Productivity Tools for Solver-Aided Programming

Sorawee Porncharoenwase

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2023

Reading Commitee:

Emina Torlak, Chair

Xi Wang

Zachary Tatlock

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

Productivity Tools for Solver-Aided Programming

Sorawee Porncharoenwase

Chair of the Supervisory Committee:
Emina Torlak
Paul G. Allen School of Computer Science & Engineering

Solver-aided programming has been employed for many automated reasoning tasks, such as program verification and program synthesis. Despite its success, making solver-aided programs is still difficult, time-consuming, and error-prone. Prior work developed *productivity tools* to help with these issues by relying on language abstraction. Yet, there is still a need for more productivity tools, due to the language's complex semantics, which is foreign to most programmers.

This dissertation aims to show that *recontextualization* and *extensibility* are keys to developing productivity tools for solver-aided programming. Recontextualization makes solver-aided programming more familiar to programmers, and extensibility allows programmers to specify domain-specific knowledge to control the tools. These guiding principles are identified from existing productivity tools and our three new contributions to improve productivity in different stages of solver-aided programming. First, our work on *formal foundation for symbolic evaluation* presents a class of reusable symbolic evaluators that simplify programming to interact with solvers. Second, our work on *profile-guided symbolic optimizer* introduces a tool for scaling up solver-aided programs. Third, our work on *expressive pretty printer* provides a practical approach to interface solver-aided programs with end users. Our contributions along with existing tools support the thesis, giving an insight into the design of future productivity tools for solver-aided programming.

Contents

Acknowledgments

First of all, I would like to thank Emina Torlak, my adivisor, for everything—advices, supports, and opportunities—throughout my graduate study. I am especially grateful for her patience and encouragement during my difficult times, and really appreciate how she accommodates my research interests. I also would like to thank my PhD committee—Xi Wang, Zach Tatlock, and Amy Ko—for their valuable feedbacks and suggestions.

I am very happy to have the opportunity to work with many great collaborators–James Bornholt, Luke Nelson, and Justin Pombrio (and also Emina and Xi). It was really fun to work with you all.

I am indebted to Shriram Krishnamurthi and Tim Nelson who sparked my interest in programming languages and formal methods research during my undergraduate study at Brown University. Similarly, I am grateful to Jittat Fakcharoenphol, Nattee Niparnan, Somchai Prasitjutrakul, Thai Pangsakulyanont, Veerakan Sinthaveelertmongkol, and Witchakorn Kamolpornwijit for getting me interested in computer science when I was in my high school.

Friends from the UNSAT group, the PLSE lab, and Thai community at UW CSE accompanied me during my graduate study, and I am very grateful for their friendships. I would like to especially acknowledge the support that I received from Jacob Van Geffen, Bill Zorn, Rachit Nigam, Nussara Tieanklin, Oliver Flatt, Gus Smith, Chandrakana Nandi, Talia Ringer, Pratyush Patel, Rashmi Mudduluru, and Mangpo Phothilimthana.

Many friends outside UW CSE also helped me a lot. I would like to thank Jack Wrenn, Pakawut Jiradilok, Pakapol Supaniratisai, Ingkarat Rak-amnouykit, Varot Premtoon, Warisara Pattanapongpaiboon, Jidapa Thanabhusest, Sahakait Benyasut, Thananun Prasertsup, Abhabongse Janthong, and Jarunetr Sae-Lim for their support as well.

My hobbies during my graduate study included contributing to the Racket programming language and administering Thai Wikipedia. I would like to express my appreciation to the Racket and Thai Wikipedia communities for providing me with valuable and rewarding experiences.

Lastly, I definitely would not be who I am today without the encouragement and support from my mom, Laksanaporn Thatayatikom, and my dad, Woraseth Porncharoenwase. I am very grateful for everything they have done for me.

Chapter 1

Introduction

Solver-aided programming is a programming model that utilizes constraint solvers during program execution. A solver-aided program interacts with a solver by submitting generated constraints to the solver, and converting solutions from the solver to native values that can be further manipulated. The availability of efficient satisfiability modulo theories (SMT) solvers [31] enables solver-aided programs to perform highly complex automated reasoning tasks in practice, such as program synthesis [83] and program verification [24] over a desired *source language*. The development cycle of a solver-aided program consists of (i) programming to interact with the solver; (ii) debugging errors or faulty results; (iii) scaling the program to handle larger inputs efficiently; and (iv) interfacing with end users. However, each step is difficult, time-consuming, and error-prone.

Thus, there have been research efforts, mostly by relying on language abstraction, to develop tools to improve *productivity* in solver-aided programming. For beginners, the productivity improvement lowers the learning curve of solver-aided programming and makes it more accessible. For experts, the productivity improvement makes solver-aided programming less time-consuming and less error-prone. To simplify programming to interact with the solver, a reusable symbolic

evaluator for a solver-aided host language [93] provides a language abstraction so that programmers do not need to directly create SMT constraints. To scale up a solver-aided program, a symbolic profiler [14] exploits the structure of the solver-aided host language to effectively profile for a performance bottleneck of the solver-aided program. And to interface with end users, a general-purpose pretty printer [96] is often employed to print synthesized code in the given source language [93].

Despite these advances from prior researches, there are still many productivity improvement opportunities. First, the expressiveness of the solver-aided host language or the pretty printing language could be limiting, making it cumbersome to express certain programs. Second, programmers may need to extend the reusable evaluator to support new SMT theories, but the correctness guarantee of existing evaluator is against a fixed set of built-in SMT theories, making it difficult to ensure that the extended evaluator is correct without redoing the entire proof. Last but not least, while a symbolic profiler can point programmers toward a performance bottleneck issue, fixing the performance bottleneck still requires a deep understanding of the solver-aided host language. These opportunities indicate that there is still a need for better productivity tools for solver-aided programming.

My thesis is that *recontextualization* and *extensibility* are keys to developing productivity tools for solver-aided programming. A productivity tool is recontextualizing if it provides an interface of traditional programming under the context of solver-aided programming. Because solver-aided programming is foreign to most programmers due to its complex semantics, recontextualization is able to hide the complexity away, allowing programmers to focus on the task at hand. A productivity tool is extensible if it allows programmers to provide domain-specific knowledge to control the tool so that it works well with the solver-aided program for the source language. Domain-specific knowledge is often required for solver-aided programming to be effective, so extensibility provides opportunities to productivity tools to exploit such knowledge. These keys

to developing productivity tools are identified from both existing tools and my three contributions to develop tools for different stages of the development cycle.

1.1 A formal foundation for symbolic evaluation

Manually generating SMT queries is difficult, error-prone, and time-consuming. Prior work [93] introduces a reusable symbolic evaluator, which allows programmers to embed an interpreter for the source language into the solver-aided host language. When applied to a source program and symbolic inputs, the embedded interpreter is executed by the reusable evaluator with the *all-path evaluation* to produce a logical encoding of the program's semantics. The solver-aided program then uses the resulting encoding to express a verification or synthesis task as a logical satisfiability query, to be submitted to the solver. While the reusable evaluator can boost the productivity significantly, its expressiveness may not suffice for certain solver-aided programs, and its correctness guarantee is only against a fixed set of built-in SMT theories.

Our formal foundation for symbolic evaluation with merging [76] introduces an underlying expressive symbolic semantics for a family of reusable evaluators and solver-aided host languages. The expressiveness of the semantics allows programmers to simplify some solver-aided programs significantly compared to prior work. This family of reusable evaluators is parameterized by the *symbolic factory*, which abstracts away the details of creation, simplification, and merging of symbolic values. To obtain a concrete reusable evaluator, programmers would instantiate the core reusable evaluator with a suitable symbolic factory. This, particularly, allows programmers to support desired SMT theories. Moreover, the foundation provides a *modular* correctness guarantee for the reusable symbolic evaluators with respect to the symbolic factory. Ensuring that a reusable symbolic evaluator is correct only requires proving that the used symbolic factory is correct.

Consequently, our reusable symbolic evaluators are productivity tools that recontextualize

the concrete semantics of traditional programming with symbolic semantics of solver-aided programming, and the symbolic factory in the formal foundation is the extensibility point that allows programmers to provide domain-specific knowledge.

1.2 Scaling up solver-aided programming

Effective symbolic evaluation is key to building scalable solver-aided tools. Using a reusable evaluator effectively requires programmers to be able to *identify* and *repair* performance bottlenecks in code under symbolic semantics (i.e., all-path evaluation), a difficult task even for experts. Prior work introduces a *symbolic profiler*, a productivity tool that recontextualizes the profiler of traditional programming under the context of solver-aided programming. That is, one can use the symbolic profiler to find *where* performance bottlenecks are in symbolic evaluation. While the symbolic profiler can point programmers toward a performance bottleneck issue, often it is still unclear to programmers *how* to fix such performance bottlenecks, which requires a deep understanding of the solver-aided host language.

Our work on *profile-guided symbolic optimizer* formulates symbolic performance repair as a search problem along with a search algorithm for this purpose. The search algorithm is profile-guided: it dynamically runs the program and utilizes the symbolic profiler as a black box to guide the search. Furthermore, the repair algorithm is parameterized by a set of *semantics-preserving rewriting rules*, which allows users to specify the kinds of repairs that are effective for the domain. Consequently, the profile-guided symbolic optimizer is a productivity tool that recontextualizes optimizing compilers in traditional programming under the context of solver-aided programming, whose extensibility in the set of rewriting rules allows the optimizer to operate effectively on the solver-aided program.

1.3 An expressive pretty printer

There are two types of solver-aided programs: those that require an end user to look at the output and those that do not. The reason a user may want to look at the output could range from reviewing synthesized code [61] to interacting with human-in-the-loop program synthesis [64]. Even for solver-aided programs that do not require human users to look at the output, it is still useful to present the output in a human-readable format for debugging purposes. Presenting the output effectively thus can be an essential part of solver-aided programming. Pretty printers have been employed for this purpose, in order to print structured data like an abstract syntax tree (AST) of the source language in a human-readable format. However, existing pretty printers can be limited in their expressiveness, in the optimality objective (which could cause the printed code to be unreadable), or in performance.

Our *expressive pretty printer* presents a new pretty printer, which is well-placed in the trade-off space between expressiveness, optimality objective, and performance. It is strictly more expressive than all published pretty printers. Its optimality objective is controlled by the user-specified *cost factory*, which is uniquely flexible compared to other pretty printers. Its time complexity is linear in the size of the document, and the evaluation result shows that the pretty printer is fast in practice. Thus, our expressive pretty printer is a productivity tool whose extensibility allows programmers to work with any source language effectively.

1.4 Contributions and Outline

The rest of this dissertation is structured as follows. Chapter 2 presents our formal foundation for symbolic evaluation with merging. Chapter 3 presents our profile-guided symbolic optimizer. Chapter 4 presents our expressive pretty printer. These chapters together support the thesis that

the keys to developing effective solver-aided tools are recontextualization and extensibility. We then conclude the dissertation and discuss our preliminary work on improving productivity in debugging solver-aided programs in Chapter 5.

Chapter 2

A formal foundation for symbolic evaluation

2.1 Introduction

Symbolic evaluation is a core component of solver-aided programs/tools, which automate program verification and synthesis tasks by reducing them to satisfiability solving. These tools work by embedding an interpreter for the tool's source language into a host language that is equipped with a *reusable symbolic evaluator* [93, 80, 94]. When applied to a source program and symbolic inputs, the embedded interpreter is executed by the host's reusable evaluator to produce a logical encoding of the program's semantics. The tool then uses the resulting encoding to express a verification or synthesis task as a logical satisfiability query, solved with a SAT or SMT solver. This pipeline produces correct results if the tool generates the right query, and both the evaluator and the solver are correct. In practice, tool developers focus on testing or verifying their use of the symbolic evaluator [99], and trust the evaluator and the solver to be correct. The trust in solvers is based on decades of community investment in their testing [102], validation [30], and verification [10]. But the trust

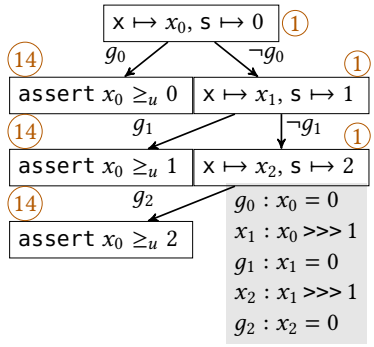
in reusable evaluators rests on a weaker foundation of ad-hoc testing and manual inspection.

This chapter presents the first formal framework for reasoning about the behavior of reusable evaluators. We develop a new symbolic semantics for reusable evaluators, which we call $\mathcal{S}_c$, and we prove that it is sound and complete with respect to the underlying concrete semantics. The framework targets a small but expressive language, λ_c, that extends Core Scheme [37] with assumptions and assertions. As such, λ_c includes the core features supported by existing reusable evaluators: branching, loops, (first-class) procedures, and specification constructs. Our symbolic semantics for λ_c incorporates *state merging* [9, 24, 25], which is key to generating small (polynomially sized) encodings. Unlike the classic merging semantics [9, 24, 25], which was developed for verification of loop-free code, $\mathcal{S}_c$ is designed to be reused by a wide range of tools, on a wide range of programs. It maintains strong invariants on the formulas that characterize the symbolic state, and on termination of halted paths. The former simplifies the formulation of queries, and the latter ensures that symbolic evaluation terminates as often as concrete execution on concrete inputs, which is vital for the development and testing of client tools. We prove these reusability properties and the correctness of $\mathcal{S}_c$ using the Lean theorem prover [32].

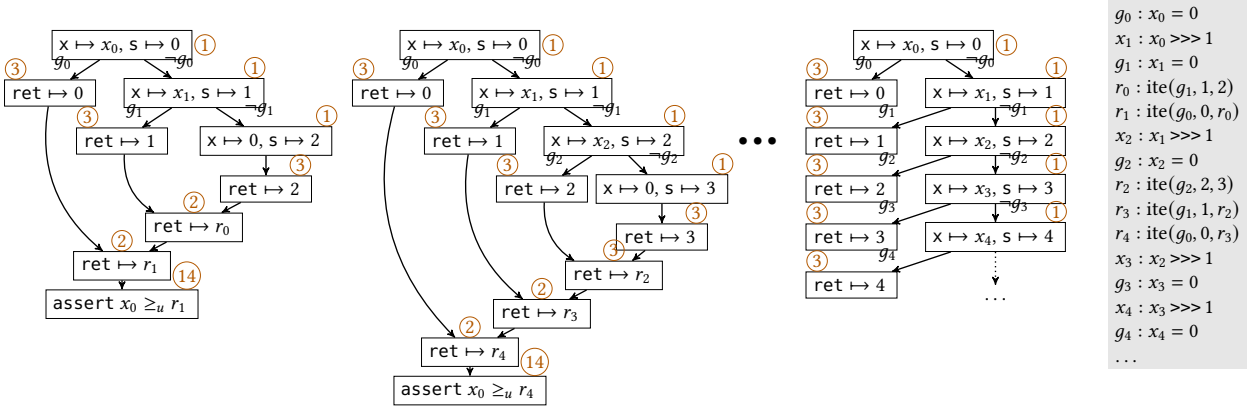
Our framework aims to provide a *general contract* for implementing and validating reusable evaluators. To meet this goal, it must accommodate a wide range of implementations, which is uniquely challenging for reusable evaluators. To see why, consider the toy verification query in Figure 2.1a. The query verifies that the unsigned value of an n-bit integer x exceeds the number of 1's in its binary representation. Figure 2.1b shows the symbolic execution tree [51, 27] for this query when $n = 2$. The tree captures all feasible concrete executions of our program, with the feasibility of each branch decided by an SMT solver. Because of this feasibility check, all implementations of symbolic execution will (in principle) generate an isomorphic tree. But reusable evaluators do not necessarily call the solver during evaluation, so their behavior depends on their strategy for constructing symbolic values, as illustrated in Figure 2.1c. If the evaluator deduces

```
1  (define (addbits x s)
2    (if (bvzero? x)                      ; If x = 0
3        s                                ; then s
4        (addbits                         ; else addbits
5         (bvlshr x (bv 1 n))             ;   (x >>> 1)
6         (bvadd (bvand x (bv 1 n)) s)))) ;   ((x & 1) + s).
7
8  (define (popcount x)      ; Returns the number of 1 bits
9    (addbits x (bv 0 n)))   ; in the n-bit bitvector x.
10
11 (define-symbolic x (bitvector n)) ; Symbolic n-bit bitvector.
12
13 (verify                    ; Verify that for every n-bit
14  (assert                   ; bitvector x, x ≥u (popcount x),
15   (bvuge x (popcount x)))) ; where ≥u is unsigned comparison.
```

(a) A toy program with a verification query.



(b) Symbolic execution tree for $n = 2$.



(c) Symbolic evaluation DAGs for $n = 2$.

Figure 2.1: A toy program with a verification query (a), along with the symbolic execution tree (b) and symbolic evaluation DAGs (c) for this query. Nodes represent symbolic states. Edges represent guarded transitions between states. Dotted edges denote omitted parts of the graph. Circled numbers refer to lines in the toy program.

that right-shifting x two or more times produces 0, it will terminate and produce a finite DAG. Otherwise, it will diverge, producing an infinite DAG. A general semantics for reusable evaluators must account for all of these behaviors.

We address this challenge by defining $\mathcal{S}_c$ with respect to a *symbolic factory*, and proving that it is *partially correct* with respect to the concrete semantics of λ_c. A symbolic factory is a parameter to the semantics, consisting of functions that abstract away the details of creation, simplification, and merging of symbolic values. Our framework specifies only what it means for these functions

to be sound; their implementation is otherwise opaque. For example, all outcomes shown in Figure 2.1c can be produced by a sound factory. To account for non-termination, our correctness criterion specifies what it means for a reusable evaluator to produce a sound and complete result *if* the evaluation terminates. All the DAGs shown in Figure 2.1c are correct according to our definition, with the infinite one satisfying the definition trivially. For programs that are free of loops, our semantics terminates and is totally correct with respect to all sound symbolic factories.

In addition to providing a general contract for reusable evaluators, our framework also aims to expose a *practical interface* to their client tools. We identify two key properties of practical symbolic evaluators, *reducibility* and *legality*, and we design our symbolic semantics so that every (correct) implementation of its rules satisfies these properties.

Reducibility states that symbolic evaluation terminates on a given program and a fully concrete input whenever the concrete semantics does so. This property is trivially satisfied by all symbolic evaluators on loop-free programs, but only symbolic execution is designed to reduce to concrete execution in the presence of loops. Our semantics integrates reducibility in its design as well, by including a mechanism for abandoning halted paths as soon as possible. If an assertion or assumption fails during symbolic evaluation, $\mathcal{S}_c$ requires the evaluator to abandon that path of execution and propagate the failure upstream, similarly to how an exception is propagated in concrete execution. This mechanism forces the evaluator to mirror the concrete semantics on fully concrete inputs, which enables the testing of client tools (e.g., [60, 61]). Beyond testing, this mechanism also enables the use of unwinding assertions [24] to bound loops in the presence of symbolic inputs (see, e.g., Figure 11 of [92]).

