was selected specifically to deal with trace sections where long gaps appear; a likely scenario when considering traces generated from executing programs. The resulting alignment score represents how well the behaviours of the program trace are explained by the sketch trace. No alignment will score perfectly, but better correspondences will align more closely.

From the target program, a set $\mathbf{T}$ of ground-truth traces is available, as well as a set $\mathbf{T}_S$ of traces from each sketch S. From this, the aim is to select the sketch that explains the set of program traces best on aggregate. For an individual sketch trace $s \in \mathbf{T}_S$, define a scoring function $q(s)$:

$$q(s) \triangleq \max\{\text{align}(s,t) \mid t \in \mathbf{T}\} \tag{7.3}$$

That is, the score for an individual sketch trace is simply its *best* alignment score with any program trace. This is intuitive; program traces may correspond to different executions, and so it cannot be expected that a single sketch trace explains all possible program traces.

Then, for a sketch S with traces $s_1, \ldots, s_N$, define an aggregated scoring function $Q(S)$:

$$Q(S) \triangleq \frac{1}{N}\sum_{i=1}^{N} q(s_i) \tag{7.4}$$

Sketches that score highest with respect to Q are those that on average, generate traces that best explain a trace from the set of program traces $\mathbf{T}$.

Finally, sketches are ranked from 1 to N according to their Q score, then assigned a sampling probability using a geometric distribution:

$$\mathbb{P}(\text{sketch } k) \triangleq (1-p)^{k-1}p \tag{7.5}$$

Haze sets $p = 0.5$, but p can be varied to reflect different priors on the structure of the ranked sketch set (i.e. if more sketches are likely to be viable, decrease p).

7.3.2.5 Instruction Type Distribution

Many synthesis procedures ultimately terminate in a resource-intensive enumerative search for a sequence of instructions or operations that comprise a correct solution. For Haze, this entails searching for an instantiation of each hole in a program sketch with a concrete program value.

To accelerate this search for concrete instructions, Haze considers the *observed* distribution of instruction types dispatched by the target component as it executes. By doing so, the synthesis process can be biased towards programs that produce similar distributions to the target.

More precisely, for a set of possible instruction opcodes:

$$O \triangleq \{o_0, o_1, \dots, o_n\} \tag{7.6}$$

Haze constructs a cumulative count C across all executions of the component:

$$C : O \mapsto \mathbb{N} \tag{7.7}$$

of how many times each opcode was observed during execution. From this count, Haze then constructs a probability distribution $\mathbb{P}$ satisfying, for some $\delta \in [0, \frac{1}{n})$:

$$\forall i \,.\, (\mathbb{P}(o_i) - \delta) \sim C(i) \;\wedge\; \mathbb{P}(o_i) \geq \delta \tag{7.8}$$

7.3.2.5.1 Search The space of potential instantiations for a set of holes is combinatorially large. A type-safe instruction opcode is sampled for each hole from the probability distribution described above, then possible arguments for that instruction are enumerated using a set of heuristics (e.g. more local arguments are preferred to distant ones). The search space is restricted as values are assigned to holes as a result of Rauw-NT's type propagation mechanisms (see Section 3.6.1).

This search for instructions from the observed instruction type distribution is the final step in Haze's core synthesis algorithm (as presented in Algorithm 5). The generated candidate programs can now be tested for correctness using the set of collected input-output examples, and eventually returned to the user.

7.3.2.6 Assumptions

When implementing and evaluating any program synthesiser, it is important to note the assumptions made of the underlying program space. Other than the assumptions implicitly encoded by the structure of its fragment library, Haze requires that the type signature of a synthesis problem uses only primitive types (integers, characters, floating-point), or pointers to those types. Structures composed of these types are not

```
int fact(int n) {
  int r = 1;
  while (n-- > 1) r *= n;
  return r;
}
```

Figure 7.6: Example synthesis problem demonstrating integer overflow; one possible cause of safety issues when generating inputs [70].

supported, but would not require conceptual extensions to do so. No fragments in Haze's library deal with pointers to pointers (e.g. `int**`), but could be implemented similarly to those described previously.

The testing framework used by Haze generates non-aliased, word-aligned pointers; the trace alignment phase does not explicitly depend on these, but is likely to produce incorrect alignments through misidentification of base addresses were a different input generation strategy to be used that did so.

An assumption shared with many synthesisers in the literature is that randomly-generated input values are sufficient to exercise all possible behaviours of the reference function used; Section 7.5.4 examines how this assumption can be validated on the dataset used to evaluate Haze.

7.3.3 Safe Synthesis

Given that little is assumed of the target components Haze targets for synthesis, it is critical that the synthesis process is safe. In particular, it is important to consider the safe generation of component inputs and the correctness of synthesis (i.e. how can solutions be verified, and is input-output equivalence a sufficient standard to establish correctness?).

7.3.3.1 Safety and Bounding

Not all programs can safely accept all possible values; some programs may exhibit incorrect or unsafe behaviour when called with certain inputs. For example, the program listed in Figure 7.6 will overflow a standard 32-bit integer for values of n greater than 12. As well as integer overflow, there are safety issues for programs that access memory (for example, if a parameter is used as an index into an array also passed to the function). Where possible, oracles can (using implementation-specific methods such as checking an interrupt register) report error states as a single bit flag; when this is the case, the faulting input is not included in the sets of IO examples used to specify correctness for a problem.

Fully random input generation methods struggle to concisely capture the behaviour of reference functions in the presence of these safety issues. For example, the "small" and "large" fully random regimes in Figure 7.3b will both generate uninteresting ($n \leq 1$) or unsafe ($n > 12$) inputs, the majority of the times they are invoked. This means that a large number of attempted inputs must be tried to produce the required number of working inputs.

Haze applies the heuristic that programs exhibit such unsafe behaviour for a continuous, infinite range of values either above or below a threshold. Under this assumption, when a *run* of unsafe behaviour is observed, it backs off and retries input generation with the first observed unsafe value as the new upper bound on input. Intuitively, this compresses the same number of linearly spaced inputs into the safe interval supported by the oracle.

7.3.3.2 Verifying Synthesised Programs

In the programming by example-like setting of grey-box synthesis, the only possible correctness specification for a synthesis problem is equivalent behaviour over a set of input-output examples. This is observational equivalence: does the solution's behaviour look the same for all the input examples attempted? Similarly to Annote and Presyn, Haze is unable to provide a general assurance of correctness that is stronger than observational equivalence.

However, if the underlying code for a particular problem is available, formal verification tools can be used to provide stronger and more precise guarantees on the correctness of a solution. KLEE [131] is a suite of tools for performing *symbolic execution* on LLVM IR programs. By doing so, it is possible to efficiently discover code paths within, or inputs to a program that cause errors such as integer overflow or out-of-bounds accesses to occur.

The application of KLEE used by Haze on reference programs with available code is to identify any inputs that cause the two functions (i.e. a reference implementation and a candidate synthesised solution) to produce different behaviour. A modified version of the basic function equivalence procedure suggested by Ramos and Engler [179] is applied to do so, with the key addition of a modernised implementation of the symbolic floating point support from KLEE-FLOAT [132]. This addition is necessary as the majority of Haze's evaluation problems are stated over floating point numbers, and cannot be symbolised using mainline KLEE.

7.4 Experimental Setup

To evaluate Haze, a dataset of synthesis problems was constructed that extends the evaluation set for Presyn described in Section 5.6. This extended set contains the same original synthesis benchmark problems as the Presyn set, but increases the difficulty of the problems derived from real-world library functions.

As well as comparing the success of Haze purely against other program synthesisers, a comparison to Helium [96] is made to evaluate how well Haze performs against heavily domain-specialised binary lifting tools. This section describes the selection process for both the dataset used to evaluate Haze, and the implementations against which it is compared.

7.4.1 Dataset

The starting point for Haze's evaluation set is a set of 112 synthesis problems used to evaluate Presyn, as described in Chapter 5. This set of problems covers a range of domains (mathematical primitives, vector operations, string manipulations), and subsumes the evaluation sets for a number of other synthesisers [70, 72, 77, 84]. Full details of how this dataset was assembled are given in Section 5.6.

This dataset was extended with a further 78 synthesis problems intended to challenge Haze's ability to lift real-world library functions. Grouped by their respective origins, these problems are as follows:

Benchmark Kernels Suites of performance benchmarks are often used to evaluate binary lifting techniques [177]; three such suites were identified that provide a natural increase in complexity from the synthesis problems above. These were UTDSP [180], DSPStone [181] and PolyBench [182] ($N = 18, N = 15, N = 30$ respectively). The 63 new problems represent individual benchmark *kernels* that are more challenging than typical program synthesis problems, with the PolyBench set in particular containing some with far greater complexity than any existing synthesis techniques are able to scale to.

Specialized Domains Image processing functions and tensor manipulations provide two novel sets of evaluation problems that model *real-world* code, in two domains relevant to the synthesisers Haze is evaluated against. A set of $N = 10$ image-processing functions derived from [96] were evaluated; these covered both low-level implementation details of individual functions, as well as high-level heuristic descriptions (such as the "Blend" and "Filter" categories identified by Ahmad et al. [98]).

Understanding and manipulating tensor operations is an important task in optimising many contemporary compute-bound workloads; $N = 5$ common tensor manipulations were generated as additional problems using the Taco compiler [26, 183]. Each manipulation had a different set of optimizations and scheduling transformations applied to it in order to evaluate how well HAZE responds to changes in implementation detail; for example, tiling of loop nests.

7.4.2 Experiments

HAZE is evaluated against the following leading program synthesisers and lifting approaches; PRESYN's results (see Section 5.8) show that it outperforms several traditional synthesisers on the set of 112 synthesis problems from its own dataset given above; SKETCHADAPT [84] is a leading example of a neural synthesiser, and Helium [96] is a domain-specific *lifter* for image-processing applications recovering high-level code from dynamic observations of an application executing.

Because these synthesisers and lifters target diverse problem domains and specification formats, running them all on the same set of benchmarks is challenging. This is a problem shared by other work in synthesis [46], and the approaches taken to normalise the additional 78 problems are similar to those used for PRESYN's evaluation set in Section 5.6.

The core of HAZE's cross-evaluation is a comparison of which implementations were able to synthesise (or lift) each benchmark problem. To do so, each benchmark problem was adapted for the specific input requirements for each implementation. For example, PRESYN required training examples of previous syntheses, and Helium required inputs and outputs to be read from image-like files.

All of the tools described above ran on the evaluation system without modification except Helium, which was not possible to build successfully using its original toolchain. To resolve this, a port of Helium's core that supported Linux binaries was produced. This porting process was substantial in places, especially when dealing with platform-specific instrumentation tools.

7.5 Results

This section evaluates and analyses HAZE's synthesis performance against existing synthesis and lifting schemes. First, the number of problems from the benchmark suite listed previously that can be successfuly synthesised is measured. Then, the time taken for HAZE to successfully synthesise these cases is examined, and an evaluation of the correctness of the solutions produced using model-checking is performed. This

Table 7.2: Summary of successfully synthesised or lifted programs across the evaluation dataset for each implementation examined. Columns show the number of successfully synthesised programs from each group for a single implementation.

	N	Presyn	SketchAdapt	Helium	Haze
Presyn	112	96	10	-	103
UTDSP	18	6	-	-	9
DSPStone	15	10	-	-	13
PolyBench	30	2	-	5	14
Image	9	2	-	10	6
Tensor	5	1	-	-	4
Total	190	117	10	15	149

is followed by an analysis of the grey-box information used by Haze, and an ablation study to establish its use for synthesis.

7.5.1 Success Rate

For each synthesis or lifting implementation under test, each of the 190 synthesis problems listed in Section 7.4.1 was attempted (with appropriate allowances to adapt the problem for each one's input format), recording the total successful syntheses for each implementation, as well as a per-group breakdown. These results are summarised visually in Figure 7.7, and listed in full in Table 7.2.

Haze is the best-performing implementation across the entire dataset, and on all but one of the individual problem groups (the image-processing kernels, where Helium's increased domain specialisation allows it to outperform Haze). It is also the only implementation able to synthesise at least one example from each of the benchmark groups.

By integrating multiple sources of grey-box knowledge, Haze is able to outperform comparable implementations across the dataset at large. For example, Haze is able to learn a group of logarithmic-time problems from Presyn's benchmarks more effectively by using runtime complexity information to predict control flow structure (Section 7.3). Presyn fails to predict the control flow structure for these, and so is unable to achieve the same syntheses.

Helium outperforms all other implementations on its own specialised domain of image processing, but fails to generalise across the dataset. SketchAdapt successfully synthesises a small number of simple examples,but is unable to scale in complexity or

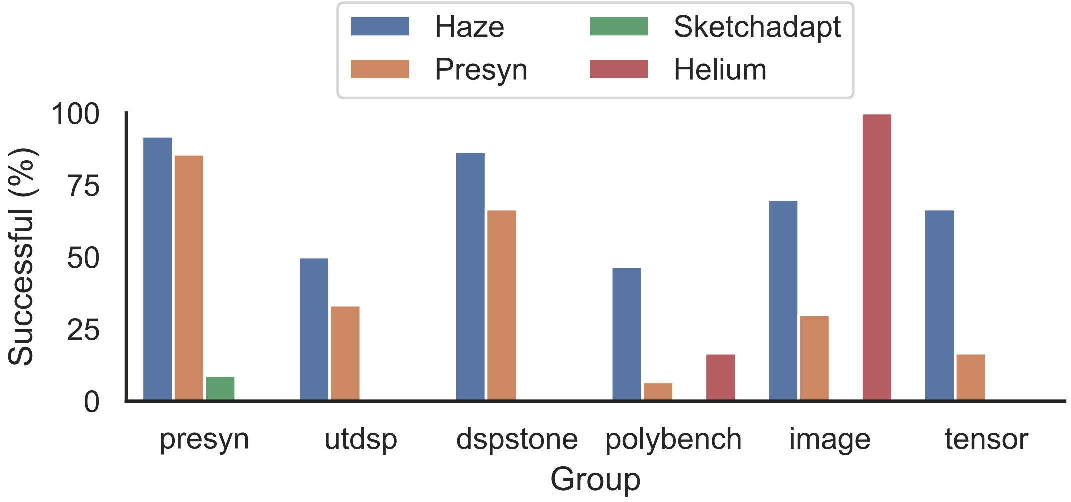


Figure 7.7: Synthesis success rate for each synthesiser and lifter evaluated, across the set of problem sources used to evaluate Haze.

across domains. Given that it is targeted at list-processing tasks, this is not surprising.

Where Haze fails, it does so most commonly because the sequence of instructions required to produce a correct solution to a problem is too long; even with the help of a probability distribution over their types, the space is too large to search effectively. Less commonly (and primarily in the PolyBench and UTDSP groups), the required control flow structures are not implementable using Haze's library of fragments. Additional fragment definitions would allow Haze to synthesise these problems, at the expense of increased specialisation and bias in its fragment library.

7.5.2 Analysis

To directly compare the success of each implementation across the entire evaluation dataset, the required number of *fragments* and *instructions* used by Haze for each successful synthesis were examined. These measures are partially correlated, and both measure the complexity of a synthesis problem. Where Haze fails to synthesise a solution achievable using another implementation, one was written by hand as an oracle. Figure 7.8 shows these measures plotted for each implementation.

Other existing synthesisers (Presyn, SketchAdapt) show similar clusters centered at low instruction counts, indicating the difficulty they experience in scaling to larger problems. Conversely, Helium synthesises very few programs in total, and its cluster is centered at high instruction counts (it does not cover small programs at all). Compared to existing implementations, Haze takes a significant step towards generalising across the entire dataset; its distribution covers the widest range of fragment and instruction

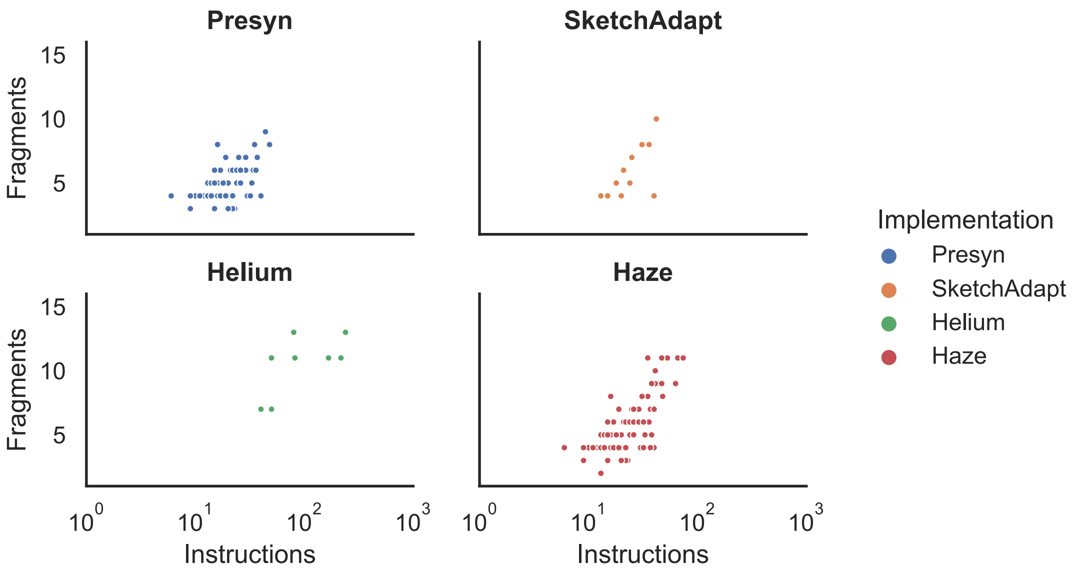


Figure 7.8: The distribution of instruction count vs. fragment count for each implementation's successful syntheses. These are partially correlated metrics of problem complexity for synthesis; more complex *control flow* requires more fragments to produce a correct solution, while more complex *data flow* requires more instructions to do so.

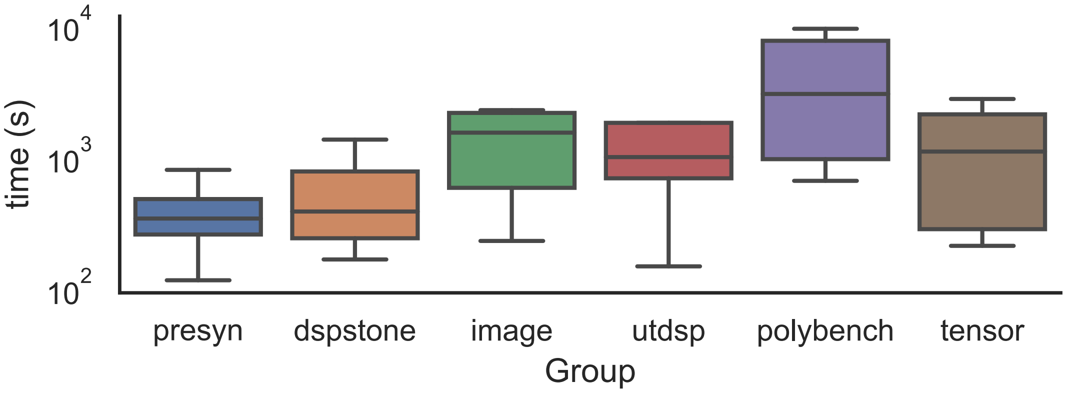


Figure 7.9: Distributions of HAZE's required synthesis time across each group of benchmark problems.

counts.