Legality simplifies the reuse of the evaluator's output, which takes the form of symbolic states. In all forms of symbolic evaluation, the symbolic state σ is characterized by two formulas, which the client tools use to construct their queries. One encodes the assumptions, $\mathsf{assumes}(\sigma)$, and the other encodes the assertions, $\mathsf{asserts}(\sigma)$, that have been made to reach the state σ. Intuitively, at least one

of these formulas should always be true, because there is no concrete execution in which both an assumption and an assertion fail: the execution stops as soon as the first failure is reached. Yet in the classic merging semantics, both formulas can become false, and client tools must account for this when generating queries to avoid unsound results (see Section 2.4.3). To address this problem, our semantics computes $\mathsf{assumes}(\sigma)$ and $\mathsf{asserts}(\sigma)$ so that every state σ generated during symbolic evaluation is legal—i.e., every model makes at least one of the formulas true. With legality, client tools can adopt a simple interpretation of symbolic states, inherent in symbolic execution, where the validity of the formula $\mathsf{assumes}(\sigma) \rightarrow \mathsf{asserts}(\sigma)$ ensures the absence of errors.

To demonstrate the suitability of our framework for implementing and validating reusable evaluators, we implement $\mathcal{S}_c$ in two different languages. The first implementation is a reference interpreter written in Lean, using a naïve symbolic factory. We use Lean to prove that the reference interpreter implements $\mathcal{S}_c$, and that it satisfies our correctness criterion. The second implementation is an optimized evaluator for Rosette [93], an existing language with a reusable symbolic evaluator that hosts a variety of verification and synthesis tools. We refer to our new evaluator as Rosette 4. Rosette's default evaluator is based on the classic merging semantics, and we refer to it as Rosette 3. Both evaluators use the same optimized symbolic factory. We validate Rosette 4 against the Lean reference interpreter using differential testing, aided by an SMT solver. We find that their behaviors match on 10,000 automatically generated test programs.

To evaluate the practical impact of using $\mathcal{S}_c$, we port 16 published verification and synthesis tools to Rosette 4, and compare the performance of the ported code to the original code built on top of Rosette 3. Our benchmarks include the 15 tools studied in prior work [14] on profiling the performance of reusable evaluators, and a recent system, Jitterbug [61], for verifying and synthesizing just-in-time compilers in the Linux kernel. Our results show that Rosette 4 is up to 14× faster than Rosette 3 across all benchmarks. The largest slowdown we observe is 6×. On average, Rosette 4 is 10–20% faster than Rosette 3. Our evaluation also shows that the

legality-preserving interface exposed by ROSETTE 4 simplifies the implementation of our most complex benchmark, JITTERBUG. The original implementation of JITTERBUG relied on custom code for tracking assumptions, which required careful manual reasoning from the developers to ensure its correctness. The ported implementation removes this custom code, leading to code that is simpler and easier to understand.

In summary, this chapter makes the following contributions:

- The first formal framework for reasoning about symbolic evaluation with merging. We develop a new parametric merging semantics for an expressive core language; we prove that our semantics is sound and complete; and we prove that it preserves legality and reducibility.
- A mechanization of this framework in the Lean theorem prover.
- Two implementations of our semantics, one in Lean and one in Rosette. We prove the Lean implementation correct against our semantics, and we use solver-aided differential testing to show the Rosette implementation matches the Lean implementation.
- An evaluation of the Rosette implementation on 16 tools for program verification and synthesis developed in prior work. Our evaluation shows that this new implementation offers better performance and a cleaner interface than Rosette's default symbolic evaluator.

The rest of this chapter is organized as follows. Section 2.2 discusses related work. Section 2.3 introduces our target language and illustrates basic applications of a reusable evaluator for this language. Section 2.4 presents the $\mathcal{S}_c$ merging semantics and compares it to the classic one. Section 2.5 states our correctness criterion and shows that $\mathcal{S}_c$ satisfies it. Section 2.6 describes our Lean and Rosette implementations of $\mathcal{S}_c$, proofs, and differential testing results. Section 2.7 evaluates the utility and performance of the Rosette implementation. Section 2.8 concludes the chapter.

2.2 Related Work

Reusable symbolic evaluators [93, 80, 94] are designed to serve as platforms for building new tools. They reduce programs to constraints via a combination of symbolic execution [51, 27] and bounded model checking [9]. Symbolic execution encodes each path through a program separately, giving rise to a potentially infinite symbolic execution tree. Each node in this tree is a symbolic state, which represents a set of concrete program states, and each path represents a set of feasible concrete paths. Symbolic execution is well-understood and has been formalized for imperative languages by recognizing a correspondence between concrete and symbolic paths in the absence of state merging [56]. Prior work [65] has also formalized symbolic execution for a language that extends lambda calculus with numbers and contract constructs, similarly to our work.

Bounded model checking uses *state merging* to optimize symbolic execution of loop-free programs. It merges symbolic states from different paths at each control-flow join, giving rise to a DAG that is asymptotically smaller than the corresponding symbolic execution tree. Reusable evaluators extend state merging to programs with loops. Because of this extension, their behavior falls outside of the well-understood semantics for both bounded model checking, which assumes loop-free programs, and symbolic execution, which assumes path-based evaluation.

In contrast to the classic merging semantics from bounded model checking, $\mathcal{S}_c$ targets programs with loops, has a mechanized proof of soundness and completeness, and preserves legality and reducibility. These features are designed to facilitate reuse. Legality helps developers reuse the evaluator's output to formulate custom queries, while reducibility helps developers test their tools. For example, if a tool targets a language with a concrete reference implementation (e.g., a CPU emulator for an ISA), reducibility makes it possible to test the tool's modeling of the reference semantics on concrete programs and inputs [60].

GL [86] is a related effort that provides verified implementations of symbolic evaluators in

the ACL2 theorem prover. GL is used to automate proofs within ACL2, and uses binary-decision diagrams (BDDs) to represent symbolic expressions. Given some symbolic input, the symbolic evaluator computes the truth value of a theorem as a BDD; if the BDD is the constant 'true', then the theorem can be shown to hold via the correctness of the symbolic evaluator. GL includes two methods of symbolic evaluation: one is a compiler that transforms concrete ACL2 functions into symbolic equivalents, and the other is an interpreter over the ACL2 syntax. Both can be parameterized with a set of primitive functions, and both merge values at control-flow joins. This merging algorithm is fixed in GL, as is the shape of the generated symbolic evaluation DAG. GL uses the classic merging semantics and forces termination through the use of fuel.

Compared to GL, our framework is more general. First, $\mathcal{S}_c$ is defined with respect to a symbolic factory interface, which is not restricted to a specific merging algorithm or underlying representation of symbolic values. Second, our target language is a strict superset of the GL language. To support reuse, it includes first-class procedures, assumptions, and assertions, which are missing from GL. Finally, $\mathcal{S}_c$ is formalized as a big-step operational semantics, via an inductive predicate in Lean, so $\mathcal{S}_c$ evaluators do not need to use fuel to bound executions, whereas GL evaluators do. The main disadvantage of a fuel-based semantics is that it forces implementations to track the fuel value. This tracking can be difficult to implement for evaluators that are realized as a shallow embedding in a host language. Such an evaluator is based on the host's unbounded interpreter, and it usually has no easy way to bound the host interpreter's behavior.

2.3 λ_c: A Core Language for Reusable Symbolic Evaluators

This section presents λ_c, a core language for reusable symbolic evaluators. We begin by describing the syntax, concrete semantics, and key features of λ_c. We then define what it means for a λ_c program to execute normally and to be correct. Finally, we use these definitions to formalize the

Expression	e	::=	`#t` | `#f` | d | $(\lambda x.e)$ | x |	Variable	$x, y, \ldots$
			$(o\ x_1 \ldots x_n)$ | $(x_1\ x_2)$ | (**let** $(x\ e_1)\ e_2$) |	Constant	$d \in D$
			(**if** $x\ e_1\ e_2$) | (**error**) | (**abort**)	Operator	$o \in O$

Figure 2.2: Syntax for λ_c, parameterized by the set of primitive values D and operators O. The literals `#t` and `#f` stand for the boolean values 'true' and 'false', respectively.

angelic execution [11] and *verification* queries for λ_c programs. Solving these queries is the core computational task performed by solver-aided tools. For example, many synthesis tools are based on the CEGIS algorithm [83], which combines a demonic verifier and an angelic guesser to solve synthesis queries. Sections 2.4 and 2.5 will describe how to solve these queries using our symbolic semantics for λ_c.

2.3.1 The Syntax and Concrete Semantics of λ_c

Figure 2.2 shows the syntax of λ_c. The language extends Core Scheme [37] with the ability to express assumptions and assertions using the (**error**) and (**abort**) expressions. Intuitively, (**error**) is equivalent to asserting false, and (**abort**) is equivalent to assuming false. Combined with conditionals, these constructs can be used to encode arbitrary assertions and assumptions over λ_c expressions. Compared to Core Scheme, we place a light restriction on the syntax of λ_c by requiring the arguments to procedures and primitive operators to be variables. This restriction simplifies our formalization and proofs without sacrificing expressiveness. In particular, every Core Scheme program can be converted to an equivalent λ_c program by using **let** expressions; e.g., the application expression $(e_1\ e_2)$ becomes (**let** $(x_1\ e_1)$ (**let** $(x_2\ e_2)\ (x_1\ x_2)$)). Like Core Scheme, the syntax of λ_c is parameterized by the set of primitive values (D) and primitive operators (O). Booleans and procedures are the only constants fixed by the language.

Figure 2.3 shows our big-step operational semantics for λ_c. The judgment $\langle e, E\rangle \Downarrow r$ states that the expression e produces the result r when evaluated in the environment E. The environment

$E \in \mathbb{E}$ is a finite map from variables to values. A value $v \in V$ is a boolean constant, a primitive constant from the set D, or a closure that combines a procedure with an environment.[1] The result $r \in R$ is $\mathsf{Ans}(v)$ if the evaluation terminates normally and produces the value v; Err if it errors; and Abt if it aborts.

The evaluation rules for λ_c are standard. Atomic expressions terminate and produce the expected results (LITERAL, CLOSURE, ERROR, and ABORT). Operator calls (CALLOP) also terminate, producing a value, aborting, or erroring. Their meaning is given by the function $op : O \to V^* \to R$, which is a parameter to the semantics of λ_c. Unlike operator calls, procedure calls need not terminate. In particular, a procedure call $(x_1\ x_2)$ evaluates both of its arguments and checks if x_1 is bound to a closure $\langle \mathbf{cl}\ x, e, E_1 \rangle$. If not, the call errors (CALLBAD). Otherwise, the call evaluates e in the environment $E_1[x \mapsto v_2]$, which binds x to the value of x_2 in E_1. The result of this evaluation, if any, is the result of the call (CALL). The rules for evaluating **let** expressions are similar (LET and LETHALT). Finally, a conditional expression (**if** $x\ e_1\ e_2$) evaluates to the result of e_1 if x is not bound to `#f`, and otherwise, it evaluates to e_2 (IFTRUE and IFFALSE). This semantics mirrors that of Core Scheme: conditionals treat every value except `#f` as 'true'.

While the syntax and semantics of λ_c are largely standard, one difference from prior work is worth noting. The language places no restrictions on the use of variable names, so a program can include free variables. Prior work on verified compilation [22] prevented this by using parametric higher-order abstract syntax. We intentionally allow programs to contain free variables, and treat them as inputs to the program, supplied by the environment. The semantics gets stuck if the environment does not bind a variable referenced during execution (VARIABLE). Treating free variables as inputs lets us introduce symbolic values into a program without having to include a dedicated syntactic construct for creating symbolic values—and having to specify some concrete semantics for it.

The semantics of λ_c is deterministic (Theorem 2.1), like that of Core Scheme. Given a program

[1] Our formalization also includes lists but we omit them from the presentation for brevity.

$$\textsc{Literal}\frac{v \in B \cup D}{\langle v, E\rangle \Downarrow \mathsf{Ans}(v)} \quad \textsc{Closure}\frac{}{\langle(\lambda x.e), E\rangle \Downarrow \mathsf{Ans}(\langle \mathbf{cl}\ x, e, E\rangle)} \quad \textsc{Variable}\frac{x \in \mathsf{dom}(E)}{\langle x, E\rangle \Downarrow \mathsf{Ans}(E[x])}$$

$$\textsc{CallOp}\frac{\langle x_1, E\rangle \Downarrow \mathsf{Ans}(v_1) \ldots \langle x_n, E\rangle \Downarrow \mathsf{Ans}(v_n)}{\langle(o\ x_1 \ldots x_n), E\rangle \Downarrow op(o, v_1, \ldots, v_n)} \quad \textsc{CallBad}\frac{\langle x_1, E\rangle \Downarrow \mathsf{Ans}(v_1) \quad \langle x_2, E\rangle \Downarrow \mathsf{Ans}(v_2) \quad v_1 \notin C}{\langle(x_1\ x_2), E\rangle \Downarrow \mathsf{Err}}$$

$$\textsc{Call}\frac{\langle x_1, E\rangle \Downarrow \mathsf{Ans}(\langle \mathbf{cl}\ x, e, E_1\rangle) \quad \langle x_2, E\rangle \Downarrow \mathsf{Ans}(v_2) \quad \langle e, E_1[x \mapsto v_2]\rangle \Downarrow r}{\langle(x_1\ x_2), E\rangle \Downarrow r}$$

$$\textsc{Let}\frac{\langle e_1, E\rangle \Downarrow \mathsf{Ans}(v_1) \quad \langle e_2, E[x \mapsto v_1]\rangle \Downarrow r}{\langle(\mathbf{let}\ (x\ e_1)\ e_2), E\rangle \Downarrow r} \quad \textsc{LetHalt}\frac{\langle e_1, E\rangle \Downarrow r \quad r = \mathsf{Err} \vee r = \mathsf{Abt}}{\langle(\mathbf{let}\ (x\ e_1)\ e_2), E\rangle \Downarrow r}$$

$$\textsc{IfTrue}\frac{\langle x, E\rangle \Downarrow \mathsf{Ans}(v) \quad v \neq \texttt{\#f} \quad \langle e_1, E\rangle \Downarrow r}{\langle(\mathbf{if}\ x\ e_1\ e_2), E\rangle \Downarrow r} \quad \textsc{IfFalse}\frac{\langle x, E\rangle \Downarrow \mathsf{Ans}(v) \quad v = \texttt{\#f} \quad \langle e_2, E\rangle \Downarrow r}{\langle(\mathbf{if}\ x\ e_1\ e_2), E\rangle \Downarrow r}$$

$$\textsc{Error}\frac{}{\langle(\mathbf{error}), E\rangle \Downarrow \mathsf{Err}} \quad \textsc{Abort}\frac{}{\langle(\mathbf{abort}), E\rangle \Downarrow \mathsf{Abt}}$$

$$E \in \mathbb{E} ::= \mathsf{Variable} \twoheadrightarrow V \quad v \in V ::= b \mid d \mid c \quad b \in B ::= \texttt{\#t} \mid \texttt{\#f} \quad c \in C ::= \langle \mathbf{cl}\ x, e, E\rangle \quad r \in R ::= \mathsf{Ans}(v) \mid \mathsf{Err} \mid \mathsf{Abt}$$

Figure 2.3: Concrete semantics for λ_c, parameterized by the set of primitive values D, operators O, and function $op : O \to V^* \to R$, which gives meaning to the operators $o \in O$. The notation V^* stands for a sequence of values.

and an environment, it either diverges or produces a unique result. In the rest of this work, when we write about the result of a program, we mean this unique result, if one exists.

Theorem 2.1 *Evaluating an expression from the same environment produces the same result:* $\forall e, E, r_1, r_2.\ (\langle e, E\rangle \Downarrow \hat{r}_1 \wedge \langle e, E\rangle \Downarrow \hat{r}_2) \to \hat{r}_1 = \hat{r}_2$.

Example 2.1 *To illustrate the syntax and semantics of λ_c, consider an instantiation $\lambda_{\mathbb{Z}}$ where D is the set of all integers, O consists of the operators $\{-, <, =\}$, and op gives these operators their standard meaning over integers. Using this instantiation, we can write the following program:*

```
1    (let (z 0)                      ; z is zero
2    (let (abs (λx . (let (b0 (< x z)) (if b0 (- x) x))))
3    (let (b1 (= y z))
4    (let (_  (if b1 (abort) #t))    ; assume y != 0
5    (let (y0 (abs y))
6    (let (b2 (< z y0))
7    (let (_  (if b2 #t (error)))    ; assert |y| > 0
8      _)))))))
```

The program, e_{abs}, asserts that the absolute value of its free variable y is positive, assuming that y is not 0. The evaluation of this program gets stuck in all environments that have no binding for y. We also have $\langle e_{\text{abs}}, \{y \mapsto -1\}\rangle \Downarrow \mathsf{Ans}(\#t)$ and $\langle e_{\text{abs}}, \{y \mapsto 0\}\rangle \Downarrow \mathsf{Abt}$.

2.3.2 The Angelic Execution and Verification Queries for λ_c

Given the syntax and semantics of λ_c, we define what it means for an execution to be normal and for a program to be correct (Definitions 2.1 and 2.2). A program executes normally in a given environment if it terminates and produces $\mathsf{Ans}(v)$ for some value $v \in V$. The execution errors if it produces Err. If a program does not error in any environment $E \in \mathcal{E}$, we say that it is correct with respect to the input space defined by the set of environments $\mathcal{E}$.

Definition 2.1 (Executions of λ_c programs) *A program e evaluates* normally *in an environment E iff $\langle e, E\rangle \Downarrow \mathsf{Ans}(v)$ for some value $v \in V$; it* errors *iff $\langle e, E\rangle \Downarrow \mathsf{Err}$; and it* aborts *iff $\langle e, E\rangle \Downarrow \mathsf{Abt}$. We denote these outcomes with $\mathsf{normal}(e, E)$, $\mathsf{errors}(e, E)$, or $\mathsf{aborts}(e, E)$, respectively.*

Definition 2.2 (Correctness of λ_c programs) *A program e is* correct *with respect to the set of environments $\mathcal{E}$ iff it does not error in any $E \in \mathcal{E}$, i.e., $\mathsf{correct}(e, \mathcal{E}) ::= \forall E \in \mathcal{E}.\, \neg\mathsf{errors}(e, E)$.*

These two definitions underlie the core computational tasks performed by solver-aided tools: angelic execution and verification. Both tasks, which we call *queries*, can be understood as forms of search. Angelic execution searches for a normal execution of a given program, while verification searches for an execution that errors. We formalize both (Definitions 2.3 and 2.4) as partial functions from a program and a set of environments to an environment that satisfies the query or **unsat** if the query is unsatisfiable. The functions are partial because the two queries may not terminate in general. But if they terminate, they must produce a correct result—i.e., we take them to be sound semi-decision procedures for the angelic execution and verification problems.