7.5.3 Synthesis Time

Optimising for synthesis time was not a primary goal when implementing HAZE. For example, no directed search methods are used when generating instruction sequences (other than the distribution of instruction opcodes obtained from the target compo-

Table 7.3: Results obtained by model-checking programs synthesised by HAZE against their respective reference implementations. Many solutions demonstrated minor floating-point inaccuracies that were explicitly not identified by observational equivalence checks. However, beyond these cases no false-positive synthesis results were identified.

	N	**FP Assoc.**	**Bugs**	**Success**
Presyn	103	11	0	100%
UTDSP	9	9	0	100%
DSPStone	13	13	0	100%
PolyBench	14	14	0	100%
Image	6	0	0	100%
Tensor	4	4	0	100%

nent). However, the results in this section suggest that HAZE is usable. All successful syntheses listed were obtained using a 3 hour threshold time on a desktop-class machine, and nearly 80% of these were obtained in under 15 minutes. Figure 7.9 shows the full distribution of time taken for successful syntheses by HAZE. It shows that the new datasets evaluated in this chapter are significantly more complex, with PolyBench providing a particular challenge. Synthesis search time, rather than gathering input-output examples and grey-box information, dominates the total time required to produce a solution (> 99.9%). No problems required more than 1,000 input-output examples to be generated.

On the Tensor benchmark set, we found that standard optimisations applied to the reference implementation oracle (for example, loop tiling) slowed, but did not defeat HAZE's synthesis. Intuitively, this is because the memory trace ranking procedure is less effective when the traces generated by the optimised oracle are different to those generated by sketch candidates (which have a "handwritten" structure when compiled). The underlying complexities are the same in each case, and so the correct fragments are still identified by HAZE. In the future, optimisations could be applied to synthesis candidates to better liken them to the reference implementation.

7.5.4 Validity

In Section 7.3.3.2, a description is given of how the KLEE symbolic execution engine [131] can be used to model-check synthesised programs against a reference implementation (if one exists). Table 7.3 summarises the results of doing so for the programs HAZE synthesises. A harness was constructed that asserts "for *all possible inputs*, the

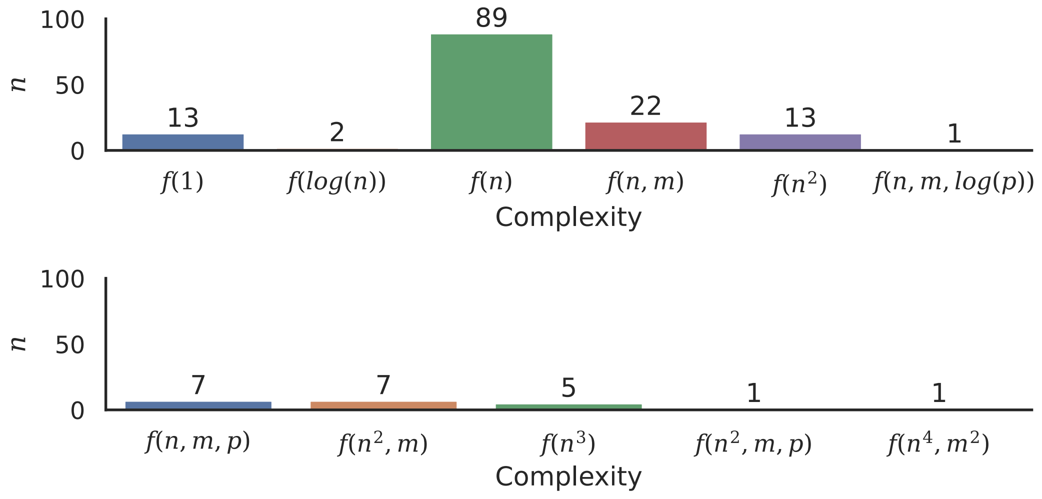


Figure 7.10: Frequencies of observed potential complexity classes across the dataset of evaluation problems (with respect to scalar parameters). Names of parameters have been standardized to $n,m,p,\ldots$. The class $f(n,m)$ includes all classes that are both $O(n)$ and $O(m)$, for fixed m and n respectively (i.e. both $O(nm)$ and $O(n+m)$ are subsumed). Classes are ordered approximately from left to right, and from top to bottom by increasing complexity.

synthesised solution produces the same output as the reference". By using KLEE's symbolic execution capabilities, this assertion can be checked formally. No programs synthesised by Haze that exhibited *significantly* different behaviour to the corresponding reference implementation were identified by doing so.

More precisely, many synthesised programs did in fact differ on minor floating-point arithmetic points when compared to the reference (for example, synthesising the expression $(a*b)*c$ instead of $a*(b*c)$ is technically incorrect in a non-associative arithmetic). Because the input-output example-based correctness checker used by Haze (see Section 3.6) adjusts for such cases using a ULP-based sliding equality check, such differences were common. These examples are summarised as **FP Assoc.** in Table 7.3.

After disregarding minor floating-point differences, no synthesised programs (i.e. those judged to be correct with respect to IO examples) were then judged to be incorrect by the model checker (**Bugs** in Table 7.3). This validates the use of an example-based, observational equivalence approach to determining correctness.

7.5.5 Grey-Box Information

Of the three sources of grey-box information considered by HAZE (performance model, memory trace and instruction distribution), only the analysis of the performance model can lead to the exclusion of correct syntheses; incorrect alignment rankings or instruction sampling distributions will only slow down the discovery of correct solutions. It is therefore important to validate the performance model results extracted by HAZE from each oracle.

To do so, the complexity inferred by HAZE for each individual parameter was compared by hand to the reference code used to implement the oracle. In all but one of the cases where HAZE predicted a non-constant asymptotic complexity for a parameter, its prediction was correct; this accuracy is to some extent an artifact of the type of function targeted by HAZE (i.e. regular, data-centric functions targeted for optimised implementations). The incorrectly predicted case was the `collatz` benchmark from PRESYN's evaluation set, which performs irregular, data-dependent control flow. For the type of synthesis problem targeted by HAZE, per-parameter runtime complexity can be predicted accurately.

7.5.6 Ablation Study

To determine the effect of each source of grey-box information on HAZE's synthesis performance, a partial ablation study was performed by omitting each of the sources of grey-box information in turn.

Four ablated variants of HAZE were constructed as follows:

Perf: Runtime performance models only, no ranking of sketches, instructions are sampled uniformly.

Perf+Trace: Runtime performance and memory traces; sketches are ranked but instructions are sampled uniformly.

Perf+Dist: Runtime performance and instruction distributions; no sketch ranking but instructions are sampled from learned distributions.

All: The full HAZE implementation.

For each variant of HAZE, each problem in the evaluation dataset was attempted, with the mean number of candidates required to do so successfully being recorded. The results achieved by doing so are shown in Figure 7.11.

Adding each source of information produces a clear improvement to the achievable synthesis performance, though they do so through different mechanisms. **Perf+Dist**

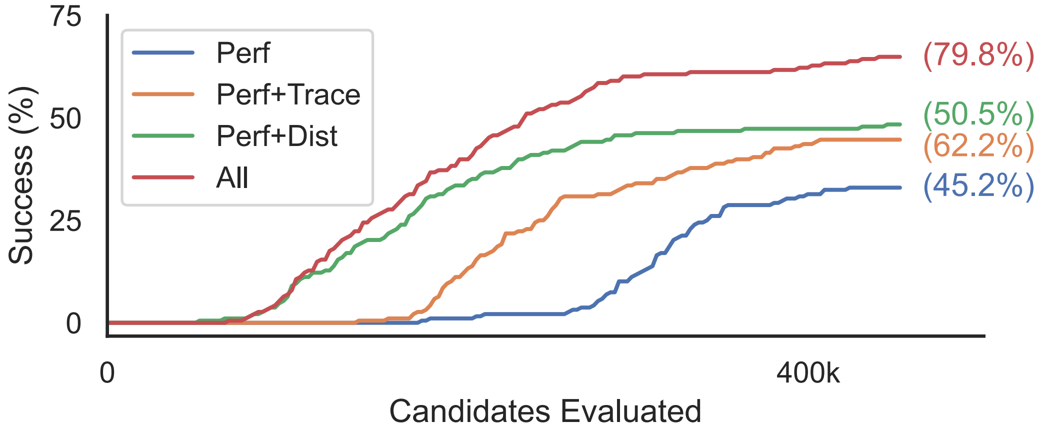


Figure 7.11: Cumulative successful syntheses as a proportion of the entire evaluation dataset for four different versions of Haze (baseline using only performance models to select sketches, adding memory traces and instruction distributions respectively, and the full Haze pipeline). The distribution of successful syntheses is long-tailed, and so for clarity this figure shows only the initial phase where the majority of successes are achieved ($< 500,000$) candidates evaluated). Parenthesised values indicate the final success rate for each version; note that in extremis, the version using memory traces outperforms the one using instruction distributions.

discovers programs with more instructions more quickly than the **Perf** baseline. Similarly, the **Perf+Trace** variant allowed programs with more complex control flow to be synthesised earlier.

Because problems with complex control flow are likely to also require more instructions to be correctly identified, the **Perf+Dist** variant requires more candidates to be evaluated before beginning to return successes. By combining the two approaches in Haze, the initial phase of synthesis is as productive as **Perf+Dist** alone, but is able to scale to complex problems similarly to **Perf+Trace**.