Definition 2.3 (Angelic execution) *Given a program e and set of environments $\mathcal{E}$, the angelic execution query* $\mathsf{guess}(e, \mathcal{E})$ *diverges or produces one of two results: either an environment $E \in \mathcal{E}$ such that* $\mathsf{normal}(e, E)$*, or* **unsat** *if no such environment exists in $\mathcal{E}$.*

Definition 2.4 (Verification) *Given a program e and set of environments $\mathcal{E}$, the verification query* $\mathsf{verify}(e, \mathcal{E})$ *diverges or produces one of two results: either an environment $E \in \mathcal{E}$ such that* $\mathsf{errors}(e, E)$*, or* **unsat** *if no such environment exists in $\mathcal{E}$.*

Example 2.2 *To illustrate Definitions 2.1–2.4, consider the program e_{abs} from Example 2.1. This program is correct with respect to $\mathcal{E}_{\mathbb{Z}}$, the set of all environments that bind y to an integer. It also has infinitely many normal executions with respect to this set. As a result,* $\mathsf{verify}(e_{\text{abs}}, \mathcal{E}_{\mathbb{Z}}) =$ **unsat***, and* $\mathsf{guess}(e_{\text{abs}}, \mathcal{E}_{\mathbb{Z}})$ *may return any environment that binds y to a non-zero value, e.g., $\{y \mapsto -1\}$. But e_{abs} is not correct with respect to $\mathcal{E}_V$, the set of all environments that bind y to* any *value. In particular, it will error at line 2 in every environment that binds y to a non-integer value, since the equality operator $=$ expects its inputs to be integers. The verification query* $\mathsf{verify}(e_{\text{abs}}, \mathcal{E}_V)$ *may return any of these environments, e.g., $\{y \mapsto \texttt{\#t}\}$. Since every environment that is in $\mathcal{E}_V$ but not in $\mathcal{E}_{\mathbb{Z}}$ leads to an error,* $\mathsf{guess}(e_{\text{abs}}, \mathcal{E}_V)$ *and* $\mathsf{guess}(e_{\text{abs}}, \mathcal{E}_{\mathbb{Z}})$ *draw their outputs from the same non-empty subset of $\mathcal{E}_{\mathbb{Z}}$. The next section shows how to automate these queries using our symbolic semantics for λ_c.*

2.4 $\mathcal{S}_c$: A Semantics for Symbolic Evaluation with Merging

To automate angelic execution and verification, solver-aided tools rely on symbolic evaluation to express the $\mathsf{guess}(e, \mathcal{E})$ and $\mathsf{verify}(e, \mathcal{E})$ queries as logical formulas. This section presents $\mathcal{S}_c$, a new symbolic semantics for reducing λ_c programs to formulas. To accommodate a wide range of practical implementations, $\mathcal{S}_c$ is parameterized by a *symbolic factory*, which is a set of abstract functions for creating and manipulating *symbolic values*. We start by formalizing these notions

and relating them to our concrete semantics. Next, we present the symbolic evaluation rules for $\mathcal{S}_c$, and contrast them to the classic merging semantics [9]. We conclude this section by defining guess$(e, \mathcal{E})$ and verify$(e, \mathcal{E})$ in terms of the symbolic states computed by $\mathcal{S}_c$. The next section establishes the correctness of $\mathcal{S}_c$ and the queries we define on top of it.

2.4.1 The Symbolic Factory Interface

At a high level, a symbolic semantics *lifts* the rules of a concrete semantics to operate on sets of concrete values, compactly represented as *symbolic values*. Practical evaluators have different ways of representing and manipulating symbolic values, and as we saw in Section 2.1, these differences can lead to fundamentally different behaviors. To cover all reasonable behaviors in our formalization, we define $\mathcal{S}_c$ against the abstract factory interface shown in Figure 2.4.

The factory interface is based on three core types: models M, symbolic values $\hat{V}$, and symbolic booleans $\hat{B}$. All three types are parameters to the factory, in addition to the parameters D, O, and *op* inherited from λ_c. Conceptually, a symbolic boolean is a logical constraint on symbolic values; a symbolic value is an expression over symbolic variables; a model maps symbolic variables to concrete constants; and each symbolic boolean and value represents a unique concrete value under a given model. The factory interface captures these relationships abstractly through the interpretation functions $[\![\cdot]\!]^{\hat{B}}$ and $[\![\cdot]\!]^{\hat{V}}$, which use models to give concrete meaning to symbolic booleans and values, respectively. We specify the rest of the interface in terms of these core types and functions, dropping the superscript from the interpreter notation when it is clear from the context.

The factory types and functions provide a basic mechanism for lifting the semantics of λ_c. They serve as the symbolic counterparts of the concrete types, functions, and predicates that are used to define the concrete evaluation rules for λ_c (Figure 2.3). We first explain how the factory components help lift these rules, and then illustrate a toy factory for the language $\lambda_{\mathbb{Z}}$ from Example 2.1.

Symbolic values	$\hat{v} \in \hat{V} \qquad [\![\cdot]\!]^{\hat{V}} : M \to \hat{V} \to V$
$\mathsf{truth} : \hat{V} \to \hat{B}$	$\forall m, \hat{v}. [\![\mathsf{truth}(\hat{v})]\!]^{\hat{B}}_m \leftrightarrow ([\![\hat{v}]\!]^{\hat{V}}_m \neq \#\mathtt{f})$
$\mathsf{lift} : (B \cup D \cup \hat{C}) \to \hat{V}$	$\forall m, b. [\![\mathsf{lift}(b)]\!]^{\hat{V}}_m = b \quad \forall m, d. [\![\mathsf{lift}(d)]\!]^{\hat{V}}_m = d \quad \forall m, \hat{c}. [\![\mathsf{lift}(\hat{c})]\!]^{\hat{V}}_m = [\![\hat{c}]\!]^{\hat{C}}_m$
$\mathsf{merge} : \mathbb{G}_{\hat{V}} \to \hat{V}$	$\forall m, G. \mathsf{one}_m(G) \to [\![\mathsf{merge}(G)]\!]_m = [\![G]\!]^{\mathbb{G}_{\hat{V}}}_m$
$\mathsf{cast} : \hat{V} \to \mathbb{G}_{\hat{C}}$	$\forall m, \hat{v}. [\![\hat{v}]\!]_m \in C \to (\mathsf{one}_m(\mathsf{cast}(\hat{v})) \wedge [\![\mathsf{cast}(\hat{v})]\!]^{\mathbb{G}_{\hat{C}}}_m = [\![\hat{v}]\!]_m)$
	$\forall m, \hat{v}. [\![\hat{v}]\!]_m \notin C \to \mathsf{none}_m(\mathsf{cast}(\hat{v}))$
$\hat{op} : O \to \hat{V}^* \to \hat{R}$	$\forall m, \hat{v}_1, \ldots, \hat{v}_k. [\![\hat{op}(\hat{v}_1, \ldots, \hat{v}_k)]\!]^{\hat{R}}_m = op([\![\hat{v}_1]\!]_m, \ldots, [\![\hat{v}_k]\!]_m) \wedge \mathsf{legal}_m(\hat{op}(\hat{v}_1, \ldots, \hat{v}_k))$
Symbolic booleans	$\hat{b} \in \hat{B} \qquad [\![\cdot]\!]^{\hat{B}} : M \to \hat{B} \to B$
$\mathtt{tt}, \mathtt{ff} : \hat{B}$	$\forall m. [\![\mathtt{tt}]\!]_m = \#\mathtt{t} \qquad \forall m. [\![\mathtt{ff}]\!]_m = \#\mathtt{f}$
$\mathsf{not} : \hat{B} \to \hat{B}$	$\forall m, \hat{b}. [\![\mathsf{not}(\hat{b})]\!]_m = \neg [\![\hat{b}]\!]_m$
$\mathsf{and} : \hat{B} \to \hat{B} \to \hat{B}$	$\forall m, \hat{b}_1, \hat{b}_2. [\![\mathsf{and}(\hat{b}_1, \hat{b}_2)]\!]_m = ([\![\hat{b}_1]\!]_m \wedge [\![\hat{b}_2]\!]_m)$
$\mathsf{or} : \hat{B} \to \hat{B} \to \hat{B}$	$\forall m, \hat{b}_1, \hat{b}_2. [\![\mathsf{or}(\hat{b}_1, \hat{b}_2)]\!]_m = ([\![\hat{b}_1]\!]_m \vee [\![\hat{b}_2]\!]_m)$
$\mathsf{imp} : \hat{B} \to \hat{B} \to \hat{B}$	$\forall m, \hat{b}_1, \hat{b}_2. [\![\mathsf{imp}(\hat{b}_1, \hat{b}_2)]\!]_m = ([\![\hat{b}_1]\!]_m \to [\![\hat{b}_2]\!]_m)$
Symbolic closure	$\hat{c} \in \hat{C} ::= \langle \hat{\mathbf{cl}}\ x, e, \hat{E} \rangle$
$[\![\cdot]\!]^{\hat{C}} : M \to \hat{C} \to V$	$[\![\langle \hat{\mathbf{cl}}\ x, e, \hat{E} \rangle]\!]^{\hat{C}}_m ::= \langle \mathbf{cl}\ x, e, [\![\hat{E}]\!]^{\hat{\mathbb{E}}}_m \rangle$
Symbolic environment	$\hat{E} \in \hat{\mathbb{E}} ::= \mathsf{Variable} \rightharpoonup \hat{V}$
$[\![\cdot]\!]^{\hat{\mathbb{E}}} : M \to \hat{\mathbb{E}} \to \mathbb{E}$	$[\![\hat{E}]\!]^{\hat{\mathbb{E}}}_m ::= \{x \mapsto v \mid x \in \mathsf{dom}(\hat{E}), \hat{v} = \hat{E}[x], [\![\hat{v}]\!]^{\hat{V}}_m = v\}$
Guarded choice	$g \in \hat{B} \times \alpha ::= \langle \hat{b}, a \rangle \qquad$ where $a \in \alpha, [\![a]\!]^{\alpha}_m \in \beta \qquad \mathsf{guard}(g) ::= \hat{b} \qquad \mathsf{choice}(g) ::= a$
Guarded choices	$G \in \mathbb{G}_{\alpha} ::= [g_1, \ldots, g_n]$
$[\![\cdot]\!]^{\mathbb{G}_{\alpha}} : M \to \mathbb{G}_{\alpha} \to \beta$	$[\![g :: G]\!]^{\mathbb{G}_{\alpha}}_m ::= \mathbf{if}\ [\![\mathsf{guard}(g)]\!]_m\ \mathbf{then}\ [\![\mathsf{choice}(g)]\!]^{\alpha}_m\ \mathbf{else}\ [\![G]\!]^{\mathbb{G}_{\alpha}}_m$
	$[\![\ [\ ]\ ]\!]^{\mathbb{G}_{\alpha}}_m \quad ::= \mathsf{default}(\beta)$
$\mathsf{one} : M \to \mathbb{G}_{\alpha} \to B$	$\mathsf{one}_m(G) \quad ::= \mathbf{length}(\mathbf{filter}(\lambda g. [\![\mathsf{guard}(g)]\!]_m, G)) = 1$
$\mathsf{none} : M \to \mathbb{G}_{\alpha} \to B$	$\mathsf{none}_m(G) ::= \mathbf{length}(\mathbf{filter}(\lambda g. [\![\mathsf{guard}(g)]\!]_m, G)) = 0$
Symbolic state	$\sigma \in \Sigma ::= \langle \hat{b}_1, \hat{b}_2 \rangle \qquad \mathsf{assumes}(\sigma) ::= \hat{b}_1 \qquad \mathsf{asserts}(\sigma) ::= \hat{b}_2$
$\mathsf{normal} : M \to \Sigma \to B$	$\mathsf{normal}_m(\sigma) ::= [\![\mathsf{assumes}(\sigma)]\!]_m \wedge [\![\mathsf{asserts}(\sigma)]\!]_m$
$\mathsf{aborts} : M \to \Sigma \to B$	$\mathsf{aborts}_m(\sigma) ::= \neg [\![\mathsf{assumes}(\sigma)]\!]_m \wedge [\![\mathsf{asserts}(\sigma)]\!]_m$
$\mathsf{errors} : M \to \Sigma \to B$	$\mathsf{errors}_m(\sigma) ::= [\![\mathsf{assumes}(\sigma)]\!]_m \wedge \neg [\![\mathsf{asserts}(\sigma)]\!]_m$
$\mathsf{legal} : M \to \Sigma \to B$	$\mathsf{legal}_m(\sigma) ::= [\![\mathsf{assumes}(\sigma)]\!]_m \vee [\![\mathsf{asserts}(\sigma)]\!]_m$
$\equiv : M \to \Sigma \to \Sigma \to B$	$\sigma_1 \equiv_m \sigma_2 ::= [\![\mathsf{assumes}(\sigma_1)]\!]_m = [\![\mathsf{assumes}(\sigma_2)]\!]_m \wedge [\![\mathsf{asserts}(\sigma_1)]\!]_m = [\![\mathsf{asserts}(\sigma_2)]\!]_m$
Symbolic result	$\hat{r} \in \hat{R} ::= \mathsf{Out}(\sigma, \hat{v}) \mid \mathsf{Halt}(\sigma)$
	$\mathsf{state}(\hat{r}) ::= \sigma \qquad \mathsf{value}(\mathsf{Out}(\sigma, \hat{v})) ::= \hat{v} \qquad \mathsf{value}(\mathsf{Halt}(\sigma)) ::= \mathsf{default}(\hat{V})$
$\mathsf{Result} : \Sigma \to \hat{V} \to \hat{R}$	$\mathsf{Result}(\sigma, \hat{v}) ::= \mathbf{if}\ (\mathsf{assumes}(\sigma) = \mathtt{ff} \vee \mathsf{asserts}(\sigma) = \mathtt{ff})\ \mathbf{then}\ \mathsf{Halt}(\sigma)\ \mathbf{else}\ \mathsf{Out}(\sigma, \hat{v})$
$[\![\cdot]\!]^{\hat{R}} : M \to \hat{R} \to R$	$[\![\mathsf{Out}(\sigma, \hat{v})]\!]^{\hat{R}}_m ::= \mathbf{if}\ \mathsf{normal}_m(\sigma)\ \mathbf{then}\ \mathsf{Ans}([\![\hat{v}]\!]_m)\ \mathbf{else\ if}\ \mathsf{aborts}_m(\sigma)\ \mathbf{then}\ \mathsf{Abt}\ \mathbf{else}\ \mathsf{Err}$
	$[\![\mathsf{Halt}(\sigma)]\!]^{\hat{R}}_m \quad ::= \mathbf{if}\ \mathsf{aborts}_m(\sigma)\ \mathbf{then}\ \mathsf{Abt}\ \mathbf{else}\ \mathsf{Err}$
$\mathsf{legal} : M \to \hat{R} \to B$	$\mathsf{legal}_m(\mathsf{Out}(\sigma, \hat{v})) ::= \mathsf{legal}_m(\sigma)$
	$\mathsf{legal}_m(\mathsf{Halt}(\sigma)) \quad ::= \mathsf{legal}_m(\sigma) \wedge \neg \mathsf{normal}_m(\sigma)$

Figure 2.4: Symbolic factory interface, parameterized by the set of all models M, symbolic values $\hat{V}$, and symbolic booleans $\hat{B}$, as well as the parameters O, D, and op from the definition of λ_c. We use $[\![\cdot]\!]^{\alpha}$ to denote the interpreter for values of type α, omitting the superscript α when it is clear from the context. The hat accent denotes the symbolic counterpart of a concrete entity; e.g., $\hat{v}$ is a symbolic value, where v is a concrete value.

Lifting constants. The function $\mathsf{lift} : (B \cup D \cup \hat{C}) \to \hat{V}$ provides a way to lift the meaning of constant expressions (LITERAL and CLOSURE in Figure 2.3). Given an input i, $\mathsf{lift}(i)$ returns a symbolic value that has the same interpretation as i under all models. The input i is either a concrete boolean, a concrete constant of type D, or a symbolic closure. A symbolic closure $\langle \hat{\mathbf{cl}}\ x, e, \hat{E} \rangle \in \hat{C}$ is a straightforward generalization of a concrete closure: it combines a procedure $(\lambda x.e)$ with a symbolic environment $\hat{E} \in \hat{\mathbb{E}}$, which maps variable names to symbolic values. Symbolic closures and environments evaluate to their concrete counterparts via the interpretation functions $[\![\cdot]\!]^{\hat{C}}$ and $[\![\cdot]\!]^{\hat{\mathbb{E}}}$. Conversely, concrete closures and environments can be made symbolic by recursively lifting their contents via $\mathsf{lift}(i)$.

Lifting conditionals. The functions $\mathsf{truth} : \hat{V} \to \hat{B}$ and $\mathsf{merge} : \mathbb{G}_{\hat{V}} \to \hat{V}$ help lift the semantics of conditional expressions (IFTRUE and IFFALSE). The former lifts the conditional test for λ_c, and the latter lifts the output of conditional evaluation. In particular, $\mathsf{truth}(\hat{v})$ encodes a logical predicate on $\hat{v}$ that is true unless $\hat{v}$ evaluates to `#f`, while $\mathsf{merge}(G)$ encodes the selection of a value from a list $G \in \mathbb{G}_{\hat{V}}$ of *guarded choices*. A guarded choice $g \in \hat{B} \times \hat{V}$ pairs a symbolic boolean with a symbolic value (or, more generally, any value with an interpreter). When a list G of such choices has one true guard under a given model, $\mathsf{merge}(G)$ evaluates to the choice with the true guard. For example, $\mathsf{merge}([\langle \hat{b}, \hat{v}_1 \rangle, \langle \mathsf{not}(\hat{b}), \hat{v}_2 \rangle])$ evaluates to $[\![\hat{v}_1]\!]_m$ if $[\![\hat{b}]\!]_m$ is true, and to $[\![\hat{v}_2]\!]_m$ otherwise. If G has no or many true guards under a given model, the behavior of $\mathsf{merge}(G)$ is unspecified for that model, and irrelevant in the context of the symbolic semantics $\mathcal{S}_c$.

Lifting procedure calls. The function $\mathsf{cast} : \hat{V} \to \mathbb{G}_{\hat{C}}$ lifts the dynamic cast from values to closures that is implicit in the semantics of procedure calls. The semantics uses two rules (CALL and CALLBAD) to handle the results of successful and failed casts on concrete values. We use $\mathsf{cast}(\hat{v})$ to make this operation explicit on symbolic values. The result of a symbolic cast is a list

of guarded symbolic closures, $\text{cast}(\hat{v}) \in \mathbb{G}_{\hat{C}}$. This list selects at most one closure under every model to match the behavior of $\hat{v}$: if $[\![\hat{v}]\!]_m$ is a closure, then $\text{cast}(\hat{v})$ evaluates to $[\![\hat{v}]\!]_m$, and if not, all guards in $\text{cast}(\hat{v})$ are false under m, indicating that the cast has failed. As an example, $\text{cast}(\text{merge}([\langle\hat{b}, \text{lift}(\hat{c})\rangle, \langle\text{not}(\hat{b}), \text{lift}(\texttt{\#f})\rangle]))$ produces a list of guarded closures that is equivalent to $[\langle\hat{b}, \hat{c}\rangle]$ under every model. This list evaluates to $[\![\hat{c}]\!]_m$ when $[\![\hat{b}]\!]_m$ is true, just like the input to the cast, and has no true guards otherwise.

Lifting operator calls. The function $\hat{op} : O \to \hat{V}^* \to \hat{R}$ lifts the function $op : O \to V^* \to R$, which defines the semantics of operator calls (CallOp). Given a sequence $\hat{v}^* = [\hat{v}_1, \ldots, \hat{v}_k]$ of symbolic values, $\hat{op}(\hat{v}^*)$ produces a *symbolic result* that evaluates to the concrete result of $op([\![\hat{v}_1]\!]_m, \ldots, [\![\hat{v}_k]\!]_m)$ under every model m. Section 2.4.2 describes symbolic results and states in detail. For now, it suffices to note that the symbolic result of $\hat{op}$ evaluates to the concrete result of op as expected.

Example 2.3 *To illustrate the factory interface, consider the toy factory in Figure 2.5. This factory implements the interface for the language $\lambda_{\mathbb{Z}}$ from Example 2.1 as follows.*

First, we define the symbolic boolean, integer, value, and model types for $\lambda_{\mathbb{Z}}$. Our implementation supports only one kind of symbolic variables, symbolic integers $z \in Z$, so models $m \in M_{\mathbb{Z}}$ map symbolic integer variables to integer constants. Symbolic values include symbolic booleans, integers, closures (Figure 2.4), and ϕ expressions, which select between two symbolic values based on a symbolic boolean guard. We use ϕ expressions to represent conditionals for simplicity; practical factories rely on more efficient representations (see, e.g., [93, 80]).

Next, we implement standard bottom-up interpreters for our base types, followed by a basic implementation of the logical operations provided by the factory. Our logical operators simplify their outputs when given concrete inputs and perform no other optimizations. In contrast, practical implementations use dozens of logical equivalences to simplify their outputs as much as possible.

Symbolic booleans $\hat{B}_{\mathbb{Z}}$, integers $\hat{D}_{\mathbb{Z}}$, values $\hat{V}_{\mathbb{Z}}$, and models $M_{\mathbb{Z}}$ for $\lambda_{\mathbb{Z}}$:

$\hat{b} \in \hat{B}_{\mathbb{Z}} ::= b \mid (\texttt{!}\ \hat{b}) \mid (\hat{b}_1\ \texttt{\&\&}\ \hat{b}_2) \mid (\hat{d}_1 \sim \hat{d}_2)$ $\quad b \in B \quad ::= \texttt{\#t} \mid \texttt{\#f}$ $\quad \sim \quad ::= < \mid =$

$\hat{d} \in \hat{D}_{\mathbb{Z}} ::= d \mid z \mid (-\hat{d})$ $\quad z \in Z \quad ::=$ symbolic integer variable $\quad d \in \mathbb{Z} ::=$ integer constant

$\hat{v} \in \hat{V}_{\mathbb{Z}} ::= \hat{b} \mid \hat{d} \mid \hat{c} \mid \phi(\hat{b}, \hat{v}_1, \hat{v}_2)$ $\quad m \in M_{\mathbb{Z}} ::= Z \to \mathbb{Z}$

Interpreters for symbolic booleans $\hat{B}_{\mathbb{Z}}$, integers $\hat{D}_{\mathbb{Z}}$, and values $\hat{V}_{\mathbb{Z}}$, with respect to models $M_{\mathbb{Z}}$:

$[\![b]\!]_m^{\hat{B}_{\mathbb{Z}}} ::= b$ $\quad [\![(\texttt{!}\ \hat{b})]\!]_m^{\hat{B}_{\mathbb{Z}}} ::= \neg [\![\hat{b}]\!]_m^{\hat{B}_{\mathbb{Z}}}$ $\quad [\![(\hat{b}_1\ \texttt{\&\&}\ \hat{b}_2)]\!]_m^{\hat{B}_{\mathbb{Z}}} ::= [\![\hat{b}_1]\!]_m^{\hat{B}_{\mathbb{Z}}} \wedge [\![\hat{b}_2]\!]_m^{\hat{B}_{\mathbb{Z}}}$ $\quad [\![(\hat{d}_1 \sim \hat{d}_2)]\!]_m^{\hat{B}_{\mathbb{Z}}} ::= [\![\hat{d}_1]\!]_m^{\hat{D}_{\mathbb{Z}}} \sim [\![\hat{d}_2]\!]_m^{\hat{D}_{\mathbb{Z}}}$

$[\![d]\!]_m^{\hat{D}_{\mathbb{Z}}} ::= d$ $\quad [\![z]\!]_m^{\hat{D}_{\mathbb{Z}}} ::= m[z]$ $\quad [\![(-\hat{d})]\!]_m^{\hat{D}_{\mathbb{Z}}} ::= -[\![d]\!]_m^{\hat{D}_{\mathbb{Z}}}$

$[\![\hat{b}]\!]_m^{\hat{V}_{\mathbb{Z}}} ::= [\![\hat{b}]\!]_m^{\hat{B}_{\mathbb{Z}}}$ $\quad [\![\hat{d}]\!]_m^{\hat{V}_{\mathbb{Z}}} ::= [\![\hat{d}]\!]_m^{\hat{D}_{\mathbb{Z}}}$ $\quad [\![\hat{c}]\!]_m^{\hat{V}_{\mathbb{Z}}} ::= [\![\hat{c}]\!]_m^{\hat{C}}$ $\quad [\![\phi(\hat{b}, \hat{v}_1, \hat{v}_2)]\!]_m^{\hat{V}_{\mathbb{Z}}} ::= \textbf{if}\ [\![\hat{b}]\!]_m^{\hat{B}}\ \textbf{then}\ [\![\hat{v}_1]\!]_m^{\hat{V}}\ \textbf{else}\ [\![\hat{v}_2]\!]_m^{\hat{V}}$

Factory operations on symbolic booleans $\hat{B}_{\mathbb{Z}}$:

$\mathsf{ff} ::= \texttt{\#f}$ $\quad \mathsf{not}(b) ::= \neg b$ $\quad \mathsf{and}(\texttt{\#f}, \hat{b}) ::= \texttt{\#f}$ $\quad \mathsf{and}(\hat{b}, b) ::= \mathsf{and}(b, \hat{b})$ $\quad \mathsf{or}(\hat{b}_1, \hat{b}_2) ::= \mathsf{not}(\mathsf{and}(\mathsf{not}(\hat{b}_1), \mathsf{not}(\hat{b}_2)))$

$\mathsf{tt} ::= \texttt{\#t}$ $\quad \mathsf{not}(\hat{b}) ::= (\texttt{!}\ \hat{b})$ $\quad \mathsf{and}(\texttt{\#t}, \hat{b}) ::= \hat{b}$ $\quad \mathsf{and}(\hat{b}_1, \hat{b}_2) ::= (\hat{b}_1\ \texttt{\&\&}\ \hat{b}_2)$ $\quad \mathsf{imp}(\hat{b}_1, \hat{b}_2) ::= \mathsf{not}(\mathsf{and}(\hat{b}_1, \mathsf{not}(\hat{b}_2)))$

Factory operations on symbolic values $\hat{V}_{\mathbb{Z}}$:

$\mathsf{lift}(i) ::= i$

$\mathsf{truth}(\hat{b}) ::= \hat{b}$ $\quad \mathsf{truth}(\hat{d}) ::= \texttt{\#t}$ $\quad \mathsf{truth}(\hat{c}) ::= \texttt{\#t}$

$\mathsf{truth}(\phi(\hat{b}, \hat{v}_1, \hat{v}_2)) ::= \mathsf{and}(\mathsf{imp}(\hat{b}, \mathsf{truth}(\hat{v}_1)), \mathsf{imp}(\mathsf{not}(\hat{b}), \mathsf{truth}(\hat{v}_2)))$

$\mathsf{merge}([\,]) ::= \texttt{\#f}$ $\quad \mathsf{merge}([g]) ::= \mathsf{choice}(g)$ $\quad \mathsf{merge}(g_1 :: g_2 :: G) ::= \phi(\mathsf{guard}(g_1), \mathsf{choice}(g_1), \mathsf{merge}(g_2 :: G))$

$\mathsf{cast}(\hat{b}) ::= []$ $\quad \mathsf{cast}(\hat{d}) ::= []$ $\quad \mathsf{cast}(\hat{c}) ::= [\langle \texttt{\#t}, \hat{c} \rangle]$

$\mathsf{cast}(\phi(\hat{b}, \hat{v}_1, \hat{v}_2)) ::= \textbf{append}(\mathsf{subcast}(\hat{b}, \hat{v}_1), \mathsf{subcast}(\mathsf{not}(\hat{b}), \hat{v}_2))$

$\mathsf{subcast}(\hat{b}, \hat{v}) ::= \textbf{map}(\lambda g.\langle \mathsf{and}(\hat{b}, \mathsf{guard}(g)), \mathsf{choice}(g) \rangle, \mathsf{cast}(\hat{v}))$

$\hat{op}(-, \hat{b}) ::= \mathsf{Halt}(\langle \texttt{\#t}, \texttt{\#f} \rangle)$ $\quad \hat{op}(-, \hat{c}) ::= \mathsf{Halt}(\langle \texttt{\#t}, \texttt{\#f} \rangle)$ $\quad \hat{op}(-, \hat{v}_1, \ldots, \hat{v}_k) ::= \mathsf{Halt}(\langle \texttt{\#t}, \texttt{\#f} \rangle)$ where $k \neq 1$

$\hat{op}(-, d) ::= \mathsf{Out}(\langle \texttt{\#t}, \texttt{\#t} \rangle, -d)$ $\quad \hat{op}(-, \hat{d}) ::= \mathsf{Out}(\langle \texttt{\#t}, \texttt{\#t} \rangle, (-\hat{d}))$ $\quad \hat{op}(-, \phi(\hat{b}, \hat{v}_1, \hat{v}_2)) ::= \ldots$ $\quad \hat{op}(\ldots) ::= \ldots$

Figure 2.5: A toy factory for the language $\lambda_{\mathbb{Z}}$ from Example 2.1. The factory reuses the definition of symbolic closures $\hat{c} \in \hat{C}$ and the interpreter $[\![\cdot]\!]^{\hat{C}}$ from Figure 2.4.

Finally, we implement lift, truth, merge, cast, *and $\hat{op}$. These are straightforward except for* merge *and $\hat{op}$. For example, $[\![\mathsf{truth}(\phi(\hat{b}, \hat{b}_1, \hat{d}_2))]\!]_m$ is equivalent to $([\![\hat{b}]\!]_m \to [\![\hat{b}_1]\!]_m) \wedge (\neg[\![\hat{b}]\!]_m \to [\![\texttt{\#t}]\!]_m)$, which says that* $\mathsf{truth}(\phi(\hat{b}, \hat{b}_1, \hat{d}_2))$ *is false only when $\hat{b}$ is true and $\hat{b}_1$ is false. Similarly,* $\mathsf{cast}(\phi(\hat{b}, \hat{c}, \hat{d}))$ *produces $[\langle \hat{b}, \hat{c} \rangle]$, which evaluates to $\hat{c}$ when $\phi(\hat{b}, \hat{c}, \hat{d})$ does and has no true guards otherwise. To see why* $\mathsf{merge}(G)$ *works, recall that it must match the interpretation of G only under models that make exactly one guard in G true. In this case,* $\mathsf{merge}([])$ *can return anything;* $\mathsf{merge}([g])$ *must return the sole chosen value; and the nested ϕ value produced by* $\mathsf{merge}(g_1 :: g_2 :: G)$ *is equivalent to $g_1 :: g_2 :: G$. For example,* $\mathsf{merge}([\langle \hat{b}, \hat{v}_1 \rangle, \langle \mathsf{not}(\hat{b}), \hat{v}_2 \rangle])$ *returns $\phi(\hat{b}, \hat{v}_1, \hat{v}_2)$. We show the base cases for $\hat{op}(-, \ldots)$ and omit the rest for brevity. For example, $\hat{op}(-)$ errors because it is given the wrong the number of arguments, while $\hat{op}(-, \hat{d})$ terminates normally and produces the right value.*

2.4.2 Evaluation Rules for $\mathcal{S}_c$

Given a symbolic factory, we define the symbolic semantics $\mathcal{S}_c$ using the rules shown in Figure 2.6. The rules lift the concrete semantics of λ_c (Figure 2.3) to work on symbolic values. The judgment $\langle e, \hat{E}, \sigma \rangle \Downarrow \hat{r}$ says that the expression e produces the symbolic result $\hat{r}$ when evaluated in the symbolic environment $\hat{E}$ and state σ. Intuitively, this judgment encodes a set of concrete executions $\langle e, [\![\hat{E}]\!]_m \rangle \Downarrow [\![\hat{r}]\!]_m$, one for each model $m \in M$. The environment $\hat{E} \in \hat{\mathbb{E}}$ maps variables to symbolic values. The input state $\sigma \in \Sigma$ consists of two formulas, $\mathsf{assumes}(\sigma)$ and $\mathsf{asserts}(\sigma)$, that jointly indicate if prior execution steps terminated normally, errored, or aborted under a given model (Figure 2.4). The result $\hat{r} \in \hat{R}$ is either $\mathsf{Out}(\sigma', \hat{v})$ or $\mathsf{Halt}(\sigma')$. The former means that e may error, abort, or terminate normally to produce $[\![\hat{v}]\!]_m$, with the outcome determined by the output state σ'. The latter means that e cannot terminate normally under any model; i.e., $\mathsf{normal}_m(\sigma')$ is guaranteed to be false. We describe the rules of $\mathcal{S}_c$ below, and discuss the main features of our design in Section 2.4.3.

$$\textsc{Literal}\frac{v \in B \cup D}{\langle v, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \mathsf{lift}(v))} \quad \textsc{Closure}\frac{}{\langle(\lambda x.e), \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \mathsf{lift}(\langle \hat{\mathbf{cl}}\ x, e, \hat{E}\rangle))}$$

$$\textsc{Variable}\frac{x \in \mathsf{dom}(\hat{E})}{\langle x, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \hat{E}[x])} \quad \textsc{CallOp}\frac{\langle x_1, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \hat{v}_1) \ldots \langle x_n, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \hat{v}_n)}{\langle(o\ x_1 \ldots x_n), \hat{E}, \sigma\rangle \Downarrow \mathsf{strengthen}(\sigma, \hat{op}(o, \hat{v}_1, \ldots, \hat{v}_n))}$$

$$\textsc{Call}\frac{\begin{array}{c}\langle x_1, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \hat{v}_1) \quad \langle x_2, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \hat{v}_2) \quad G_1 = \mathsf{cast}(\hat{v}_1) \quad n = \mathbf{length}(G_1) = \mathbf{length}(G_2) \\ \gamma = \mathsf{some}(\mathsf{guard}, G_1) \quad \sigma' = \mathsf{assert}(\sigma, \gamma) \quad \gamma \neq \mathsf{ff} \wedge \mathsf{assumes}(\sigma') \neq \mathsf{ff} \wedge \mathsf{asserts}(\sigma') \neq \mathsf{ff} \\ \forall i \in [0, n).\ \mathbf{let}\ \langle \hat{b}_1, \langle \hat{\mathbf{cl}}\ x, e, \hat{E}_1\rangle\rangle ::= G_1[i], \langle \hat{b}_2, \hat{r}\rangle ::= G_2[i]\ \mathbf{in}\ (\hat{b}_1 = \hat{b}_2 \wedge \langle e, \hat{E}_1[x \mapsto \hat{v}_2], \mathsf{assume}(\sigma', \hat{b}_1)\rangle \Downarrow \hat{r})\end{array}}{\langle(x_1\ x_2), \hat{E}, \sigma\rangle \Downarrow \mathsf{merge}_{\hat{R}}(\sigma', G_2)}$$

$$\textsc{CallBad}\frac{\begin{array}{c}\langle x_1, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \hat{v}_1) \quad \langle x_2, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \hat{v}_2) \quad G_1 = \mathsf{cast}(\hat{v}_1) \\ \gamma = \mathsf{some}(\mathsf{guard}, G_1) \quad \sigma' = \mathsf{assert}(\sigma, \gamma) \quad \gamma = \mathsf{ff} \vee \mathsf{assumes}(\sigma') = \mathsf{ff} \vee \mathsf{asserts}(\sigma') = \mathsf{ff}\end{array}}{\langle(x_1\ x_2), \hat{E}, \sigma\rangle \Downarrow \mathsf{Halt}(\sigma')}$$

$$\textsc{Let}\frac{\langle e_1, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma', \hat{v}_1) \quad \langle e_2, \hat{E}[x \mapsto \hat{v}_1], \sigma'\rangle \Downarrow \hat{r}}{\langle(\mathbf{let}\ (x\ e_1)\ e_2), \hat{E}, \sigma\rangle \Downarrow \hat{r}} \quad \textsc{LetHalt}\frac{\langle e_1, \hat{E}, \sigma\rangle \Downarrow \mathsf{Halt}(\sigma')}{\langle(\mathbf{let}\ (x\ e_1)\ e_2), \hat{E}, \sigma\rangle \Downarrow \mathsf{Halt}(\sigma')}$$

$$\textsc{IfTrue}\frac{\langle x, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \hat{v}) \quad \mathsf{truth}(\hat{v}) = \mathsf{tt} \quad \langle e_1, \hat{E}, \sigma\rangle \Downarrow \hat{r}}{\langle(\mathbf{if}\ x\ e_1\ e_2), \hat{E}, \sigma\rangle \Downarrow \hat{r}} \quad \textsc{IfFalse}\frac{\langle x, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \hat{v}) \quad \mathsf{truth}(\hat{v}) = \mathsf{ff} \quad \langle e_2, \hat{E}, \sigma\rangle \Downarrow \hat{r}}{\langle(\mathbf{if}\ x\ e_1\ e_2), \hat{E}, \sigma\rangle \Downarrow \hat{r}}$$

$$\textsc{IfSym}\frac{\begin{array}{c}\langle x, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \hat{v}) \quad \mathsf{truth}(\hat{v}) \neq \mathsf{tt} \wedge \mathsf{truth}(\hat{v}) \neq \mathsf{ff} \\ \langle e_1, \hat{E}, \mathsf{assume}(\sigma, \mathsf{truth}(\hat{v}))\rangle \Downarrow \hat{r}_1 \quad \langle e_2, \hat{E}, \mathsf{assume}(\sigma, \mathsf{not}(\mathsf{truth}(\hat{v})))\rangle \Downarrow \hat{r}_2\end{array}}{\langle(\mathbf{if}\ x\ e_1\ e_2), \hat{E}, \sigma\rangle \Downarrow \mathsf{merge}_{\hat{R}}(\sigma, [\langle \mathsf{truth}(\hat{v}), \hat{r}_1\rangle, \langle \mathsf{not}(\mathsf{truth}(\hat{v})), \hat{r}_2\rangle])}$$

$$\textsc{Error}\frac{}{\langle(\mathbf{error}), \hat{E}, \sigma\rangle \Downarrow \mathsf{Halt}(\mathsf{assert}(\sigma, \mathsf{ff}))} \quad \textsc{Abort}\frac{}{\langle(\mathbf{abort}), \hat{E}, \sigma\rangle \Downarrow \mathsf{Halt}(\mathsf{assume}(\sigma, \mathsf{ff}))}$$