7.6 Related Work

Haze aims to strike a balance between the generality of inductive program synthesis and the power of domain-specific lifting. By using inductive synthesis techniques, Haze is able to make fewer assumptions on the structure of solutions than lifters do. For example, Kamil et al. [97] and Ahmad et al. [98] both require that their target program is an image-processing kernel compatible with a particular domain-specific language, while Rodríguez et al. [169] operate purely within the polyhedral model.

The "grey-box" approach taken by Haze is validated by recent work in synthesis that investigates similar sources of "grey-box" information to Haze. For example, Hu et al. [184] apply *asymptotic resource bounds* to traditional functional specifications. Their use of recursive resource bound typing is analogous (though more expressive) to Haze's construction of solutions based on asymptotic complexity annotations. Similarly, work in multi-modal synthesis demonstrates the utility of considering even more diverse sources of information than Haze does; Chen et al. [92] achieve state-of-the-art performance in regular expression synthesis by combining functional specifications with natural language.

At a high level, the use of additional information beyond a problem's functional specification is likely to become more and more relevant as the desired complexity of program synthesis grows; Haze proposes an initial attempt to do so. Integrating further concrete sources of information (such as dynamic control flow data [185]) is likely to lead to further work in specific application contexts.

7.7 Conclusion

This chapter has presented a new program lifting approach using *grey-box* behaviour, automatically constructing a program to match the behaviour of an unknown component. It generalises across domains, synthesising and lifting more programs than prior techniques, without any external assistance. The grey-box synthesis methodology is validates using bounded model checking, demonstrating that the synthesised programs are correct.

Chapter 8

Conclusions

This thesis was motivated by the problem of rejuvenating legacy software to take better advantage of newly available hardware and software components (e.g. new heterogeneous devices or software libraries). Doing so by hand is possible, but projects are often hamstrung by limited development resources. If an automated solution were possible, then legacy software could be modernised and rejuvenated without requiring up-front developer effort. To enable an automated approach, a formal model of the behaviour of software components would be necessary. The solution proposed in this thesis was to *synthesise* programs matching the behaviour of components, so that compatible regions of application code could be automatically refactored to take advantage of them. Chapter 1 sets out the motivating scenario in more detail.

Then, in Chapter 3, the background material for this thesis was presented. The technical components of the work carried out builds on existing tools: the LLVM compiler intermediate representation [8], and the CAnDL domain-specific language for constraint-based search over LLVM programs [12]. For both of these tools, an overview of their key features and uses was given. Then, an overview of a minimal, free-form language for program *sketching* was presented, and the connections between LLVM, CAnDL and sketching program synthesis were explained. Chapter 2 summarised and reviewed important contributions from the literature relevant to this thesis.

Next, Chapters 4 to 7 described the implementation and evaluation of three distinct approaches to program synthesis. Annote used contextual information known by users of a library to direct the search for potential program sketches, while Presyn trained a probabilistic model as a more general approach to the same problem. Presyn's synthesis is used to drive an automated refactoring tool in Chapter 6. Finally, Haze used non-correctness specification ("grey-box information") to implement an improved synthesis procedure.

Finally, this chapter concludes the work in this thesis, and is structured as follows:

Section 8.1 lists the technical and abstract contributions made in the course of carrying out the research in this thesis. Then, Section 8.2 considers the potential threats to validity for this thesis, and how they could be addressed. Finally, Section 8.3 proposes interesting future directions for research.

8.1 Contributions

This section gives a brief overview of the most important technical and experimental results produced in the course of this thesis. They are as follows:

8.1.1 Approaches to Synthesis

The key contributions of this thesis are the three proposed approaches to synthesis: Annote, Presyn and Haze. While they share a common objective (learning executable models of software components), each examines a different source of information to inform their search for candidate programs. Being able to model and understand what information and assistance is available for a synthesis problem is a key step towards general synthesis for existing components.

Annote examines the use of information that is known only informally about a problem, by encoding that information as part of a type signature. Presyn uses a dataset of known functions or synthesised solutions to train probabilistic models of program structure, so that subsequent programs can be synthesised more easily. Haze examines how non-correctness specifications can reduce the size of a synthesis search space. Together these three approaches comprehensively examine the information that may be available when synthesising a model for an existing software component, and demonstrate its practical application to synthesis.

8.1.2 Synthesis Framework

To evaluate these approaches, a substantial shared implementation framework was created, covering both practical and theoretical concerns. In order to integrate with the CAnDL language's tooling, LLVM IR was chosen as their shared target language. Synthesising LLVM IR required the implementation of several tools and libraries, including a language extension supporting symbolic holes.

More abstractly, a domain-specific language for specifying and manipulating general program sketches was designed. This language allows synthesisers to specify their own arbitrary sketch fragments, while providing a common set of tools for manipulating, compiling and testing sketches composed of these fragments.

8.1.3 Grey-Box Synthesis

Haze (in Chapter 7) proposes the idea of *grey-box synthesis*, where non-correctness specifications derived from observations of a component's behaviour are used to drive synthesis. This technique is a novel one within the areas of oracle-driven and multimodal synthesis: often, oracle-driven techniques are stated abstractly, rather than considering observations of a real device. Similarly, multimodal synthesis is often restricted to limited domains such as natural-language specification. Combining these approaches in practice is a useful contribution.

8.1.4 Evaluation of Synthesisers

The difficulty in comparing inductive synthesisers against one another is well-established in the literature [46, 47]. To address this difficulty, at least partially, this thesis proposed a common set of synthesis problems stated as reference C implementations, together with mechanisms to prepare problem inputs for several common synthesisers from the same reference implementation.

Providing shared inputs for different synthesisers is only one part of establishing a fair comparison between them. As well as those tools, this thesis gives a detailed evaluation of the assistance and inputs given to each synthesiser to evaluate *why* they may or may not succeed on a particular problem. Together, the set of benchmarks and qualitative comparison methodology are a substantial contribution.

8.1.5 Semantic API Migration Tool

Finally, this thesis proposes a set of tools for performing API migration on LLVM IR problems. Regions of code described by a CAnDL search result can be extracted and replaced by a call to a library function. Doing so safely, in a way that considers the local context of the original code is difficult.

8.2 Critical Evaluation

This section considers the most important threats to the validity of the research in this thesis. Their impact on the results as presented and an assessment of how they are accounted for in this thesis are given.

8.2.1 Difficulty of Benchmarking

As stated above, comparing different synthesisers against one another is an extremely challenging problem. Every implementation targets a different language, using a different set of search procedures and atomic components. As a result, providing fair comparisons between their respective performances is difficult.

This problem is addressed in this thesis in two ways. First, through the construction of a common set of reference problems that can be converted automatically into idiomatic specifications for different synthesisers (e.g. by restating array problems as lists for functional implementations). Secondly, by testing synthesiser implementations under multiple scenarios; one where input similar to their original evaluation is supplied, and one more similar to the inputs given to the synthesiser from this thesis. By doing so, the effect of context and implicit assumptions can be measured.

8.2.2 Cost of Synthesis

Because the synthesisers described in this thesis are intended for use in practical workflows (i.e. by compiler developers as a mechanism for creating novel, high-level optimisations, or by application developers in the course of maintaining their software), the time taken to synthesise each new program is an important consideration. However, none of the synthesisers implemented in this thesis make synthesis time an explicit design goal.

Broadly, across all three synthesisers, the time taken to synthesise a given library function is in line with their expected usage (as one-off tools not invoked during every compilation of an application). Some successful syntheses take up to several hours to complete. Many potential techniques exist to accelerate and optimise synthesis processes of this kind. For example, hill-climbing [72] can be used to more effectively direct the instantiation of symbolic holes. Alternatively, as the quality of formal models for LLVM IR continue to improve [186], the application of a CEGIS loop becomes a more viable implementation choice.

CAnDL search and code replacement must be performed during every compilation of an application, however. The impact of these phases on compilation time was found by the original authors to typically be within a factor of 2–3× [12, 112]. Optimising this part of the compilation and migration process is outwith the scope of this thesis.

8.2.3 Generalisation and Bias

For any synthesiser, the choice of program components, search procedure and evaluation dataset represent natural sources of bias. In this thesis, while manually-curated components of each synthesiser's implementation are present, where possible they are selected in such a way as to minimise bias. In cases where bias remains, it is noted explicitly.

The most significant implicit bias present in the three synthesisers implemented is the type of library function targeted (i.e. evaluation cases not derived from existing synthesis benchmarks). In each case, these programs are regular, data-processing functions likely to present a performance bottleneck (where a *regular* function is one that performs a similar operation across its input range, and does not exhibit significant divergence on a sparse subset of that range). Decisions made in the "meta" design level of each synthesiser rely on this assumption (for example, Haze assumes that programs access many memory locations to produce traces).

This bias arises naturally from the problem statement in Chapter 1: to rejuvenate and optimise applications by taking better advantage of newly available libraries and components. Doing so requires that performance-sensitive cases be examined. Additionally, synthesising the entire surface area of a given library is an intractable problem without further constraints. In summary, the implicit assumptions made in the course of this thesis are necessary for problem tractability, and arise naturally from the motivating scenario.

8.2.4 Performance

The analysis of potential performance gains through porting applications to new libraries in Chapter 4 is limited to 3 different platforms and libraries. A more detailed evaluation of which components and optimisation choices are likely to benefit each application (perhaps leading to a predictive performance model) is necessary to conclude in general that performance can be improved without hand-coded optimisation. Ginsbach et al. [23] demonstrate a technique by which memory transfers between the CPU and GPU can be made lazily, reducing the downside risk of porting an application.

8.2.5 Input Generation

Being able to generate valid, "interesting" inputs for an arbitrary black-box oracle (or even a known C program [150]) is an unsolved problem in the general case. This means that inputs to oracle components and synthesised functions must be generated according to predefined heuristics; if a component exhibits behaviour outwith the scope of these heuristics, then that behaviour will never be captured by synthesis.

Using branch coverage, M^3 (Chapter 6) experimentally validates that randomly-generated input examples adequately cover the behaviour of the reference programs in its evaluation dataset. Similarly, the performance models used by Haze are validated to ensure that its input generation strategies are adequate. In the context of the programs synthesised in this thesis, heuristic input generation strategies are valid; however, as synthesis scales to more complex, general components it is likely that improvements to input generation will be necessary.

8.3 Future Work

8.3.1 Neural Synthesis

Increasingly, neural program synthesis is becoming a viable tool for learning complex programs. Combining their ability to derive patterns from large datasets and integrate multiple modalities of information with new developments in representation structure (e.g. graph neural networks) has yielded significant new results in synthesis. Applying neural methods to parts of the workflows suggested in this thesis (e.g. prediction of structure in Presyn, or more complex tasks such as instruction sequence generation) may yield improved results given appropriate training data.

8.3.2 Lifting

The work in this thesis focuses on the refactoring and rejuvenation of *low-level* user code (i.e. the LLVM IR generated from original source code). This suits the synthesis and code search tools used in this thesis, but does not generalise to more abstract representations of code. Future work could examine the combination of lifting with synthesis of library functions: if a program could be lifted to a higher-level representation, and synthesis performed at that level, it may be possible to improve the success of refactoring and migration techniques.

Additionally, combining black- and grey-box synthesis with “white-box” lifting in the spirit of Helium [96] (and other domain-specific lifters) would be an appropriate development of this thesis: by allowing tools to take advantage of all available information, more general results could be obtained.

8.3.3 Higher-Order Learning

New hardware devices are frequently *programmable* to some degree; their behaviours are expressed in terms of state machines, and may be configurable in some way. The synthesis techniques in this thesis scale to straightforward “computational” devices and libraries, but do not currently scale to stateful or programmable interfaces. Generalising existing synthesis and refactoring tools to support programmability would be a difficult, but substantial expansion of the ideas in this thesis.

8.3.4 Tool Support

This thesis proposes a number of tools for the manipulation of LLVM IR, in particular in relation to symbolic holes (for example, the RAUW-NT primitive operation). Integrating these contributions with existing higher-level languages to build solver-aided tools that take advantage of existing compiler pipelines and mechanisms is an interesting future direction.

8.3.5 Input Generation

As described above, the generation of viable input examples for a component oracle or synthesised program is likely to become a bottleneck as their complexity scales in the future. Being able to synthesise interesting input examples without access to component implementation details is therefore a useful future task.

One possible direction for this work is related to the idea of *grey-box* synthesis: instead of making use of a verification tool to construct inputs, observations of grey-box behaviour are used to learn what structure of input examples is likely to produce new behaviour in a component. A co-operative loop between input generation and behavioural observation could therefore be implemented.

8.4 Summary

This chapter summarised the general structure of this thesis, giving a brief summary of each chapter. The key technical and theoretical contributions were summarised, along with a critical evaluation of the primary threats to validity for the research carried out. Finally, an overview of interesting future directions is given.

Bibliography

[1] Bruce Collie and Michael F. P. O'Boyle. Augmenting Type Signatures for Program Synthesis. *arXiv:1907.05649 [cs]*, July 2019.

[2] Bruce Collie, Philip Ginsbach, and Michael F. P. O'Boyle. Type-Directed Program Synthesis and Constraint Generation for Library Portability. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 55–67, September 2019. doi: 10.1109/PACT.2019.00013.

[3] Bruce Collie and Michael F.P. O'Boyle. Retrofitting Symbolic Holes to LLVM IR. *arXiv:2006.05875 [cs]*, June 2020.

[4] Bruce Collie, Philip Ginsbach, Jackson Woodruff, Ajitha Rajan, and Michael F. P. O'Boyle. M3: Semantic API Migrations. In *Proceedings of the Thirty-Fifth International Conference on Automated Software Engineering*, ASE '20, Virtual Event, Australia, 2020. ACM. ISBN 978-1-4503-6768-4. doi: 10.1145/3324884.3416618.

[5] Bruce Collie, Jackson Woodruff, and Michael F. P. O'Boyle. Modeling Black-Box Components with Probabilistic Synthesis. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2020, pages 1–14, New York, NY, USA, November 2020. Association for Computing Machinery. ISBN 978-1-4503-8174-1. doi: 10.1145/3425898.3426952.

[6] Bruce Collie and Michael F. P. O'Boyle. Program Lifting using Gray-Box Behavior. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 60–74, September 2021. doi: 10.1109/PACT52795.2021.00012.

[7] William Allan Wulf, Richard K. Johnsson, Charles B. Weinstock, Steven O. Hobbs, and Charles M. Geschke. *The Design of an Optimizing Compiler*. Elsevier Science Inc. ISBN 978-0-444-00158-0.

[8] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis amp; transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, March 2004. doi: 10.1109/CGO.2004.1281665.

[9] GCC, the GNU Compiler Collection. URL `https://gcc.gnu.org/`.

[10] GHC: The Glasgow Haskell Compiler. URL `https://www.haskell.org/ghc/`.

[11] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: Bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 65–79. Association for Computing Machinery. ISBN 978-1-4503-8391-2. doi: 10.1145/3453483.3454030.

[12] Philip Ginsbach, Lewis Crawford, and Michael F. P. O'Boyle. CAnDL: A Domain Specific Language for Compiler Analysis. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 151–162, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5644-2. doi: 10.1145/3178372.3179515.

[13] Pinaki Chakraborty. Fifty years of peephole optimization. *Current Science*, 108 (12):2186–2190, 2015. ISSN 0011-3891.

[14] T. Glek and J. Hubicka. Optimizing real world applications with GCC Link Time Optimization. *arXiv: 1010.2196 [cs]*, July 2010.

[15] Rubens E.A. Moreira, Sylvain Collange, and Fernando Magno Quintão Pereira. Function Call Re-Vectorization. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 313–326. ACM. ISBN 978-1-4503-4493-7. doi: 10.1145/3018743.3018751.

[16] Leslie Lamport. The parallel execution of do loops. *Commun. ACM*, 17(2):83–93, feb 1974. ISSN 0001-0782. doi: 10.1145/360827.360844.

[17] L. Crawford and M. O'Boyle. Specialization Opportunities in Graphical Workloads. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 272–283. doi: 10.1109/PACT.2019.00029.

[18] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A Compiler Infrastructure for the End of Moore's Law. *arXiv: 2002.11054 [cs]*, February 2020.

[19] Zheng Wang and Michael O'Boyle. Machine Learning in Compiler Optimization. *Proceedings of the IEEE*, 106(11):1879–1901, 2018. ISSN 1558-2256. doi: 10.1109/JPROC.2018.2817118.

[20] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462176.

[21] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Lift: A Functional Data-parallel IR for High-performance GPU Code Generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 74–85, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5090-4931-8. doi: 10.1109/cgo.2017.7863730.

[22] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. High Performance Stencil Code Generation with Lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, pages 100–112. ACM. ISBN 978-1-4503-5617-6. doi: 10.1145/3168824.

[23] Philip Ginsbach, Bruce Collie, and Michael F. P. O'Boyle. Automatically harnessing sparse acceleration. In *Proceedings of the 29th International Conference on Compiler Construction*, CC 2020, pages 179–190, New York, NY, USA, February 2020. Association for Computing Machinery. ISBN 978-1-4503-7120-9. doi: 10.1145/3377555.3377893.

[24] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`.

[25] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga,

Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, Red Hook, NY, USA, 2019. Curran Associates Inc. doi: 10.5555/3454287.3455008.

[26] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. Taco: A tool to generate tensor algebra kernels. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 943–948, Urbana-Champaign, IL, USA, October 2017. IEEE Press. ISBN 978-1-5386-2684-9. doi: 10.1109/ASE.2017.8115709.

[27] Daniele G. Spampinato and Markus Püschel. A Basic Linear Algebra Compiler. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 23–32. ACM. ISBN 978-1-4503-2670-4. doi: 10.1145/2544137.2544155.

[28] Intel® Math Kernel Library (MKL). `https://software.intel.com/mkl`.

[29] CUDA Toolkit. URL `https://developer.nvidia.com/cuda-toolkit`.

[30] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science Engineering*, 12(3): 66–73, 2010. ISSN 1521-9615. doi: 10.1109/MCSE.2010.69.

[31] W. Xu, Z. Liu, J. Wu, X. Ye, S. Jiao, D. Wang, F. Song, and D. Fan. Auto-Tuning GEMV on Many-Core GPU. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, pages 30–36, December 2012. doi: 10.1109/ICPADS.2012.15.

[32] Jong-Ho Byun, Richard Lin, Katherine A. Yelick, and James Demmel. Autotuning Sparse Matrix-Vector Multiplication for Multicore. Technical Report UCB/EECS-2012-215, EECS Department, University of California, Berkeley. URL `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-215.html`.

[33] Urban Borštnik, Joost VandeVondele, Valéry Weber, and Jürg Hutter. Sparse matrix multiplication: The distributed block-compressed sparse row library. *Parallel Computing*, 40(5):47–58. ISSN 0167-8191. doi: 10.1016/j.parco.2014.03.012.