$$\begin{array}{rll}
\mathsf{assume} : \Sigma \to \hat{B} \to \Sigma & \mathsf{assume}(\sigma, \hat{b}) ::= \langle \mathsf{and}(\mathsf{assumes}(\sigma), \mathsf{imp}(\mathsf{asserts}(\sigma), \hat{b})), \mathsf{asserts}(\sigma)\rangle \\
\mathsf{assert} : \Sigma \to \hat{B} \to \Sigma & \mathsf{assert}(\sigma, \hat{b}) ::= \langle \mathsf{assumes}(\sigma), \mathsf{and}(\mathsf{asserts}(\sigma), \mathsf{imp}(\mathsf{assumes}(\sigma), \hat{b}))\rangle \\
\mathsf{merge}_{\hat{R}} : \Sigma \to \mathbb{G}_{\hat{R}} \to \hat{R} & \mathsf{merge}_{\hat{R}}(\sigma, G) ::= \mathbf{if}\ \mathbf{andmap}(\lambda g.\, \mathsf{choice}(g) \in \mathsf{Halt}(\cdot), G) & \\
 & \quad \mathbf{then}\ \mathsf{Halt}(\mathsf{merge}_{\Sigma}(\sigma, G)) & \\
 & \quad \mathbf{else}\ \mathsf{Result}(\mathsf{merge}_{\Sigma}(\sigma, G), \mathsf{merge}_{\hat{V}}(G)) \\
\mathsf{merge}_{\Sigma} : \Sigma \to \mathbb{G}_{\hat{R}} \to \Sigma & \mathsf{merge}_{\Sigma}(\sigma, G) ::= \langle \mathsf{and}(\mathsf{assumes}(\sigma), \mathsf{all}(\lambda\langle \hat{b}, \hat{r}\rangle.\, \mathsf{imp}(\hat{b}, \mathsf{assumes}(\mathsf{state}(\hat{r})))), G)), & \\
 & \quad \mathsf{and}(\mathsf{asserts}(\sigma), \mathsf{all}(\lambda\langle \hat{b}, \hat{r}\rangle.\, \mathsf{imp}(\hat{b}, \mathsf{asserts}(\mathsf{state}(\hat{r})))), G))\rangle \\
\mathsf{merge}_{\hat{V}} : \mathbb{G}_{\hat{R}} \to \hat{V} & \mathsf{merge}_{\hat{V}}(G) ::= \mathsf{merge}(\mathbf{map}(\lambda\langle \hat{b}, \hat{r}\rangle.\, \langle \hat{b}, \mathsf{value}(\hat{r})\rangle, G)) \\
\mathsf{strengthen} : \Sigma \to \hat{R} \to \hat{R} & \mathsf{strengthen}(\sigma, \mathsf{Halt}(\sigma')) ::= \mathsf{Halt}(\mathsf{compose}(\sigma, \sigma')) & \\
 & \mathsf{strengthen}(\sigma, \mathsf{Out}(\sigma', \hat{v})) ::= \mathsf{Result}(\mathsf{compose}(\sigma, \sigma'), \hat{v}) \\
\mathsf{compose} : \Sigma \to \Sigma \to \Sigma & \mathsf{compose}(\sigma, \sigma') ::= \langle \mathsf{and}(\mathsf{assumes}(\sigma), \mathsf{imp}(\mathsf{asserts}(\sigma), \mathsf{assumes}(\sigma'))), & \\
 & \quad \mathsf{and}(\mathsf{asserts}(\sigma), \mathsf{imp}(\mathsf{assumes}(\sigma), \mathsf{asserts}(\sigma')))\rangle \\
\mathsf{some} : (\alpha \to \hat{B}) \to \alpha^* \to \hat{B} & \mathsf{some}(f, A) ::= \mathbf{foldr}(\lambda a.\hat{b}.\mathsf{or}(f(a), \hat{b}), \mathsf{ff}, A) \\
\mathsf{all} : (\alpha \to \hat{B}) \to \alpha^* \to \hat{B} & \mathsf{all}(f, A) ::= \mathbf{foldr}(\lambda a.\hat{b}.\mathsf{and}(f(a), \hat{b}), \mathsf{tt}, A)
\end{array}$$

Figure 2.6: Symbolic semantics $\mathcal{S}_c$ for λ_c, parameterized by the symbolic factory types and functions (Figure 2.4), as well as the parameters O, D, and op from the syntax and concrete semantics of λ_c (Figure 2.2, 2.3).

Lifting constants and variables. The rules for evaluating constants (Literal, Closure) and variables (Variable) are straightforward. Each lifts the concrete evaluation rule of the same name (Figure 2.3), by replacing all the concrete constructs with their symbolic counterparts. For example, the symbolic Variable rule looks up the variable x in the symbolic environment $\hat{E}$, just as the concrete Variable rule looks up x in the concrete environment E. Similarly, the symbolic Literal and Closure rules return the result of lifting a given literal and closure, respectively, via $\mathsf{lift}(i)$.

Lifting error and abort expressions. The rules Error and Abort are more interesting. Each produces a result of the form $\mathsf{Halt}(\sigma')$, since evaluating (**error**) or (**abort**) always leads to abnormal termination. The two rules differ in how they compute the output state σ': Error uses $\mathsf{assert}(\sigma, \mathsf{ff})$ and Abort uses $\mathsf{assume}(\sigma, \mathsf{ff})$. The resulting output states update one component of the input state—asserts or assumes, respectively—and leave the other component unchanged. These updates are symmetric, which is a key feature of $\mathcal{S}_c$ discussed in Section 2.4.3. For now, we note that $\mathsf{errors}_m(\mathsf{assert}(\sigma, \mathsf{ff}))$ and $\mathsf{aborts}_m(\mathsf{assume}(\sigma, \mathsf{ff}))$ if σ is normal under m, and the two states are equivalent ($\equiv_m$) to σ otherwise. In other words, errors are treated as failed assertions; aborts are treated as failed assumptions; and both are no-ops when the input state is already abnormal.

Example 2.4 *To illustrate the Error rule (and, by symmetry, the Abort rule), suppose that we have* $\langle(\textbf{error}), \hat{E}, \sigma\rangle \Downarrow \mathsf{Halt}(\sigma')$*, where* σ' *stands for* $\mathsf{assert}(\sigma, \mathsf{ff})$*. If the state* σ *is normal under a model* m*, then we have:*

$$
\begin{aligned}
\mathsf{errors}_m(\sigma') &= [\![\mathsf{assumes}(\sigma')]\!]_m \wedge \neg[\![\mathsf{asserts}(\sigma')]\!]_m \\
&= [\![\mathsf{assumes}(\sigma)]\!]_m \wedge \neg[\![\mathsf{and}(\mathsf{assumes}(\sigma), \mathsf{imp}(\mathsf{assumes}(\sigma), \mathsf{ff}))]\!]_m \\
&= [\![\mathsf{assumes}(\sigma)]\!]_m \wedge \neg([\![\mathsf{asserts}(\sigma)]\!]_m \wedge ([\![\mathsf{assumes}(\sigma)]\!]_m \rightarrow [\![\mathsf{ff}]\!]_m)) \\
&= \#t \wedge \neg(\#t \wedge (\#t \rightarrow \#f)) \\
&= \#t
\end{aligned}
$$

In this case, $\mathsf{Halt}(\sigma')$ *evaluates to* Err *under m, as expected. Similar reasoning shows that if* σ *is abnormal under m, then* $\sigma' \equiv_m \sigma$*, so* $\mathsf{Halt}(\sigma')$ *evaluates to* Err *or* Abt*, depending on* σ'*, and the* ERROR *rule propagates the existing cause of abnormal termination.*

Lifting procedure calls. The rules CALL and CALLBAD lift the semantics of procedure calls $(x_1\ x_2)$ as follows. Given $\langle x_1, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \hat{v}_1)$ and $\langle x_2, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \hat{v}_2)$, both rules compute $\mathsf{cast}(\hat{v}_1)$ to extract all possible symbolic closures G_1 from $\hat{v}_1$. Then, they use G_1 to construct the formula $\gamma = \mathsf{some}(\mathsf{guard}, G_1)$, which is true when some guard in G_1 is true, and the state $\sigma' = \mathsf{assert}(\sigma, \gamma)$, which asserts that γ holds. If the factory reduces either γ or a component of σ' to ff, we know that $\hat{v}_1$ is not a closure or σ' is not normal under any model. In this case, CALLBAD triggers and produces $\mathsf{Halt}(\sigma')$. Otherwise, CALL establishes that, for every choice $\langle \hat{b}_i, \hat{c}_i\rangle \in G_1$, applying the closure $\hat{c}_i$ to the value $\hat{v}_2$ in the state $\mathsf{assume}(\sigma', \hat{b}_i)$ produces the result $\hat{r}_i$. The guarded results $\langle \hat{b}_i, \hat{r}_i\rangle$ form the list G_2 of all possible outcomes of applying $\hat{v}_1$ to $\hat{v}_2$. CALL merges these outcomes into the result $\mathsf{merge}_{\hat{R}}(\sigma', G_2)$, which evaluates to $[\![G_2]\!]_m$ if σ' is normal under m, and to $[\![\mathsf{Halt}(\sigma')]\!]_m$ otherwise. In a nutshell, CALL applies the closures from $\mathsf{cast}(\hat{v}_1)$ separately to $\hat{v}_2$ and merges the results, and CALLBAD short-circuits this process when the factory is able to determine that $\mathsf{cast}(\hat{v}_1)$ can never succeed.

Example 2.5 *Consider evaluating the procedure call* $(x_1\ x_2)$ *in the environment* $\hat{E} = \{x_1 \mapsto \hat{v}_1, x_2 \mapsto \hat{v}_2\}$ *and state* $\sigma = \langle \mathsf{tt}, \mathsf{tt}\rangle$*, using the toy symbolic factory from Example 2.3. In this setting, we have* $\langle x_1, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \hat{v}_1)$ *and* $\langle x_2, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \hat{v}_2)$*, and we illustrate the two call rules as follows.*

First, suppose that $\hat{v}_1 = 42$*. Using the definitions from Figure 2.5, we see that* $G_1 = \mathsf{cast}(\hat{v}_1) = [\,]$*,* $\gamma = \mathsf{some}(\mathsf{guard}, G_1) = \mathsf{ff}$*, and* $\sigma' = \mathsf{assert}(\sigma, \gamma) = \mathsf{assert}(\sigma, \mathsf{ff}) = \langle \mathsf{tt}, \mathsf{ff}\rangle$*. These preconditions trigger CALLBAD to return* $\mathsf{Halt}(\sigma')$*, which evaluates to* Err *under all models.*

Second, suppose that $\hat{v}_1 = \phi(\hat{b}, \hat{c}_1, \hat{c}_2)$*, where* $\hat{b} = (z < 0)$*,* $\hat{c}_1 = \langle \hat{\mathbf{cl}}\ x, 0, \hat{E}_1\rangle$*, and* $\hat{c}_2 = \langle \hat{\mathbf{cl}}\ x, 1, \hat{E}_2\rangle$*. We now have* $G_1 = \mathsf{cast}(\hat{v}_1) = [\langle \hat{b}, \hat{c}_1\rangle, \langle !\ \hat{b}, \hat{c}_2\rangle]$*,* $\gamma = !((!(!\ \hat{b}))\ \&\&\ (!\ \hat{b}))$*, and* $\sigma' = \mathsf{assert}(\sigma, \gamma) =$

$\langle \mathsf{tt}, \gamma \rangle$. *This triggers the* Call *rule to establish that* $G_2 = [\langle \hat{b}, \mathsf{Out}(\sigma_1, 0)\rangle, \langle !\ \hat{b}, \mathsf{Out}(\sigma_2, 1)\rangle]$, *where* $\sigma_1 = \mathsf{assume}(\sigma', \hat{b}) = \langle !(\gamma \mathbin{\&\&} !\ \hat{b}), \gamma\rangle$ *and* $\sigma_2 = \mathsf{assume}(\sigma', !\ \hat{b}) = \langle !(\gamma \mathbin{\&\&} !\ !\ \hat{b}), \gamma\rangle$. *Finally, we have* $\mathsf{merge}_{\hat{R}}(\sigma', G_2) = \mathsf{Out}(\mathsf{merge}_{\Sigma}(\sigma', G_2), \mathsf{merge}_{\hat{V}}(G_2)) = \mathsf{Out}(\langle \hat{b}_1, \hat{b}_2\rangle, \phi(\hat{b}, 0, 1))$, *where* $\hat{b}_1 = !(!\ \hat{b} \mathbin{\&\&} !\ !(\gamma \mathbin{\&\&} !\ !\ \hat{b})) \mathbin{\&\&} !(\hat{b} \mathbin{\&\&} !\ !(\gamma \mathbin{\&\&} !\ \hat{b}))$ *and* $\hat{b}_2 = \gamma \mathbin{\&\&} (!(!\ \hat{b} \mathbin{\&\&} !\ \gamma) \mathbin{\&\&} !(\hat{b} \mathbin{\&\&} !\ \gamma))$. *Applying basic logical simplifications to* γ, $\hat{b}_1$, *and* $\hat{b}_2$, *we see that they are all equivalent to* tt. *So, the result of the call is* $\mathsf{Out}(\langle \mathsf{tt}, \mathsf{tt}\rangle, \phi(\hat{b}, 0, 1))$, *which matches the interpretation of* G_2 *under all models.*

Lifting let expressions and conditionals. The rules Let, LetHalt, IfTrue, IfFalse, and IfSym are analogous to Call and CallBad. In particular, Let and IfSym provide a general mechanism for lifting let expressions and conditionals, and the remaining rules short-circuit this mechanism in important special cases. LetHalt ensures that the expression (**let** $(x\ e_1)\ e_2$) halts for the same reason as e_1 when e_1 is guaranteed to halt under all models. IfTrue and IfFalse avoid executing infeasible branches of a conditional when the conditional test is a logical constant and therefore has the same value under all models. All three of these special-case rules mirror the corresponding concrete evaluation rules in Figure 2.3. The two general rules, Let and IfSym, behave similarly to Call.

Lifting operator calls. The rule CallOp lifts the semantics of operator calls using the factory function $\hat{op}$ and the auxiliary function strengthen. The function $\mathsf{strengthen}(\sigma, \hat{r})$ updates the result $\hat{r}$ of $\hat{op}$ so that it evaluates to $[\![\hat{r}]\!]_m$ when σ is normal under the model m and to $[\![\mathsf{Halt}(\sigma)]\!]_m$ otherwise. In essence, $\hat{op}$ calculates its result assuming an unconstrained start state, i.e., $\langle \mathsf{tt}, \mathsf{tt}\rangle$, and CallOp strengthens this result to reflect the constraints imposed by the input state σ.

Example 2.6 *Suppose that we want to evaluate* $(-\ x)$ *in the environment* $\hat{E} = \{x \mapsto 0\}$ *and state* σ, *using the toy factory from Example 2.3. In this case, we have* $\hat{r} = \hat{op}(-, 0) = \mathsf{Out}(\langle \#t, \#t\rangle, 0)$. *Assuming that neither component of* σ *is constant, we have* $\mathsf{strengthen}(\sigma, \hat{r}) = \mathsf{Result}(\mathsf{compose}(\sigma, \langle \#t, \#t\rangle), 0)$

$= \mathsf{Result}(\langle \mathsf{and}(\mathsf{assumes}(\sigma), \mathsf{imp}(\mathsf{asserts}(\sigma), \#t)), \mathsf{and}(\mathsf{asserts}(\sigma), \mathsf{imp}(\mathsf{assumes}(\sigma), \#t))\rangle, 0)$, *simplified to* $\mathsf{Out}(\langle \mathsf{assumes}(\sigma), \mathsf{asserts}(\sigma)\rangle, 0) = \mathsf{Out}(\sigma, 0)$. *This result matches* $[\![\hat{r}]\!]_m$ *when* σ *is normal under* m*, and it evaluates to* Err *or* Abt *otherwise, depending on* σ.

2.4.3 Properties of $\mathcal{S}_c$

The evaluation rules for $\mathcal{S}_c$ are designed to preserve three key properties: *legality*, *reducibility*, and *determinism*. This last property is general and shared by other approaches. The first two are inherent in symbolic execution but missing from prior merging semantics [9]. All three are important guarantees for tools built on top of reusable symbolic evaluators.

Legality Legality is a property of symbolic states that gives client tools a simple interpretation of what these states mean. In particular, the two bits of state distinguish normal termination, where both bits are true, from errors and aborts, where one bit is true and the other is false. A state with two false bits has no natural meaning, so a state is legal under a model m if at least one of its components, $\mathsf{assumes}(\sigma)$ or $\mathsf{asserts}(\sigma)$, is true under m (Figure 2.4). Our semantics supports this intuitive interpretation by ensuring that every legal state leads to a legal result.

Theorem 2.2 *Evaluating an expression from a legal symbolic state leads to a legal symbolic result:* $\forall e, \hat{E}, \sigma, \hat{r}, m.\ (\mathsf{legal}_m(\sigma) \wedge \langle e, \hat{E}, \sigma\rangle \Downarrow \hat{r}) \longrightarrow \mathsf{legal}_m(\hat{r})$.

Preserving legality amounts to guaranteeing that each state σ is free of errors when $\mathsf{assumes}(\sigma) \longrightarrow \mathsf{asserts}(\sigma)$ holds under a given model. Symbolic execution maintains this relation by calling the solver to check the symbolic state after each update, and proceeding only if the state remains feasible and free of errors. Both symbolic execution and the classic merging semantics update the symbolic state in the same way: $\mathsf{assert}(\sigma, \hat{b})$ from Figure 2.6 updates the assertions, and

assume$'(\sigma, \hat{b}) ::= \langle \mathsf{and}(\mathsf{assumes}(\sigma), \hat{b}), \mathsf{asserts}(\sigma) \rangle$ updates the assumptions. But the classic semantics does not check the updated states for errors and feasibility, and without these checks, the updates can produce illegal results. Our semantics uses a different updating function, $\mathsf{assume}(\sigma, \hat{b})$, which preserves legality by construction, as illustrated in Example 2.7.

Example 2.7 *To illustrate the difference between $\mathcal{S}_c$ and the classic merging semantics, consider evaluating the following program in the environment $\hat{E} = \{x \mapsto \hat{v}_1, y \mapsto \hat{v}_2\}$ and state $\sigma = \langle \mathsf{tt}, \mathsf{tt} \rangle$:*

```
1    (let (_ (if x #t (error))) ; assert truth(x)
2    (let (_ (if y #t (abort))) ; assume truth(y)
3      _))
```

Given these inputs, $\mathcal{S}_c$ produces a result that is equivalent to $\hat{r} = \mathsf{Out}(\mathsf{assume}(\mathsf{assert}(\sigma, \hat{b}_1), \hat{b}_2), \cdot)$, and the classic semantics produces one equivalent to $\hat{r}' = \mathsf{Out}(\mathsf{assume}'(\mathsf{assert}(\sigma, \hat{b}_1), \hat{b}_2), \cdot)$, where $\hat{b}_1 = \mathsf{truth}(\hat{v}_1)$ and $\hat{b}_2 = \mathsf{truth}(\hat{v}_2)$. After simplifying the resulting states, we get $\mathsf{state}(\hat{r}) = \langle \mathsf{imp}(\hat{b}_1, \hat{b}_2), \hat{b}_1 \rangle$ and $\mathsf{state}(\hat{r}') = \langle \hat{b}_2, \hat{b}_1 \rangle$. The former is legal under all models, and it is free of errors when $[\![\mathsf{assumes}(\mathsf{state}(\hat{r})) \longrightarrow \mathsf{asserts}(\mathsf{state}(\hat{r}))]\!]_m = [\![\hat{b}_1]\!]_m$ holds. The latter is illegal when $\hat{b}_1$ and $\hat{b}_2$ are both false, and tools must account for this when formulating verification queries.