[34] Athena Elafrou, Vasileios Karakasis, Theodoros Gkountouvas, Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. SparseX: A Library for High-Performance Sparse Matrix-Vector Multiplication on Multicore Platforms. *ACM Trans. Math. Softw.*, 44(3):26:1–26:32. ISSN 0098-3500. doi: 10.1145/3134442.

[35] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. PolyMage: Automatic Optimization for Image Processing Pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 429–443. ACM. ISBN 978-1-4503-2835-7. doi: 10.1145/2694344.2694364.

[36] Gordon E. Moore. Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35. ISSN 1098-4232. doi: 10.1109/N-SSC.2006.4785860.

[37] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268. ISSN 1558-173X. doi: 10.1109/JSSC.1974.1050511.

[38] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4892-8. doi: 10.1145/3079856.3080246.

[39] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 27–40. ACM. ISBN 978-1-4503-4892-8. doi: 10.1145/3079856.3080254.

[40] Ruizhe Cai, Ao Ren, Ning Liu, Caiwen Ding, Luhao Wang, Xuehai Qian, Massoud Pedram, and Yanzhi Wang. VIBNN: Hardware Acceleration of Bayesian Neural Networks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 476–488. ACM. ISBN 978-1-4503-4911-6. doi: 10.1145/3173162.3173212.

[41] Jaeha Kung, Yun Long, Duckhwan Kim, and Saibal Mukhopadhyay. A Programmable Hardware Accelerator for Simulating Dynamical Systems. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 403–415. ACM. ISBN 978-1-4503-4892-8. doi: 10.1145/3079856.3080252.

[42] V. Gogte, A. Kolli, M. J. Cafarella, L. D'Antoni, and T. F. Wenisch. HARE: Hardware accelerator for regular expressions. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, MICRO '16, pages 1–12. doi: 10.1109/MICRO.2016.7783747.

[43] J. Koenig, D. Biancolin, J. Bachrach, and K. Asanovic. A Hardware Accelerator for Computing an Exact Dot Product. In *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*, pages 114–121. doi: 10.1109/ARITH.2017.38.

[44] Cristina David and Daniel Kroening. Program synthesis: Challenges and opportunities. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 375(2104):20150403, October 2017. doi: 10.1098/rsta.2015.0403.

[45] Sumit Gulwani. Dimensions in Program Synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP '10, pages 13–24, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0132-9. doi: 10.1145/1836089.1836091.

[46] Edward Pantridge, Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. On the Difficulty of Benchmarking Inductive Program Synthesis Methods. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*,

GECCO '17, pages 1589–1596, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4939-0. doi: 10.1145/3067695.3082533.

[47] Thomas Helmuth and Lee Spector. General Program Synthesis Benchmark Suite. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 1039–1046, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3472-3. doi: 10.1145/2739480.2754769.

[48] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 404–415, San Jose, California, USA, October 2006. Association for Computing Machinery. ISBN 978-1-59593-451-2. doi: 10.1145/1168857.1168907.

[49] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. Sketching concurrent data structures. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 136–148. Association for Computing Machinery. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375599.

[50] Susmit Jha and Sanjit A Seshia. A theory of formal synthesis via inductive learning. *Acta Informatica*, 54(7):693–726, 2017. doi: 10.1007/s00236-017-0294-5.

[51] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Counterexample Guided Inductive Synthesis Modulo Theories. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 270–288. Springer International Publishing. ISBN 978-3-319-96145-3. doi: 10.1007/978-3-319-96145-3_15.

[52] 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*, Onward! 2013, pages 135–152, Indianapolis, Indiana, USA, 2013. ACM. ISBN 978-1-4503-2472-4. doi: 10.1145/2509578.2509586.

[53] James Bornholt and Emina Torlak. Synthesizing Memory Models from Framework Sketches and Litmus Tests. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 467–481. ACM. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062353.

[54] Eric Butler, Emina Torlak, and Zoran Popović. Synthesizing Interpretable Strategies for Solving Puzzle Games. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, FDG '17, pages 10:1–10:10, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5319-9. doi: 10.1145/3102071.3102084.

[55] Armando Solar-Lezama. Introduction to Program Synthesis, 2018. URL `https://people.csail.mit.edu/asolar/SynthesisCourse`. Lecture notes, accessed 18/01/2022.

[56] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing Program Input Grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 95–110. ACM, . ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062349.

[57] Rohan Bavishi, Hiroaki Yoshida, and Mukul R. Prasad. Phoenix: Automated data-driven synthesis of repairs for static analysis violations. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 613–624. Association for Computing Machinery. ISBN 978-1-4503-5572-8. doi: 10.1145/3338906.3338952.

[58] Azadeh Farzan and Victor Nicolet. Synthesis of Divide and Conquer Parallelism for Loops. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 540–555, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062355.

[59] Grigory Fedyukovich, Maaz Bin Safeer Ahmad, and Rastislav Bodik. Gradual Synthesis for Static Parallelization of Single-pass Array-processing Programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 572–585, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062382.

[60] Armando Solar-Lezama. The Sketching Approach to Program Synthesis. In *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 4–13. Springer, Berlin, Heidelberg, December 2009. doi: 10.1007/978-3-642-10672-9_3.

[61] Armando Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer*, 15(5):475–495, October 2013. ISSN 1433-2787. doi: 10.1007/s10009-012-0249-7.

[62] 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, pages 452–466, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062365.

[63] Osbert Bastani, Xin Zhang, and Armando Solar-Lezama. Synthesizing Queries via Interactive Sketching, . URL `http://arxiv.org/abs/1912.12659`.

[64] Rohit Singh, Rishabh Singh, Zhilei Xu, Rebecca Krosnick, and Armando Solar-Lezama. Modular Synthesis of Sketches Using Models. In Kenneth L. McMillan and Xavier Rival, editors, *Verification, Model Checking, and Abstract Interpretation*, Lecture Notes in Computer Science, pages 395–414. Springer Berlin Heidelberg, 2014. ISBN 978-3-642-54013-4.

[65] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. Optimizing Synthesis with Metasketches. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 775–788, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837666.

[66] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8. doi: 10.1109/FMCAD.2013.6679385.

[67] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 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, pages 436–449, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5698-5. doi: 10.1145/3192366.3192410.

[68] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of Loop-free Programs. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 62–73, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993506.

[69] Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105. ISSN 0001-0782. doi: 10.1145/2240236.2240260.

[70] Sunbeom So and Hakjoo Oh. Synthesizing Imperative Programs from Examples Guided by Static Analysis. In Francesco Ranzato, editor, *Static Analysis*, Lecture

Notes in Computer Science, pages 364–381, Freiburg, Germany, 2017. Springer International Publishing. ISBN 978-3-319-66706-5.

[71] Kensen Shi, Jacob Steinhardt, and Percy Liang. FrAngel: Component-based synthesis with control structures. *Proceedings of the ACM on Programming Languages*, 3:73:1–73:29. doi: 10.1145/3290386.

[72] Christopher D. Rosin. Stepping stones to inductive synthesis of low-level looping programs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 2362–2370. AAAI Press, 2019. doi: 10.1609/aaai.v33i01.33012362.

[73] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. Souper: A Synthesizing Superoptimizer. *arXiv:1711.04422 [cs]*, November 2017.

[74] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. Scaling Up Superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 297–310, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4091-5. doi: 10.1145/2872362.2872387.

[75] Kensen Shi, David Bieber, and Rishabh Singh. TF-Coder: Program Synthesis for Tensor Manipulations. *arXiv: 2003.09040 [cs]*, July 2020.

[76] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. *ACM SIGPLAN Notices*, 52(6):422–436. ISSN 0362-1340. doi: 10.1145/3140587.3062351.

[77] 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*, PLDI '15, pages 229–239, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737977.

[78] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed Synthesis: A Type-theoretic Interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 802–815. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837629.

[79] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv: 1611.01989 [cs]*, 2016.

[80] Shengwei An, Rishabh Singh, Sasa Misailovic, and Roopsha Samanta. Augmented example-based synthesis using relational perturbation properties. *Proc. ACM Program. Lang.*, 4(POPL), Dec. 2019. doi: 10.1145/3371124.

[81] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven Synthesis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 408–418, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594297.

[82] Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL `https://openreview.net/forum?id=H1gfOiAqYm`. (unpublished submission; accepted as poster).

[83] Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging Grammar and Reinforcement Learning for Neural Program Synthesis. *arXiv: 1805.04276 [cs]*.

[84] Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. Learning to infer program sketches. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 4861–4870. PMLR, 09–15 Jun 2019. URL `https://proceedings.mlr.press/v97/nye19a.html`.

[85] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. DreamCoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 835–850. Association for Computing Machinery. ISBN 978-1-4503-8391-2. doi: 10.1145/3453483.3454080.

[86] Richard Shin, Neel Kant, Kavi Gupta, Christopher Bender, Brandon Trabucco, Rishabh Singh, and Dawn Song. Synthetic Datasets for Neural Program Synthesis. *arXiv:1912.12345 [cs, stat]*, December 2019.

[87] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples. *arXiv:1804.01186 [cs]*, September 2018.

[88] Tadas Baltrusaitis, Chaitanya Ahuja, and Louis-Philippe Morency. Multimodal machine learning: A survey and taxonomy. *IEEE Trans. Pattern Anal. Mach. Intell.*, 41(2):423–443, feb 2019. ISSN 0162-8828. doi: 10.1109/TPAMI.2018.2798607.

[89] Mehdi Manshadi, Daniel Gildea, and James Allen. Integrating programming by example and natural language programming. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, AAAI'13, pages 661–667. AAAI Press. URL `https://ojs.aaai.org/index.php/AAAI/article/view/8695`.

[90] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. Compositional program synthesis from natural language and examples. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 792–800. AAAI Press, 2015. URL `http://ijcai.org/Abstract/15/117`.

[91] Yanju Chen, Ruben Martins, and Yu Feng. Maximal Multi-layer Specification Synthesis. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 602–612, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-5572-8. doi: 10.1145/3338906.3338951.

[92] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. Multi-modal synthesis of regular expressions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 487–502. Association for Computing Machinery, . ISBN 978-1-4503-7613-6. doi: 10.1145/3385412.3385988.

[93] Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. Optimal Neural Program Synthesis from Multimodal Specifications. *arXiv: 2010.01678 [cs]*, October 2020.

[94] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. Stratified Synthesis: Automatically Learning the x86-64 Instruction Set. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 237–250, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4261-2. doi: 10.1145/2908080.2908121.

[95] Shantanu Thakoor, Simoni Shah, Ganesh Ramakrishnan, and Amitabha Sanyal. Synthesis of programs from multimodal datasets. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), Apr. 2018. URL `https://ojs.aaai.org/index.php/AAAI/article/view/11303`.

[96] Charith Mendis, Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao, and Saman Amarasinghe. Helium: Lifting High-performance Stencil Kernels from Stripped x86 Binaries to Halide DSL Code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 391–402, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737974.

[97] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified Lifting of Stencil Computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 711–726, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4261-2. doi: 10.1145/2908080.2908117.

[98] Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. Automatically Translating Image Processing Libraries to Halide. *ACM Trans. Graph.*, 38(6):204:1–204:13, November 2019. ISSN 0730-0301. doi: 10.1145/3355089.3356549.

[99] Niranjan Hasabnis and R. Sekar. Lifting Assembly to Intermediate Representation: A Novel Approach Leveraging Compilers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 311–324. ACM. ISBN 978-1-4503-4091-5. doi: 10.1145/2872362.2872380.

[100] Sandeep Dasgupta, Sushant Dinesh, Deepan Venkatesh, Vikram S. Adve, and Christopher W. Fletcher. Scalable validation of binary lifters. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 655–671. Association for Computing Machinery. ISBN 978-1-4503-7613-6. doi: 10.1145/3385412.3385964.

[101] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API Property Inference Techniques. *IEEE Transactions on Software Engineering*, 39(5):613–637, May 2013. ISSN 0098-5589. doi: 10.1109/TSE.2012.63.

[102] Z. Xing and E. Stroulia. API-Evolution Support with Diff-CatchUp. *IEEE Trans-*

actions on Software Engineering, 33(12):818–836, December 2007. ISSN 0098-5589. doi: 10.1109/TSE.2007.70747.

[103] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. Mining API Mapping for Language Migration. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 195–204. ACM. ISBN 978-1-60558-719-6. doi: 10.1145/1806799.1806831.

[104] H. D. Phan, A. T. Nguyen, T. D. Nguyen, and T. N. Nguyen. Statistical Migration of API Usages. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 47–50, May 2017. doi: 10.1109/ICSE-C.2017.17.

[105] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N Nguyen. Exploring API embedding for API usages and applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 438–449. IEEE, 2017. doi: 10.1109/ICSE.2017.47.

[106] C. Teyton, J. Falleri, and X. Blanc. Automatic discovery of function mappings between similar libraries. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 192–201, October 2013. doi: 10.1109/WCRE.2013.6671294.

[107] Chunyang Chen, Zhenchang Xing, Yang Liu, and Kent Long Xiong Ong. Mining Likely Analogical APIs across Third-Party Libraries via Large-Scale Unsupervised API Semantics Embedding. *IEEE Transactions on Software Engineering*, pages 432–447, . ISSN 1939-3520. doi: 10.1109/TSE.2019.2896123.

[108] Hussein Alrubaye and Mohamed Wiem Mkaouer. Automating the Detection of Third-party Java Library Migration at the Function Level. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, CASCON '18, pages 60–71, Riverton, NJ, USA, 2018. IBM Corp. doi: 10.5555/3291291.3291299.

[109] ThanhVu Nguyen, Westley Weimer, Deepak Kapur, and Stephanie Forrest. Connecting program synthesis and reachability: Automatic program repair using test-input generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 301–318. Springer, 2017. doi: 10.1007/978-3-662-54577-5_17.

[110] A. Shaw, D. Doggett, and M. Hafiz. Automatically Fixing C Buffer Overflows Using Program Transformations. In *2014 44th Annual IEEE/IFIP International*

Conference on Dependable Systems and Networks, pages 124–135, June 2014. doi: 10.1109/DSN.2014.25.

[111] Kevin Angstadt, Jean-Baptiste Jeannin, and Westley Weimer. Accelerating Legacy String Kernels via Bounded Automata Learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 235–249, Lausanne, Switzerland, March 2020. Association for Computing Machinery. ISBN 978-1-4503-7102-5. doi: 10.1145/3373376.3378503.

[112] Philip Ginsbach, Toomas Remmelg, Michel Steuwer, Bruno Bodin, Christophe Dubach, and Michael F. P. O'Boyle. Automatic Matching of Legacy Code to Heterogeneous APIs: An Idiomatic Approach. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 139–153, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-4911-6. doi: 10.1145/3173162.3173182.

[113] Jonathan Anderson, Robert N. M. Watson, David Chisnall, Khilan Gudka, Ilias Marinos, and Brooks Davis. TESLA: Temporally Enhanced System Logic Assertions. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 19:1–19:14. ACM, 2014. ISBN 978-1-4503-2704-6. doi: 10.1145/2592798.2592801.

[114] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 249–265, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4493-7. doi: 10.1145/3018743.3018758.

[115] LLVM Language Reference Manual — LLVM 13 documentation. https://llvm.org/docs/LangRef.html#abstract, 2021.

[116] Alonzo Church. Application of recursive arithmetic to the problem of circuit synthesis. *Journal of Symbolic Logic*, 28(4):289–290, 1963. doi: 10.2307/2271310.

[117] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. Scalable verification of border gateway protocol configurations with an SMT solver. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 765–780. Association for Computing Machinery. ISBN 978-1-4503-4444-9. doi: 10.1145/2983990.2984012.

[118] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided Component-based Program Synthesis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 215–224, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: 10.1145/1806799.1806833.

[119] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. Live functional programming with typed holes. *Proceedings of the ACM on Programming Languages*, 3(POPL):14:1–14:32, January 2019. doi: 10.1145/3290327.

[120] Ada Sedova, Arnold Tharrington, and Bronson Messer. Portability in Scientific Computing: The Molecular Dynamics Non-bonded Forces Calculation as a Case Study. Technical report, Oak Ridge National Laboratory, August 2018. URL `https://performanceportability.org/case_studies/md/md.pdf`.

[121] Dirk Pflüger, Miriam Mehl, Julian Valentin, Florian Lindner, David Pfander, Stefan Wagner, Daniel Graziotin, and Yang Wang. The scalability-efficiency/maintainability-portability trade-off in simulation software engineering: Examples and a preliminary systematic literature review. In *Proceedings of the Fourth International Workshop on Software Engineering for HPC in Computational Science and Engineering*, SE-HPCCSE '16, pages 19–27, Piscataway, NJ, USA, 2016. IEEE Press. ISBN 978-1-5090-5224-0. doi: 10.1109/SE-HPCCSE.2016.8.

[122] PyTorch. `https://pytorch.org/`.

[123] Joseph Redmon. Darknet: Open source neural networks in C. `http://pjreddie.com/darknet/`, 2013–2016.

[124] NVIDIA cuBLAS. `https://developer.nvidia.com/cublas`.

[125] clMathLibraries clBLAS. `https://github.com/clMathLibraries/clBLAS`.

[126] NVIDIA cuDNN. `https://developer.nvidia.com/cudnn`, 2018.

[127] Intel® Compute Library for Deep Neural Networks (clDNN). `https://01.org/cldnn`.

[128] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T. L. Windus, and W. A. de Jong. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477–1489, September 2010. ISSN 0010-4655. doi: 10.1016/j.cpc.2010.04.018.

[129] AlDanial/cloc. `https://github.com/AlDanial/cloc`.

[130] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Funct ional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM. ISBN 978-1-58113-202-1. doi: 10.1145/351240.351266.

[131] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association. doi: 10.5555/1855741.1855756.

[132] Daniel Liew, Daniel Schemmel, Cristian Cadar, Alastair F. Donaldson, Rafael Zähl, and Klaus Wehrle. Floating-point symbolic execution: A case study in n-version programming. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 601–612, Urbana-Champaign, IL, USA, October 2017. IEEE Press. ISBN 978-1-5386-2684-9. doi: 10.1109/ASE.2017.8115670.

[133] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision – ECCV 2016*, Lecture Notes in Computer Science, pages 525–542. Springer International Publishing, 2016. ISBN 978-3-319-46493-0. doi: 10.1007/978-3-319-46493-0_32.

[134] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You Only Look Once: Unified, Real-Time Object Detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788, June 2016. doi: 10.1109/CVPR.2016.91.

[135] J. Redmon and A. Farhadi. YOLO9000: Better, Faster, Stronger. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6517–6525, July 2017. doi: 10.1109/CVPR.2017.690.

[136] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, March 2012.

[137] Cedric Nugteren. CLBlast: A Tuned OpenCL BLAS Library. In *Proceedings of the International Workshop on OpenCL*, IWOCL '18, pages 5:1–5:10, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-6439-3. doi: 10.1145/3204919.3204924.

[138] J. Turner, J. Cano, V. Radu, E. J. Crowley, M. O'Boyle, and A. Storkey. Characterising across-stack optimisations for deep convolutional neural networks. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 101–110, Sep. 2018. doi: 10.1109/IISWC.2018.8573503.