Reducibility Reducibility is a property of symbolic evaluation that lets client tools treat the symbolic evaluator as a generalized concrete interpreter. Informally, a reducible symbolic evaluator behaves like the underlying concrete interpreter when applied to a (lifted) concrete environment. Reducibility is crucial for testing of client code (see, e.g., [60]), and for ensuring that symbolic evaluation abandons infeasible paths as soon as possible. It is baked into symbolic execution, which reduces to evaluating the same program path as concrete execution in a concrete environment. In contrast, the classic merging semantics mirrors the concrete semantics only on loop-free programs—loops can cause it to diverge even when the corresponding concrete execution terminates. Our symbolic semantics reduces to the concrete semantics on all programs (Theorem 2.3), when coupled with a symbolic factory that takes (lifted) concrete inputs to (lifted) concrete outputs

(Definition 2.5). This optimization is common to practical factories, as well as the toy one from Example 2.3. Unlike the classic merging semantics, $\mathcal{S}_c$ takes advantage of this optimization to abandon halted paths and avoid infeasible infinite loops whenever possible. The key is to introduce the notion of halted results, $\mathsf{Halt}(\sigma)$, and the rules for propagating them, as shown in Example 2.8.

Definition 2.5 (Reducing factory) *A symbolic factory is a* reducing factory *if and only if it satisfies the factory specification (Figure 2.4) and takes lifted concrete inputs to lifted concrete outputs:*

$$\mathsf{truth}(\mathsf{lift}_V(\#f)) = \mathsf{ff}$$
$$\forall v.\ v \neq \#f \longrightarrow \mathsf{truth}(\mathsf{lift}_V(v)) = \mathsf{tt}$$
$$\forall v.\ \mathsf{merge}([\langle \mathsf{tt}, \mathsf{lift}_V(v)\rangle]) = \mathsf{lift}_V(v)$$
$$\forall x, e, E.\ \mathsf{cast}(\mathsf{lift}_V(\langle \mathbf{cl}\ x, e, E\rangle)) = [\langle \mathsf{tt}, \langle \hat{\mathbf{cl}}\ x, e, \mathsf{lift}_{\mathbb{E}}(E)\rangle\rangle]$$
$$\forall v.\ v \notin C \longrightarrow \mathsf{cast}(\mathsf{lift}_V(v)) = []$$
$$\forall v_1, \ldots, v_n.\ \hat{op}(o, \mathsf{lift}_V(v_1), \ldots, \mathsf{lift}_V(v_n)) = \mathsf{lift}_R(op(o, v_1, \ldots, v_n))$$
$$\forall b.\ \mathsf{not}(b) = \mathsf{lift}_B(\neg b)$$
$$\forall b_1, b_2.\ \mathsf{and}(b_1, b_2) = \mathsf{lift}_B(b_1 \wedge b_2)$$
$$\forall b_1, b_2.\ \mathsf{or}(b_1, b_2) = \mathsf{lift}_B(b_1 \vee b_2)$$
$$\forall b_1, b_2.\ \mathsf{imp}(b_1, b_2) = \mathsf{lift}_B(b_1 \longrightarrow b_2)$$

Here, the lifting functions generalize lift *as follows:*

$$\mathsf{lift}_V(b) ::= \mathsf{lift}(b) \qquad \mathsf{lift}_V(d) ::= \mathsf{lift}(d) \qquad \mathsf{lift}_V(\langle \mathbf{cl}\ x, e, E\rangle) ::= \mathsf{lift}(\langle \hat{\mathbf{cl}}\ x, e, \mathsf{lift}_{\mathbb{E}}(E)\rangle)$$
$$\mathsf{lift}_B(\#t) ::= \mathsf{tt} \qquad \mathsf{lift}_B(\#f) ::= \mathsf{ff} \qquad \mathsf{lift}_{\mathbb{E}}(E) ::= \{x \mapsto \mathsf{lift}_V(E[x]) \mid x \in \mathsf{dom}(E)\}$$
$$\mathsf{lift}_R(\mathsf{Err}) ::= \mathsf{Halt}(\langle \mathsf{tt}, \mathsf{ff}\rangle) \qquad \mathsf{lift}_R(\mathsf{Abt}) ::= \mathsf{Halt}(\langle \mathsf{ff}, \mathsf{tt}\rangle) \qquad \mathsf{lift}_R(\mathsf{Ans}(v)) ::= \mathsf{Out}(\langle \mathsf{tt}, \mathsf{tt}\rangle, \mathsf{lift}_V(v))$$

Theorem 2.3 *If $\mathcal{S}_c$ is parameterized with a reducing symbolic factory, then it reduces to the concrete semantics of λ_c in all lifted concrete environments: $\forall e, E, r.\ \langle e, \mathsf{lift}_{\mathbb{E}}(E), \langle \mathsf{tt}, \mathsf{tt}\rangle\rangle \Downarrow \mathsf{lift}_R(r) \leftrightarrow \langle e, E\rangle \Downarrow r$.*

Example 2.8 *Consider evaluating the following program in the environment $\hat{E} = \{x_1 \mapsto \mathsf{lift}_V(\#f), x_2 \mapsto \mathsf{lift}_V(\#f)\}$ and state $\sigma = \langle \mathsf{tt}, \mathsf{tt}\rangle$, using a reducing factory such as the toy factory from Example 2.3:*

```
1    (let (x2 (x1 x2))
2    (let (y (λy . (y y)))
3      (y y)))
```

From Definition 2.5, we see that $\mathsf{cast}(\mathsf{lift}_V(\#f)) = []$ *so* $\mathsf{some}(\mathsf{guard}, []) = \mathsf{ff}$*, triggering* CallBad *to return* $\mathsf{Halt}(\sigma') = \mathsf{Halt}(\mathsf{assert}(\sigma, \mathsf{ff})) = \mathsf{Halt}(\langle \mathsf{tt}, \mathsf{ff} \rangle)$*. This result then triggers* LetHalt *to return* $\mathsf{Halt}(\langle \mathsf{tt}, \mathsf{ff} \rangle) = \mathsf{lift}_R(\mathsf{Err})$ *as the output of the program, matching its concrete execution in the environment* $E = \{x_1 \mapsto \#f, x_2 \mapsto \#f\}$*. Now consider evaluating the same program with the classic merging semantics, which has no notion of halted results or rules for handling them. Without this mechanism,* Call *would return* $\mathsf{Out}(\mathsf{assert}(\sigma, \mathsf{ff}), \cdot)$ *and trigger* Let*, leading to an infinite loop.*

Determinism In addition to preserving legality and reducibility, $\mathcal{S}_c$ is also deterministic (Theorem 2.4): it always produces the same result when applied to the same environment and state. This is important for the development and debugging of client tools, as well as for their usability. Assuming that the underlying solver is deterministic too, a client query is guaranteed to behave consistently across runs, by consuming the same amount of resources to produce the same output.

Theorem 2.4 *Evaluating an expression from the same symbolic environment and state produces the same symbolic result:* $\forall e, \hat{E}, \sigma, \hat{r}_1, \hat{r}_2.\ (\langle e, \hat{E}, \sigma \rangle \Downarrow \hat{r}_1 \wedge \langle e, \hat{E}, \sigma \rangle \Downarrow \hat{r}_2) \longrightarrow \hat{r}_1 = \hat{r}_2$.

2.4.4 Angelic Execution and Verification with $\mathcal{S}_c$

Given an implementation of $\mathcal{S}_c$ and a solver for formulas $\hat{b} \in \hat{B}$ (Definition 2.6), we can implement the angelic execution and verification queries as shown in Definition 2.7. Both queries use $\mathcal{S}_c$ to evaluate the input program e in the symbolic environment $\hat{E}$ that represents a set of concrete environments $\mathcal{E} = \{E \mid \exists m.\ [\![\hat{E}]\!]_m = E\}$. Angelic execution then uses the solver to search for a model in which the resulting symbolic state is normal, and verification searches for a model in which the resulting state errors. The next section shows that this correctly implements Definitions 2.3 and 2.4, respectively.

Definition 2.6 (Solver) *Given a symbolic boolean $\hat{b} \in \hat{B}$, a solver* $\mathsf{solve}(\hat{b})$ *diverges or produces one of two results: either a model $m \in M$ such that $[\![\hat{b}]\!]_m$ is true, or* **unsat** *if no such model exists.*

Definition 2.7 (Queries) *Let $\hat{E}$ be a symbolic environment that represents a set of concrete states $\mathcal{E} = \{E \mid \exists m.\ [\![\hat{E}]\!]_m = E\}$. We define* $\mathsf{guess}(e, \hat{E})$ *and* $\mathsf{verify}(e, \hat{E})$ *as follows:*

$$\mathsf{guess}(e, \hat{E}) ::= \textbf{let } \langle e, \hat{E}, \langle \mathsf{tt}, \mathsf{tt} \rangle \rangle \Downarrow \hat{r}, \sigma = \mathsf{state}(\hat{r}) \textbf{ in } \mathsf{lower}(\hat{E}, \mathsf{solve}(\mathsf{and}(\mathsf{assumes}(\sigma), \mathsf{asserts}(\sigma))))$$

$$\mathsf{verify}(e, \hat{E}) ::= \textbf{let } \langle e, \hat{E}, \langle \mathsf{tt}, \mathsf{tt} \rangle \rangle \Downarrow \hat{r}, \sigma = \mathsf{state}(\hat{r}) \textbf{ in } \mathsf{lower}(\hat{E}, \mathsf{solve}(\mathsf{and}(\mathsf{assumes}(\sigma), \mathsf{not}(\mathsf{asserts}(\sigma)))))$$

$$\mathsf{lower}(\hat{E}, m) ::= [\![\hat{E}]\!]_m$$

$$\mathsf{lower}(\hat{E}, \textbf{unsat}) ::= \textbf{unsat}$$

2.5 Correctness of $\mathcal{S}_c$

This section establishes the correctness of $\mathcal{S}_c$ and the queries implemented on top of it. We formulate and prove the soundness and completeness theorem for $\mathcal{S}_c$. Because $\mathcal{S}_c$ may diverge in the presence of loops, this formulation considers all diverging runs to be trivially correct. We therefore prove an additional theorem showing that $\mathcal{S}_c$ is guaranteed to terminate on all loop-free programs: like the classic merging semantics, $\mathcal{S}_c$ defines a total, sound, and complete symbolic evaluation function for this class of programs. To conclude, we establish that our implementations of $\mathsf{guess}(e, \hat{E})$ and $\mathsf{verify}(e, \hat{E})$ satisfy the definitions of angelic execution and verification given in Section 2.3.

2.5.1 Soundness and Completeness of $\mathcal{S}_c$

What does it mean for a symbolic semantics to be correct, i.e., sound and complete? In the case of symbolic execution, soundness and completeness can be formulated by relating paths in the symbolic execution tree to concrete execution traces (see, e.g., [56, 41]): soundness means that every

concrete trace of length n is covered by some feasible symbolic path of length n, and completeness means that every feasible symbolic path corresponds to a concrete trace. This formulation works for all programs and all symbolic execution trees. But because it assumes path-based evaluation, it does not apply to merging semantics such as $\mathcal{S}_c$. To reason about symbolic evaluation with merging, we take inspiration from prior work on formalizing static analyzers [49, 48] based on abstract interpretation [28, 29].

We adapt two ideas from this work [49, 48, 28, 29] to our setting. First, we define what it means for a merging semantics to be correct by relating the final result of symbolic evaluation to the final results of concrete evaluation, instead of relating paths in the symbolic evaluation graph to concrete traces. Second, we phrase our notion of correctness so that all diverging runs of the symbolic evaluator are trivially correct. In prior work [49, 48], this phrasing avoids the need to prove that abstract interpretation terminates, which is always possible but may be tedious. In our setting, this phrasing is necessary because symbolic evaluation may not terminate in the presence of loops.

Analogously to prior work, we formulate the soundness and completeness theorem for $\mathcal{S}_c$ (Theorem 2.5) by viewing symbolic environments and results as sets of concrete environments and results. Recall from Section 2.4 that the factory interpretation functions $[\![\cdot]\!]^{\hat{\alpha}}$ use models $m \in M$ to relate symbolic objects $\hat{a} \in \hat{\alpha}$ to their concrete counterparts $a = [\![\hat{a}]\!]^{\hat{\alpha}}_m \in \alpha$. So, every symbolic object $\hat{a}$ represents the set of all concrete objects $a \in \alpha$ to which $\hat{a}$ can evaluate via some model. We formalize this relation by writing $a \in \hat{a}$ to denote that $a = [\![\hat{a}]\!]^{\hat{\alpha}}_m$ for some model $m \in M$ (Definition 2.8). Given the relation $\in$, our correctness theorem for $\mathcal{S}_c$ states the following. Let $\hat{E}$ and $\hat{r}$ be a symbolic environment and result such that $\langle e, \hat{E}, \langle \mathsf{tt}, \mathsf{tt} \rangle\rangle \Downarrow \hat{r}$. Then, $\hat{r}$ both overapproximates and underapproximates the set of concrete results that can be reached from $\hat{E}$: if r is the result of a concrete run from an environment $E \in \hat{E}$, then $r \in \hat{r}$; and every $r \in \hat{r}$ can be produced by a concrete run from some environment $E \in \hat{E}$. In other words, $\hat{r}$ precisely captures the set of all concrete results that are reachable from $\hat{E}$.

Definition 2.8 (Concrete membership) *Let* $[\![\cdot]\!]^{\hat{\alpha}} : M \to \hat{\alpha} \to \alpha$ *be an interpretation function provided by a symbolic factory (Figure 2.4). This interpreter defines the* concrete membership relation $\in$ *from* α *to* $\hat{\alpha}$ *as follows:* $\forall a, \hat{a}.\ a \in \hat{a} \leftrightarrow \exists m.\ a = [\![\hat{a}]\!]^{\hat{\alpha}}_m$.

Theorem 2.5 (Soundness and Completeness) *The semantics* $\mathcal{S}_c$ *is* sound *and* complete *with respect to the concrete semantics of* λ_c.

$$\mathcal{S}_c \textit{ SOUNDNESS:} \quad \forall e, \hat{E}, \hat{r}.\ \langle e, \hat{E}, \langle \mathsf{tt}, \mathsf{tt} \rangle\rangle \Downarrow \hat{r} \to \forall E.\ E \in \hat{E} \to \forall r.\ \langle e, E\rangle \Downarrow r \to r \in \hat{r}$$

$$\mathcal{S}_c \textit{ COMPLETENESS:} \quad \forall e, \hat{E}, \hat{r}.\ \langle e, \hat{E}, \langle \mathsf{tt}, \mathsf{tt} \rangle\rangle \Downarrow \hat{r} \to \forall r.\ r \in \hat{r} \to \exists E.\ \langle e, E\rangle \Downarrow r \wedge E \in \hat{E}$$

As noted earlier, our definition of correctness is partial, and this formulation is necessary to account for programs with loops. But solver-aided tools often target *finite* programs, which are free of loops and procedure calls. For this class of programs, $\mathcal{S}_c$ provides the same strong guarantee as symbolic execution and the classic merging semantics: evaluation always terminates (Theorem 2.6) with a sound and complete result (Theorem 2.5).

Theorem 2.6 (Termination on finite programs) *Let* e *be a program that contains no procedure calls, and let* $\hat{E}$ *be a symbolic environment that binds every free variable in* e*. Then, there exists a symbolic result* $\hat{r}$ *such that* $\langle e, \hat{E}, \langle \mathsf{tt}, \mathsf{tt} \rangle\rangle \Downarrow \hat{r}$.

2.5.2 Correctness of Queries Based on $\mathcal{S}_c$

Building on the correctness of $\mathcal{S}_c$, we use Theorems 2.7 and 2.8 to show that our implementations of angelic execution and verification (Definition 2.7) satisfy Definitions 2.3 and 2.4, respectively. Theorem 2.7 establishes that $\mathsf{guess}(e, \hat{E})$ produces a correct output whenever it terminates. If the output of $\mathsf{guess}(e, \hat{E})$ is an environment E, then e executes normally in $E \in \hat{E}$. But if the output is **unsat**, then there is no environment $E \in \hat{E}$ in which e executes normally. Theorem 2.8 is symmetric.

Theorem 2.7 *If* $\mathsf{guess}(e, \hat{E})$ *terminates, then its output is correct according to Definition 2.3.*

$$\textit{SAT:} \quad \forall e, \hat{E}, E.\ \mathsf{guess}(e, \hat{E}) = E \rightarrow (E \in \hat{E} \wedge \mathsf{normal}(e, E))$$

$$\textit{UNSAT:} \quad \forall e, \hat{E}.\ \mathsf{guess}(e, \hat{E}) = \textbf{unsat} \rightarrow \forall E.\ (E \in \hat{E} \rightarrow \neg\mathsf{normal}(e, E))$$

Theorem 2.8 *If* $\mathsf{verify}(e, \hat{E})$ *terminates, then its output is correct according to Definition 2.4.*

$$\textit{SAT:} \quad \forall e, \hat{E}, E.\ \mathsf{verify}(e, \hat{E}) = E \rightarrow (E \in \hat{E} \wedge \mathsf{errors}(e, E))$$

$$\textit{UNSAT:} \quad \forall e, \hat{E}.\ \mathsf{verify}(e, \hat{E}) = \textbf{unsat} \rightarrow \forall E.\ (E \in \hat{E} \rightarrow \neg\mathsf{errors}(e, E))$$

2.6 Implementing $\mathcal{S}_c$: a Case Study of Two Evaluators

To demonstrate the suitability of our framework for developing and validating reusable evaluators, we write and validate two different implementations of $\mathcal{S}_c$. One is a reference evaluator written in Lean, and the other is an optimized evaluator written in Racket. We use Lean to prove that the reference evaluator correctly implements $\mathcal{S}_c$, and we use solver-aided differential testing to validate the optimized evaluator against the reference one. This section presents our implementations, correctness theorems, testing setup, and test results.

2.6.1 Leanette: A Verified Implementation of $\mathcal{S}_c$ in Lean

The reference evaluator, which we call Leanette, is implemented as a generic symbolic interpreter for λ_c. Like $\mathcal{S}_c$, it is parameterized by a symbolic factory and relies on the factory interface to construct and deconstruct symbolic values. Because Lean requires all functions to terminate, Leanette ensures termination in the standard way, by using a *fuel* parameter $n \in \mathbb{N}$ to bound the depth of the recursive call stack. With the addition of fuel, $\textsc{Leanette}(n, e, \hat{E}, \sigma)$ produces either

a symbolic result $\hat{r}$ such that $\langle e, \hat{E}, \sigma\rangle \Downarrow \hat{r}$, or **none**, to indicate that the evaluation is stuck. This implementation is both sound and complete with respect to our symbolic semantics (Theorem 2.9): $\mathcal{S}_c$ admits every result produced by LEANETTE, and, given enough fuel, LEANETTE can produce every result admitted by $\mathcal{S}_c$.

Theorem 2.9 (LEANETTE soundness and completeness) *The LEANETTE symbolic evaluator is* sound *and* complete *with respect to the semantics* $\mathcal{S}_c$.

$$\textit{SOUNDNESS:} \quad \forall n, e, \hat{E}, \sigma, \hat{r}.\ \textit{LEANETTE}(n, e, \hat{E}, \sigma) = \hat{r} \rightarrow \langle e, \hat{E}, \sigma\rangle \Downarrow \hat{r}$$

$$\textit{COMPLETENESS:} \quad \forall e, \hat{E}, \sigma, \hat{r}.\ \langle e, \hat{E}, \sigma\rangle \Downarrow \hat{r} \rightarrow \exists n.\ \textit{LEANETTE}(n, e, \hat{E}, \sigma) = \hat{r}$$

We instantiate LEANETTE with a naïve symbolic factory F_L, which supports integers, lists, and basic operations on these datatypes (`+`, `*`, `=`, `<`, `cons`, `car`, `cdr`, `null?`). The factory supports two types of symbolic variables, booleans and integers, and models $m \in M$ map these variables to concrete values of the right type. Symbolic booleans $\hat{B}$, integers $\hat{D}$, and values $\hat{V}$ are defined similarly to the toy factory in Figure 2.5, with one main exception: we use *symbolic unions* instead of ϕ values to represent the output of conditional evaluation. In our implementation, a symbolic union is a list of guarded values, $[\langle \hat{b}_1, \hat{v}_1\rangle, \ldots, \langle \hat{b}_k, \hat{v}_k\rangle] \in \mathbb{G}_{\hat{V}}$, that are themselves not unions (i.e., $\hat{v}_i \notin \mathbb{G}_{\hat{V}}$). This representation is a simplified version of the symbolic unions used in Rosette [93]. Unlike Rosette's factory, F_L offers no guarantees on the size of the unions or formulas generated during evaluation. In fact, both are worst-case exponential in the size of the (unrolled) input program. But F_L is easier to reason about, and we prove that it satisfies the symbolic factory interface (Theorem 2.10). We also prove that F_L is a reducing factory, so our instantiation of LEANETTE is a sound, complete, and reducible symbolic evaluator for λ_c with integers and lists.

Theorem 2.10 (LEANETTE factory correctness and reducibility) *The LEANETTE factory* F_L

satisfies the symbolic factory interface (Figure 2.4) and the definition of a reducing factory (Definition 2.5).

2.6.2 Rosette 4: An Optimized Implementation of $\mathcal{S}_c$ in Racket

The optimized evaluator, Rosette 4, implements $\mathcal{S}_c$ for the entire Rosette language [93], which is a superset of λ_c. The Rosette language extends Racket [36, 39] with constructs for creating symbolic values, emitting assertions, and formulating solver-aided queries. Rosette's existing reusable evaluator, which we call Rosette 3, is based on the classic merging semantics, and it has been used to develop over 30 solver-aided tools for a wide variety of applications. Rosette 4 replaces the core evaluation rules of Rosette 3 with those of $\mathcal{S}_c$.

Rosette 4 also makes two changes to the Rosette 3 interface, to reflect the switch to $\mathcal{S}_c$. First, it extends the Rosette language to include assumptions, `(assume e)`, which act as syntactic sugar for the (**abort**) expression in λ_c. This change makes assumptions a first-class construct in Rosette 4; they can appear anywhere in a program. In contrast, Rosette 3 has limited support for assumptions via `#:assume e` clauses in verification and synthesis queries. These clauses require client tools to emit all required assumptions upfront, which is not always feasible, leading to complex custom code for precondition tracking (see Section 2.7.1). Second, Rosette 4 exposes a different interface for *symbolic reflection* [93]. Symbolic reflection is a mechanism for allowing client tools to observe the symbolic state during evaluation, and to control the performance of the evaluator using high-level language constructs. In Rosette 3, this interface exposes concepts such as the assertion stack and the path condition, which are part of the classic merging semantics. In Rosette 4, this interface exposes concepts such as the symbolic state with assumptions and assertions, as defined by $\mathcal{S}_c$.

The changes to the evaluator and the interface account for the major differences between the two implementations. At the code level, this amounts to roughly 2,000 line insertions and deletions.

The rest of the code base is shared (roughly 5,000 lines), including the datatypes and procedures for operating on symbolic values. So both implementations share the same symbolic factory.

2.6.3 Validating ROSETTE 4 against LEANETTE with Solver-Aided Differential Testing

To gain confidence that ROSETTE 4 is correct, we perform differential testing [57] against LEANETTE. Our testing setup consists of a *generator* for constructing input programs and environments, and an *oracle* for checking if the two evaluators produce equivalent results on the same input. We use this setup to check that ROSETTE 4 behaves like LEANETTE on a total of 10,000 test programs and environments. The rest of this section describes our test oracle, generator, and results.

Test oracle Given a program e and symbolic environment $\hat{E}$, the oracle compares the outputs produced by ROSETTE 4$(e, \hat{E}, \langle \mathrm{tt}, \mathrm{tt} \rangle)$ and LEANETTE$(n, e, \hat{E}, \langle \mathrm{tt}, \mathrm{tt} \rangle)$, within a timeout of 40 seconds and a fuel limit of $n = 100$. This comparison succeeds only when both outputs are symbolic results, and these results are equivalent under every model, i.e., ROSETTE 4$(e, \hat{E}, \langle \mathrm{tt}, \mathrm{tt} \rangle) = \hat{r}_R$, LEANETTE$(n, e, \hat{E}, \langle \mathrm{tt}, \mathrm{tt} \rangle) = \hat{r}_L$, and $\forall m. [\![\hat{r}_R]\!]^{\hat{R}}_m = [\![\hat{r}_L]\!]^{\hat{R}}_m$. To implement the equivalence check, the oracle imports $\hat{r}_L$ into ROSETTE 4, and uses Z3 [31] to solve the following query: `(verify (assert (equal?` $[\![\hat{r}_R]\!]^{\hat{R}}$ $[\![\hat{r}_L]\!]^{\hat{R}}$`)))`. This query searches for a model m, if any, under which the two symbolic results evaluate to different concrete results according to `equal?`. The function `equal?` considers its inputs to be equivalent if they have the same representation; e.g., two lists are `equal?` if they have the same length and `equal?` elements, and two closures are `equal?` if they have `equal?` lambda terms and environments. This equivalence relation behaves like the equality relation = in Lean, so the oracle reflects the notion of equality used in our Lean formalization.

Test generator Our test generator produces closed programs that couple λ_c expressions with their symbolic environments. Each generated test is a valid Rosette program that consists of a sequence of `(define-symbolic x T)` expressions, followed by an expression e from the λ_c grammar. The definition sequence populates the symbolic environment by binding each free variable in e to a fresh symbolic variable of type boolean or integer. More complex values can then be constructed by the body e from these primitives. The tests are generated in two steps. First, the generator uses fair enumeration combinators [63] to create a bijection η between the set of natural numbers and the set of all possible test programs. The bijection η is set up so that larger numbers tend to correspond to larger programs. Next, the generator uses η to convert random natural numbers into test programs of varying sizes. In particular, the generator takes two inputs: the number N of tests to generate, and an indirect bound k on the size of the generated tests. Given these inputs, it produces N distinct tests by repeatedly calling $\eta(\mathsf{random}(2^i))$, where i increases linearly from 0 to k, and $\mathsf{random}(2^i)$ generates a natural number from 0 to 2^i uniformly at random. This process can generate any test program given a suitable random seed, N, and k.

Example 2.9 *To illustrate our differential testing setup, suppose that the generator produces the two tests shown in Figure 2.7. Both tests are conditional expressions that branch on the value of the boolean expression $0 * n = 0$, where n is a symbolic integer. Test 2.7a returns `#t` if this expression is true and diverges otherwise. Test 2.7b errors if the expression is true and returns `#t` otherwise. Applying the oracle to these tests, we find that Rosette 4 and Leanette behave differently on 2.7a and equivalently on 2.7b.*

When applied to 2.7a, Rosette 4 produces $\mathsf{Out}(\langle \mathsf{tt}, \mathsf{tt} \rangle, \mathsf{tt})$*, and Leanette runs out of fuel. In fact, Leanette will always run out of fuel on this program because its symbolic factory is unable to reduce $0 * n$ to 0. Rosette 4 is able to perform this reduction, so it terminates with the expected value. Both of these outcomes are correct according to formalization (Section 2.5).*

When applied to 2.7b, Rosette 4 produces $\mathsf{Halt}(\langle \mathsf{tt}, \mathsf{ff} \rangle)$*, and Leanette produces* $\mathsf{Out}(\sigma, \mathsf{tt})$*, where*

```
1 (define-symbolic n integer?) ; Symbolic integer
2
3 (if (zero? (* 0 n)) ; If 0 * n = 0
4     #t              ; then true
5     (let loop ()    ; else loop infinitely
6       (loop)))
```

(a) A program that terminates under ROSETTE 4 but does not terminate under LEANETTE.

```
1 (define-symbolic n integer?) ; Symbolic integer
2
3 (if (zero? (* 0 n)) ; If 0 * n = 0
4     (assert #f)     ; then error
5     #t)             ; else true
```

(b) A program that produces Halt(·) under ROSETTE 4 but produces Out(·, ·) under LEANETTE.

Figure 2.7: Two test programs that combine a λ_c expression and a symbolic environment. The programs are shown in the Rosette syntax for brevity. The test oracle determines that ROSETTE 4 and LEANETTE differ on (a) due to non-termination, and that they agree on (b) according to our equivalence relation.

$\sigma = \langle \mathrm{tt}, !(0 = (0 * n)) \rangle$. *These symbolic results have different representations because, once again,* ROSETTE 4 *reduces* $0 * n$ *to* 0 *and* LEANETTE *does not. But they evaluate to the same concrete result under every model* m*, i.e.,* $[\![\mathrm{Halt}(\langle \mathrm{tt}, \mathrm{ff} \rangle)]\!]^{\hat{R}}_m = [\![\mathrm{Out}(\sigma, \mathrm{tt})]\!]^{\hat{R}}_m$*, so the oracle considers them equivalent.*

Test results We validate ROSETTE 4 against LEANETTE on a set T of 10,000 randomly generated test programs. For each program in T, we collect the oracle output and, for each evaluator, the final symbolic state and the running time. Table 2.1 shows the test results, where T is partitioned into buckets by AST size. The largest program in T consists of 158 expressions, and the average program size is 63 expressions. Both LEANETTE and ROSETTE 4 terminate and produce a symbolic result on every program in T, within the oracle timeout and fuel limit.

Using T, our testing setup quickly discovers an intentional semantics discrepancy between LEANETTE and ROSETTE 4. In LEANETTE, the `cons` operator creates only lists. It takes as input a value v_0 and a list $[v_1, \ldots, v_k]$, and returns the list $[v_0, v_1, \ldots, v_k]$. In ROSETTE 4, `cons` takes as input any two values and returns a pair, which represents a list when the second argument is list. After we adjust the semantics of `cons` in ROSETTE 4 to match that of LEANETTE, we find that the two evaluators behave equivalently on all programs in T, as shown in the last column of Table 2.1.

Table 2.1: Summary of the testing results, which are bucketed by AST size. Under "Leanette state", the "Sym" column refers to the number of non-trivial resulting symbolic states in Leanette evaluation. The "Max" and "Avg" columns are the maximum and average resulting symbolic state size *among the non-trivial ones*. Columns under "Rosette 4 state" have the same meanings. "Validated?" column is the oracle output, where "✓" means the evaluation results from both evaluators agree on all programs in a bucket.

Bucket (by AST size)			Leanette state			Rosette 4 state			Leanette time		Rosette 4 time		
Range	Count	Avg	Sym	Max	Avg	Sym	Max	Avg	Max (s)	Avg (s)	Max (s)	Avg (s)	Validated?
1 - 12	1,006	6	220	635	41	88	20	4	2.4	2.2	0.3	0.1	✓
13 - 24	1,021	19	349	1,142	77	174	36	6	2.3	2.2	0.2	0.1	✓
25 - 38	1,042	32	430	2,855	102	224	34	6	2.3	2.2	0.8	0.1	✓
39 - 50	1,036	44	433	16,979	242	187	34	7	2.8	2.2	0.2	0.1	✓
51 - 65	1,040	58	450	7,523	186	227	30	8	2.4	2.2	0.2	0.1	✓
66 - 77	1,040	72	459	30,386	474	239	45	8	2.8	2.2	0.2	0.1	✓
78 - 88	1,053	83	454	19,835	426	231	36	8	2.5	2.2	0.2	0.1	✓
89 - 102	1,042	95	427	45,893	482	190	41	9	3.5	2.2	0.2	0.1	✓
103 - 119	1,009	111	425	61,439	686	218	45	9	4.1	2.2	1.0	0.1	✓
120 - 158	711	129	324	293,171	2,001	159	36	9	36.6	2.2	0.3	0.1	✓

Table 2.1 also shows two sets of metrics that characterize the performance of these evaluators. The "time" columns display the maximum and average evaluation time for all programs in a bucket. The "state" columns show the number of non-trivial symbolic states produced by the evaluator, as well as the maximum and average size of the non-trivial states in each bucket. A state is trivial iff both of its components are constant (ff or tt). The size of a state is the number of nodes in the graph representation of its components. In particular, both Leanette and Rosette 4 represent symbolic booleans as abstract syntax graphs. These graphs are trees in Leanette and DAGs in Rosette 4.

The data in Table 2.1 exhibits two notable trends. First, a large fraction of the computed states are trivial: 60% for Leanette and 80% for Rosette 4. This trend is expected for randomly generated programs, which have a high probability of containing trivial errors (e.g., type errors). Second, Rosette 4 is orders-of-magnitude more efficient than Leanette in terms of both running time and state size. This trend is also expected because Rosette 4 uses a heavily optimized symbolic factory, while Leanette uses a naïve one. In the worst case, Leanette generates a

Table 2.2: Coverage results of the randomly generated programs. Each of the injected faults β violates soundness. "✓" indicates that the injected fault is caught by the generated test set T (i.e., ROSETTE 4 and BADLEANETTE(β) disagree on at least one program in T).

Injected mistake (β)	Location	Caught?
Make and$(b_a, b_b) = b_a$ where b_a and b_b are non-literal symbolic booleans	symbolic factory	✓
Make or$(b, b) =$ tt where b is a non-literal symbolic boolean	symbolic factory	✓
Make $\hat{op}(<, z_a, z_b) = (z_b < z_a)$ where either z_a or z_b is a non-literal symbolic integer	symbolic factory	✓
In CALLOP, skip strengthen	symbolic evaluator	✓
In CALL, change the guards for evaluation of the body to be tt	symbolic evaluator	✓
In CALLBAD, use the input symbolic state without asserting γ	symbolic evaluator	✓
In LET, do not bind any variable	symbolic evaluator	✓
In IFSYM, swap not(truth($\hat{v}$)) and truth($\hat{v}$) when calling merge$_{\hat{R}}(\cdot, \cdot)$	symbolic evaluator	✓
In IFTRUE and IFFALSE, swap the conditions	symbolic evaluator	✓
In Err, do ABORT	symbolic evaluator	✓
In ABORT, do Err	symbolic evaluator	✓

symbolic state that is exponential in program size. ROSETTE 4, in contrast, generates polynomially sized states.

Coverage To gain further confidence in our test results, we inject 11 faults into LEANETTE, one at a time, and check that they are uncovered by at least one test in T. For each fault β, we create a new evaluator BADLEANETTE(β) that applies β to LEANETTE, and we test ROSETTE 4 against BADLEANETTE(β) on T. Table 2.2 describes the faults and the results of the differential testing. Three faults are in the symbolic factory, and the rest are in the implementation of the $\mathcal{S}_c$ rules. Our test suite discovers all of them, giving us confidence that ROSETTE 4 correctly implements $\mathcal{S}_c$.

2.7 Utility and Performance of an $\mathcal{S}_c$ Evaluator: Experiments

This section evaluates the utility and performance of ROSETTE 4 by comparing them to those of ROSETTE 3. Our evaluation aims to answer the following research questions:

1. Can the interface provided by an $\mathcal{S}_c$ evaluator simplify the implementation of client tools?

2. Can an $\mathcal{S}_c$ evaluator match the performance of a classic evaluator when used within state-

of-the-art solver-aided tools?

We conduct two sets of experiments and find a positive answer to both questions.

2.7.1 Comparing the Interface of Rosette 4 and Rosette 3

We evaluate the utility of $\mathcal{S}_c$ by porting two sets of benchmarks to Rosette 4, developed in prior work on SymPro [14] and Jitterbug [61]. SymPro is a tool for profiling the performance of symbolic evaluators, and its benchmarks are drawn from the evaluation suites of 15 published verification and synthesis tools. Each SymPro benchmark applies one of these tools to its slowest available input(s). The Jitterbug benchmarks are drawn from the evaluation suite of Jitterbug, a framework for developing and verifying just-in-time (JIT) compilers for the Berkeley Packet Filter (BPF) [40] language in the Linux kernel. Jitterbug performs verification on each BPF opcode individually, which amounts to 668 verification queries across the six architectures supported by Jitterbug (Arm32, Arm64, RV32, RV64, x86-32, and x86-64). All tools in our benchmark sets were originally developed in Rosette 3.

As we saw in Section 2.6.2, Rosette 4 extends the Rosette 3 language to include `assume` expressions, and it exposes different constructs for symbolic reflection. To port our benchmarks to Rosette 4, we adapted their code to use these new constructs and the extended language. We detail the porting effort for the SymPro benchmarks next, and then describe how Rosette 4 enabled us to simplify the implementation of Jitterbug.

Porting SymPro Benchmarks

Table 2.3 summarizes the differences between the ported and the original code for each benchmark. The first four columns report the line count for both implementations, and the number of insertions and deletions by which the ported code differs from the original code. The last column shows which

Table 2.3: Differences between the implementations of the ported and original benchmarks.

Benchmark	ROSETTE 3 LoC	ROSETTE 4 LoC	LoC diff		Solver
Bagpipe [100]	2,643	2,643	+2	-2	Z3
Bonsai [19]	437	437	+1	-1	Boolector
Cosette [23]	774	774	+0	-0	Z3
Ferrite [12]	348	348	+2	-2	Z3
Fluidics [101]	98	98	+0	-0	Z3
GreenThumb [74]	3,554	3,556	+13	-11	Boolector
IFCL [93]	483	483	+13	-13	Boolector
MemSynth [13]	13,303	13,307	+8	-4	Z3
Neutrons [70]	37,174	37,174	+2	-2	Z3
Nonograms [17]	3,368	3,374	+11	-5	Z3
Quivela [1]	1,010	1,009	+10	-11	Z3
RTR [50]	1,649	1,640	+12	-21	CVC4
SynthCL [93]	2,257	2,256	+42	-43	Boolector
Wallingford [15]	2,532	2,533	+12	-11	Z3
WebSynth [93]	1,181	1,189	+14	-6	Z3
Jitterbug [61]	29,280	28,935	+478	-823	Boolector

SMT solver is used for a given benchmark: Z3 v4.8.8 [31], Boolector v3.2.1 [66], or CVC4 v1.8 [3].

As the data in Table 2.3 indicates, porting the SYMPRO benchmarks to ROSETTE 4 required relatively few changes to their code. Most changes were mechanical: we either adapted the code to use the new symbolic reflection constructs, or replaced `#:assume` clauses with equivalent `assume` expressions. One exception is the RTR tool [50], which we simplified and made faster (see Table 2.4a) using `assume` expressions. RTR is a type checker for a refinement type system for Ruby, and it works on an intermediate verification language that includes `assume` statements. Since first-class assumptions are not available in ROSETTE 3, RTR implements `assume` statements as early exits that return a special value. We removed this custom code and implemented the `assume` statement directly using the `assume` expression. It took one author 4 days to port all 15 tools to ROSETTE 4.