[139] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic Superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 305–316, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1870-9. doi: 10.1145/2451116.2451150.

[140] José P. Cambronero, Thurston H. Y. Dang, Nikos Vasilakis, Jiasi Shen, Jerry Wu, and Martin C. Rinard. Active Learning for Software Engineering. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, pages 62–78, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6995-4. doi: 10.1145/3359591.3359732.

[141] Sumit Gulwani. Programming by examples - and its applications in data wrangling. In Javier Esparza, Orna Grumberg, and Salomon Sickert, editors, *Dependable Software Systems Engineering*, volume 45 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 137–158. IOS Press, 2016. doi: 10.3233/978-1-61499-627-9-137.

[142] 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*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926423.

[143] Alan Leung, John Sarracino, and Sorin Lerner. Interactive Parser Synthesis by Example. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 565–574, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2738002.

[144] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. Cosette: An automated prover for SQL. In *CIDR 2017, 8th Biennial Conference on Innovative*

Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings, 2017. URL `http://cidrdb.org/cidr2017/papers/p51-chu-cidr17.pdf`.

[145] Null-terminated byte strings. `https://en.cppreference.com/w/c/string/byte`.

[146] Mathfu. `https://github.com/google/mathfu`.

[147] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28:135–151, 2001. doi: 10.1145/567806.567807.

[148] C674x DSPLIB - Texas Instruments Wiki. `http://processors.wiki.ti.com/index.php/C674x_DSPLIB`.

[149] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. Synthesizing Benchmarks for Predictive Modeling. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 86–99, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5090-4931-8. doi: 10.1109/CGO.2017.7863731.

[150] A. F. da Silva, B. C. Kind, J. W. de Souza Magalhães, J. N. Rocha, B. C. Ferreira Guimarães, and F. M. Quinão Pereira. ANGHABENCH: A Suite with One Million Compilable C Benchmarks for Code-Size Reduction. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 378–390, February 2021. doi: 10.1109/CGO51591.2021.9370322.

[151] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-Symbolic Program Synthesis. *arXiv:1611.01855 [cs]*, November 2016.

[152] Paul Jaccard. The Distribution of the Flora in the Alpine Zone. *New Phytologist*, 11(2):37–50, 1912. ISSN 1469-8137. doi: 10.1111/j.1469-8137.1912.tb05611.x.

[153] Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Joshua B. Tenenbaum. Library Learning for Neurally-guided Bayesian Program Induction. In *Proceedings of the 32Nd International Conference on Neural Information Processing Systems*, NIPS'18, pages 7816–7826, USA, 2018. Curran Associates Inc.

[154] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to Infer Graphics Programs from Hand-Drawn Images. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 6059–6068. Curran Associates, Inc., 2018.

[155] Louis Wasserman. Scalable, example-based refactorings with refaster. In *Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools*, WRT '13, page 25–28, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450326049. doi: 10.1145/2541348.2541355.

[156] Valeriy Savchenko, Konstantin Sorokin, Georgiy Pankratenko, Sergey Markov, Alexander Spiridonov, Ilia Alexandrov, Alexander Volkov, and Kwangwon Sun. Nobrainer: An Example-Driven Framework for C/C++ Code Transformations. In *Perspectives of System Informatics*, Lecture Notes in Computer Science, pages 140–155, Cham, 2019. Springer International Publishing. ISBN 978-3-030-37487-7. doi: 10.1007/978-3-030-37487-7_12.

[157] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea. Testing closed-source binary device drivers with ddt. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, page 12, USA, 2010. USENIX Association. doi: 10.5555/1855840.1855852.

[158] Wei Tang, Ping Luo, Jialiang Fu, and Dan Zhang. Libdx: A cross-platform and accurate system to detect third-party libraries in binary code. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 104–115. IEEE, 2020. doi: 10.1109/SANER48275.2020.9054845.

[159] André Miranda and João Pimentel. On the use of package managers by the C++ open-source community. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, pages 1483–1491, Pau, France, April 2018. Association for Computing Machinery. ISBN 978-1-4503-5191-1. doi: 10.1145/3167132.3167290.

[160] Fabien Coelho and François Irigoin. Api compilation for image hardware accelerators. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4): 1–25, 2013.

[161] CWE-126: Buffer over-read, Feb 2020. URL `https://cwe.mitre.org/data/definitions/126.html`.

[162] About Strsafe.h - Windows applications. URL `https://docs.microsoft.com/en-us/windows/win32/menurc/strsafe-ovw`.

[163] Nadia Polikarpova and Ilya Sergey. Structuring the synthesis of heap-manipulating programs. *Proceedings of the ACM on Programming Languages*, 3 (POPL):72:1–72:30, January 2019. doi: 10.1145/3290385.

[164] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1027–1040, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367127. doi: 10.1145/3314221.3314596.

[165] Discovering likely mappings between APIs using text mining. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 231–240, 2015. doi: 10.1109/SCAM.2015.7335419.

[166] Nghi Bui. Towards zero knowledge learning for cross language api mappings. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 123–125, 2019. doi: 10.1109/ICSE-Companion.2019.00054.

[167] Shengzhe Xu, Ziqi Dong, and Na Meng. Meditor: Inference and application of api migration edits. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 335–346, 2019. doi: 10.1109/ICPC.2019.00052.

[168] Zijiang Yang, Jinru Hua, Kaiyuan Wang, and Sarfraz Khurshid. Edsynth: Synthesizing api sequences with conditionals and loops. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 161–171, 2018. doi: 10.1109/ICST.2018.00025.

[169] Gabriel Rodríguez, José M. Andión, Mahmut T. Kandemir, and Juan Touriño. Trace-based affine reconstruction of codes. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, page 139–149, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450337786. doi: 10.1145/2854038.2854056.

[170] Saed Alrabaee, Mourad Debbabi, Paria Shirani, Lingyu Wang, Amr Youssef, Ashkan Rahimian, Lina Nouh, Djedjiga Mouheb, He Huang, and Aiman Hanna. Binary analysis overview. In *Binary Code Fingerprinting for Cybersecurity*, pages 7–44. Springer, 2020. doi: 10.1007/978-3-030-34238-8_2.

[171] Lorenzo Chelini, Andi Drebes, Oleksandr Zinenko, Albert Cohen, Nicolas Vasilache, Tobias Grosser, and Henk Corporaal. Progressive raising in multi-level ir. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 15–26, 2021. doi: 10.1109/CGO51591.2021.9370332.

[172] 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*, PLDI '15, pages 619–630, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2738007.

[173] Peter-Michael Osera. Constraint-based Type-directed Program Synthesis. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development*, TyDe 2019, pages 64–76, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6815-5. doi: 10.1145/3331554.3342608.

[174] Alexandru Calotoiu, Torsten Hoefler, Marius Poke, and Felix Wolf. Using automated performance modeling to find scalability bugs in complex codes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 1–12, New York, NY, USA, November 2013. Association for Computing Machinery. ISBN 978-1-4503-2378-9. doi: 10.1145/2503210.2503277.

[175] Alexandru Calotoiu, David Beckinsale, Christopher W. Earl, Torsten Hoefler, Ian Karlin, Martin Schulz, and Felix Wolf. Fast Multi-parameter Performance Modeling. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 172–181, September 2016. doi: 10.1109/CLUSTER.2016.57.

[176] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200. Association for Computing Machinery. ISBN 978-1-59593-056-9. doi: 10.1145/1065010.1065034.

[177] Gabriel Rodríguez, Mahmut T. Kandemir, and Juan Touriño. Affine Modeling of Program Traces. *IEEE Transactions on Computers*, 68(2):294–300, February 2019. ISSN 2326-3814. doi: 10.1109/TC.2018.2853747.

[178] Osamu Gotoh. Optimal sequence alignment allowing for long gaps. *Bulletin of Mathematical Biology*, 52(3):359–373, 1990. ISSN 0092-8240. doi: 10.1016/S0092-8240(05)80216-2.

[179] David A Ramos and Dawson R. Engler. Practical, Low-effort Equivalence Verification of Real Code. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 669–685, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-22109-5. doi: 10.5555/2032305.2032360.

[180] R. Saghir. *Application-Specific Instruction-Set Architectures for Embedded DSP Applications*. PhD thesis, 1998. URL `https://hdl.handle.net/1807/14794`.

[181] V. Zivojnovic, J. Martinez, C. Schläger, and Heinrich Meyr. DSPstone: A DSP-Oriented Benchmarking Methodology. In *Proc. of ICSPAT'94 - Dallas*, October 1994. URL `https://www.ice.rwth-aachen.de/publications/publication/Zivojnovic94icspat/`.

[182] Louis-Noël Pouchet et al. Polybench: The polyhedral benchmark suite, 2012. URL `http://www.cs.ucla.edu/pouchet/software/polybench`.

[183] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):77:1–77:29, October 2017. doi: 10.1145/3133901.

[184] Qinheping Hu, John Cyphert, Loris D'Antoni, and Thomas Reps. Synthesis with asymptotic resource bounds. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, pages 783–807, Cham, 2021. Springer International Publishing. ISBN 978-3-030-81685-8. doi: 10.1007/978-3-030-81685-8_37.

[185] Andrei Rimsa, José Nelson Amaral, and Fernando M. Q. Pereira. Practical dynamic reconstruction of control flow graphs. *Software: Practice and Experience*, 51(2):353–384. ISSN 1097-024X. doi: 10.1002/spe.2907. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2907`.

[186] Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. Modular, compositional, and executable formal semantics for LLVM IR. *Proceedings of the ACM on Programming Languages*, 5:67:1–67:30. doi: 10.1145/3473572.