Porting JITTERBUG

JITTERBUG is the most complex code base we ported to ROSETTE 4 (see Table 2.3). At the core of JITTERBUG's proof strategy is its *per-instruction correctness* specification: given a source instruction

and a JIT context (e.g., compiler configurations), Jitterbug proves that the execution of the target instructions produced by the JIT exhibits the same behavior as that of the source instruction. To do so, Jitterbug first symbolically evaluates the JIT implementation with a BPF instruction and a JIT context to produce a symbolic representation of all possible sequences of target instructions. Next, it symbolically evaluates both the source instruction and target instructions on source and target states, respectively, to encode their semantics. Finally, it checks that the resulting source and target states are equivalent, assuming that the original source and target states are equivalent. This requires Jitterbug to model the assumptions made by the JIT, such as the well-formedness of BPF instructions and the memory layout of the Linux kernel.

Since Rosette 3 does not support assumptions, Jitterbug implements its own system for tracking assumptions. The system works by escaping to Racket to capture and store the assumptions outside of symbolic evaluation. It then reintroduces them as preconditions in later assertions. This process is subtle and does not preserve legality, requiring careful manual reasoning to ensure that the final verification query, which takes the form $pre \wedge \neg post$, is sound (c.f., Example 2.7).

When porting Jitterbug to Rosette 4, we replaced this workaround with `assume` expressions. This change simplified both the Jitterbug implementation and the formulation of the top-level correctness specification, which no longer has to recover and incorporate the stored assumptions. The legality guarantee provided by Rosette 4 also increased the confidence in the soundness of the verification queries emitted by Jitterbug. The porting process took one author 2 weeks. Our experience shows that the interface exposed by Rosette 4 can simplify the implementation of a complex client tool, and that the developer burden to utilize these new features is low.

Table 2.4: Performance results for the ported and original SYMPRO (2.4a) and JITTERBUG (2.4b) benchmarks, along with statistics (2.4c) summarizing the performance ratios between the ported and original code. In 2.4a and 2.4b, the columns "Total (s)", "Eval. (s)", and "Solving (s)" are the elapsed wall clock time, symbolic evaluation time, and solving time respectively. The column "Terms ($\times 10^3$)" indicates the encoding size. The average ratio in 2.4c is the geometric mean of the per-benchmark ratios for a given performance metric.

(a) Performance results for the 15 ported and original SYMPRO benchmarks.

	ROSETTE 3				ROSETTE 4			
Benchmark	Total (s)	Eval. (s)	Solving (s)	Terms ($\times 10^3$)	Total (s)	Eval. (s)	Solving (s)	Terms ($\times 10^3$)
Bagpipe	23	22	< 1	47	23	23	< 1	48
Bonsai	20	18	2	664	40	37	3	1,222
Cosette	15	7	7	131	15	8	8	154
Ferrite	19	12	7	34	26	12	14	40
Fluidics	19	6	13	284	23	7	17	308
GreenThumb	53	7	46	239	4	2	2	74
IFCL	157	10	147	383	119	10	109	438
MemSynth	20	18	2	61	22	20	2	163
Neutrons	36	36	< 1	444	10	10	< 1	172
Nonograms	9	3	5	51	10	3	7	73
Quivela	31	29	2	1,113	34	31	4	1,340
RTR	329	312	18	741	113	82	32	1,106
SynthCL	258	13	246	726	253	15	238	732
Wallingford	5	< 1	4	4	5	< 1	4	5
WebSynth	10	10	< 1	1,012	16	16	< 1	778

(b) Performance results for the 668 ported and original JITTERBUG benchmarks.

Evaluator	Total (s)			Eval. (s)			Solving (s)			Terms ($\times 10^3$)		
	mean	med.	max	mean	med.	max	mean	med.	max	mean	med.	max
ROSETTE 3	50	22	9,963	2	1	69	48	20	9,894	119	15	1,678
ROSETTE 4	38	20	4,100	1	1	73	36	19	4,027	120	23	1,837

(c) Performance ratios between the ported and original code for SYMPRO and JITTERBUG benchmarks.

Benchmark	Total			Eval.			Solving			Terms		
	best	worst	avg.	best	worst	avg.	best	worst	avg.	best	worst	avg.
SYMPRO	0.07	1.99	0.81	0.26	2.10	0.87	0.04	2.21	0.95	0.31	2.65	1.06
JITTERBUG	0.23	6.03	0.91	0.33	2.18	0.87	0.22	6.48	0.91	0.79	8.12	1.09

2.7.2 Comparing the Performance of Rosette 4 and Rosette 3

To evaluate the performance of $\mathcal{S}_c$, we compare the running time and encoding size of Rosette 4 and Rosette 3 on the ported and original version of each benchmark, respectively. We collected all performance data using Racket v8.1 on an Intel Core i5-6600K at 3.50 GHz with 16 GB of RAM. Table 2.4a shows the results for each SymPro benchmark, and Table 2.4b summarizes the results across all Jitterbug benchmarks. Both figures report the total time, symbolic evaluation time, solving time, and encoding size under Rosette 4 and Rosette 3. The encoding size is given as the number of symbolic terms generated during symbolic evaluation. This number overapproximates the size of the encoding sent to the solver and offers a rough measure of the total amount of work performed by the evaluator. The symbolic evaluation time includes both the time spent to generate the encoding and to pipe it to the solver. The solving time is computed as the difference between the wall clock time and the CPU time consumed by Racket. Finally, Table 2.4c reports the best, worst, and average ratio of the Rosette 4 and Rosette 3 performance across all metrics and benchmarks. For example, comparing the symbolic evaluation time of the ported and original SymPro benchmarks, we find that the ported ones are up to 2.1× slower and up to 3.8× faster than the original ones, with an average speedup of 13%.

The analysis in Table 2.4c shows that, on average, Rosette 4 generates 6–9% more terms than Rosette 3, taking roughly 10–20% less time to do so, and producing an encoding that is about 5–9% faster to solve. To understand the difference in the number of generated terms, recall from Section 2.4.3 that $\mathcal{S}_c$ updates the assumption component of the symbolic state using $\mathsf{assume}(\sigma, \hat{b})$, while the classic merging semantics uses $\mathsf{assume}'(\sigma, \hat{b})$. In the worst case, the symbolic factory for Rosette 4 and Rosette 3 creates 3 terms to represent $\mathsf{assumes}(\mathsf{assume}(\sigma, \hat{b}))$ and only 1 term to represent $\mathsf{assumes}(\mathsf{assume}'(\sigma, \hat{b}))$. Additionally, Rosette 3 uses a simpler function for merging two symbolic states, which generates no new terms, while the $\mathcal{S}_c$ function $\mathsf{merge}_\Sigma(\sigma, G)$

can generate up to $4|G|+2$ terms. These two differences are the main reason why ROSETTE 4 emits up to 8× more terms than ROSETTE 3 in our benchmarks. We avoid this blowup in the average case by specializing ROSETTE 4 with rewriting rules tailored for the terms generated by $\mathsf{assume}(\sigma, \hat{b})$ and $\mathsf{merge}_{\Sigma}(\sigma, G)$.

Given that ROSETTE 4 emits more terms than ROSETTE 3, it may seem surprising that it is, on average, slightly faster than ROSETTE 3, and that the resulting formula is slightly easier to solve. In general, these differences are difficult to fully explain since they depend on many factors, and the behavior of both systems is sensitive to small initial differences, e.g., a factory simplification that triggers in one system but not the other. With this caveat in mind, our experience suggests that the observed difference is due to two factors. First, at every point during and after the evaluation, ROSETTE 4 operates on a more constrained symbolic state than ROSETTE 3. Both of its state components carry terms that are not available in ROSETTE 3 and that may trigger additional simplifications. Second, thanks to legality, ROSETTE 4 emits the query $\mathsf{assumes}(\sigma) \wedge \neg\mathsf{asserts}(\sigma)$ (Definition 2.7), while ROSETTE 3 emits $\neg\mathsf{asserts}(\sigma)$. It is sound for ROSETTE 4 to emit just $\neg\mathsf{asserts}(\sigma)$, but we find that the redundant formula tends to elicit better performance in practice; for example, the JITTERBUG benchmarks run on average 25% slower without the redundant formula. Overall, even though ROSETTE 4 produces a larger encoding than ROSETTE 3, our results show that it matches the runtime performance of ROSETTE 3 in a wide range of tools.

2.8 Conclusion

This chapter presented the first formal framework for reasoning about reusable symbolic evaluators. The framework is based on $\mathcal{S}_c$, a new symbolic semantics with merging. This semantics is defined with respect to the symbolic factory interface, which abstracts away the details of how symbolic values are represented, created, and manipulated. As such, $\mathcal{S}_c$ admits a wide range is implementations.

We use Lean to prove that $\mathcal{S}_c$ is sound and complete with respect to the concrete semantics of its target language, λ_c, which extends Core Scheme with assumptions and assertions. We also prove that $\mathcal{S}_c$ preserves two properties, legality and reducibility, that are key to reusing a symbolic evaluator in a client tool. Leanette and Rosette 4, two implementations of $\mathcal{S}_c$ in Lean and Racket, respectively, show that $\mathcal{S}_c$ provides a general contract for building and validating reusable evaluators. By porting 16 published verification and synthesis tools from Rosette 3 to Rosette 4, we demonstrate that $\mathcal{S}_c$ provides a practical interface for client tools: Rosette 4 both simplifies the implementation of two of the benchmarks and matches the performance of Rosette 3 across these tools. All source code accompanying this work is publicly available at `https://github.com/emina/rosette`.

Chapter 3

Scaling up solver-aided programming

3.1 Introduction

Solver-aided programs or tools based on SMT solving have automated vital programming tasks in many domains, from verifying safety-critical properties of medical software [70] to synthesizing fast computational kernels for cryptographic applications [73]. These tools employ *symbolic evaluation* [51, 9] to reduce the semantics of all paths through a loop-free (i.e., *finite*) program to logical constraints. The resulting constraints are then used to express queries about program behavior as logical satisfiability queries, discharged with an SMT solver. Since the solvability of such queries hinges on the compactness and simplicity of the underlying constraints, effective symbolic evaluation is key to building effective solver-aided tools.

Building a tool used to require crafting a custom symbolic evaluator, a difficult task that can take years of expert work. Today, this burden is much lower thanks to reusable symbolic evaluators (Chapter 2) provided by *solver-aided host languages* [92, 94] and frameworks [16, 79]. To build a tool, developers simply write an interpreter for the tool's source language in the solver-aided host language. When this interpreter executes a source program, the host's symbolic evaluator reduces both

the interpreter and the program to constraints. The interpreter can control its symbolic evaluation, and thus the encoding, through constructs [90, 78] exposed by the host language and through the structure of its implementation [14]. By exploiting these control mechanisms, developers can create, in weeks, state-of-the-art tools [60] that outperform a custom symbolic execution engine [62, 82].

But if an interpreter performs poorly on a host symbolic evaluator, finding and fixing the bottleneck can be daunting. Recent work on *symbolic profiling* [14] explains why: classic performance engineering techniques assume a single path of execution, and the all-path execution model of symbolic evaluation violates this assumption. As a result, standard profiling tools (e.g., time-based) fail to identify the code that needs to be optimized, and standard optimizations (e.g., breaking early out of a loop) can make performance asymptotically worse under symbolic evaluation. Symbolic profiling addresses the first problem, providing a new performance model for symbolic evaluation and an automatic technique for identifying performance bottlenecks in solver-aided code. The second problem, however, remains open, with developers relying on experience and ad-hoc experimentation to optimize their code.

To address this problem, we present a new method for automatic repair of common performance bottlenecks in solver-aided code. The key idea is to formulate the *symbolic performance repair* problem as combinatorial search in a space of semantics-preserving transformations, or *repairs*. Our technique, SymFix, takes as input a solver-aided program and a workload, and it searches the repair space for a semantically equivalent program that minimizes the cost of symbolic evaluation [14] on the input workload. The choice of repairs and the search strategy are critical to the usefulness and completeness of SymFix. This work contributes a small set of generic repairs that combine to fix common bottlenecks, and an effective algorithm for combining repairs into (optimal) fixes.

What makes a generic repair useful for code under symbolic evaluation? Intuitively, a repair is useful if its application reduces the cost of symbolic evaluation for a large class of programs. This cost depends on the program's control structure and the evaluator's strategy for splitting and

merging states [14, 53, 93]. So useful repairs change the program's control structure or evaluation strategy.

Based on this insight, we develop a set of three repairs that employ *deforestation* [95] to simplify program structure and *symbolic reflection* [93] to simplify the evaluation strategy. Deforestation is a classic optimization for functional programming languages that eliminates intermediate lists. Under concrete evaluation, deforestation improves performance by a constant factor. Under symbolic evaluation, however, it can improve performance asymptotically when the intermediate lists are symbolic. We use deforestation based on build/foldr fusion [44] as one of our repairs. We also develop two repairs for host languages that support symbolic reflection—a set of language constructs that a program can use to inspect its symbolic state and control its symbolic evaluation (e.g., by forcing a split on a merged state). These two repairs work by creating more opportunities for concrete evaluation. As such, they can both improve performance asymptotically and, in some cases, fix divergence due to loss of concreteness.

The search space defined by our repairs is finite for every program, so it supports complete and optimal search. But it is also intractably large for real programs. We therefore formulate SymFix as an anytime algorithm, equipped with a pruning mechanism that exploits *precedence* of repairs and a prioritization heuristic that exploits symbolic profiling information. The pruning mechanism is inspired by partial order reduction [26]: if two repairs can always be reordered so that one is applied before the other without changing the result, SymFix explores only one of the orders. The prioritization heuristic uses ranking information computed by symbolic profiling to decide what parts of the program to repair first. In particular, symbolic profiling takes as input a program and a workload, and ranks the locations in the program from most to least likely bottlenecks. SymFix uses this ranking to quickly drive the search toward most promising solutions.

We implement SymFix for Rosette [88, 93, 92], a solver-aided host language that extends Racket [36, 39] with support for symbolic evaluation, reflection, and profiling. To evaluate Sym-

Fix's effectiveness, we apply it to 15 solver-aided tools [1, 12, 13, 15, 17, 19, 23, 50, 74, 70, 93, 100, 101] studied in the paper on symbolic profiling [14], as well as 3 more recent tools [60, 64, 73]. SymFix improves the performance of 12 tools by a factor of 1.1×–91.7×, and 4 of these fixes match or outperform those written by experts. SymFix also finds 5 fixes that were missed by experts. We further show that the improvements made by SymFix generalize to unseen workloads, and that its search strategy is essential for finding useful fixes.

In summary, this chapter makes the following contributions:

1. A formulation of the symbolic performance repair problem as combinatorial search in a space of semantics-preserving transformations, or repairs.

2. SymFix, a new technique for solving this problem. SymFix contributes a set of repairs based on deforestation and symbolic reflection, and an effective anytime algorithm for combining these repairs into useful fixes.

3. An implementation of SymFix for the Rosette solver-aided language [88, 93].

4. An evaluation of SymFix's effectiveness on 18 published solver-aided tools built in Rosette, showing that it can find repairs that outperform expert fixes and that generalize to unseen workloads.

The rest of the chapter illustrates symbolic performance repair on a small example (Section 3.2); formulates the problem of repairing performance bottlenecks in solver-aided code (Section 3.3); presents the SymFix algorithm, repairs, and implementation for Rosette (Section 3.4); shows the effectiveness of SymFix at repairing bottlenecks in real solver-aided tools hosted by Rosette (Section 3.5); discusses related work (Section 3.6); and concludes with a summary of key points (Section 3.7).

```
1  ; cpu state: program counter and registers
2  (struct cpu (pc regs) #:mutable)
3
4  ; interpret a program from a cpu state
5  (define (interpret c program)       ; A
6    (define i (fetch c program))      ; B
7    (match i
8      [(list opcode rd rs imm)
9       (execute c opcode rd rs imm)
10      (when (not (equal? opcode 'ret))
11        (interpret c program))]))
12
13 ; fetch an instruction at the current pc
14 (define (fetch c program)
15   (define pc (cpu-pc c))
16   (vector-ref program pc))          ; C
17
18 ; read register rs
19 (define (cpu-reg c rs)
20   (vector-ref (cpu-regs c) rs))
21
22 ; write value v to register rd
23 (define (set-cpu-reg! c rd v)
24   (vector-set! (cpu-regs c) rd v))
25
26 ; execute instruction (opcode rd rs imm)
27 (define (execute c opcode rd rs imm)
28   (define pc (cpu-pc c))
29   (case opcode
30     [(ret)
31      (set-cpu-pc! c 0)]             ; D
32     [(bnez)
33      (if (not (= (cpu-reg c rs) 0))
34          (set-cpu-pc! c imm)        ; E
35          (set-cpu-pc! c (+ 1 pc)))] ; F
36     [(sgtz)
37      (set-cpu-pc! c (+ 1 pc))
38      (if (> (cpu-reg c rs) 0)
39          (set-cpu-reg! c rd 1)      ; G
40          (set-cpu-reg! c rd 0))]    ; H
41     [(sltz)
42      (set-cpu-pc! c (+ 1 pc))
43      (if (< (cpu-reg c rs) 0)
44          (set-cpu-reg! c rd 1)      ; I
45          (set-cpu-reg! c rd 0))]    ; J
46     [(li)
47      (set-cpu-pc! c (+ 1 pc))       ; K
48      (set-cpu-reg! c rd imm)]))     ; L
49
50 (define sgnt #( ; sign in ToyRISC
51  (sltz 1 0 #f)  ; 0. r1 = r0<0 ? 1 : 0
52  (bnez #f 1 4)  ; 1. branch to 4 if r1!=0
53  (sgtz 0 0 #f)  ; 2. r0 = r0>0 ? 1 : 0
54  (ret #f #f #f) ; 3. return
55  (li 0 #f -1)   ; 4. r0 = -1
56  (ret #f #f #f) ; 5. return
57  ))
58
59 (define-symbolic X Y integer?)
60 (define c (cpu 0 (vector X Y)))
61 (interpret c sgnt)
62 (verify
63   (assert (= (cpu-reg c 0) (sgn X))))
```

Figure 3.1: A ToyRISC interpreter and program in Rosette, adapted from Serval [60].

3.2 Overview

This section illustrates symbolic performance repair on a small solver-aided program (Figure 3.1). The program is adapted from Serval [60], a framework for verifying systems code at the instruction level. Serval is built in Rosette [88], and it supports creating scalable automated verifiers by writing interpreters. Serval's authors show how to profile this program with a symbolic profiler, and manually fix the bottleneck using a custom construct implemented as a Rosette macro. We first revisit this analysis to highlight the challenges of repairing bottlenecks in solver-aided code, and then show how SymFix repairs the problem automatically and generically, using a repair based on symbolic reflection [93].

Solver-aided programming. Figure 3.1 shows a small program [60] written in Rosette, a solver-aided host language that extends Racket [36] with support for symbolic evaluation. Rosette

programs behave like Racket programs when executed on concrete values. But Rosette also *lifts* programs, via symbolic evaluation, to operate on *symbolic values*. These values are used to formulate *solver-aided queries*, such as verifying that a program satisfies its specification, expressed as assertions, on all inputs. The example verifies a program in ToyRISC, a small subset of RISC-V [98], by lifting its interpreter to work on symbolic values.

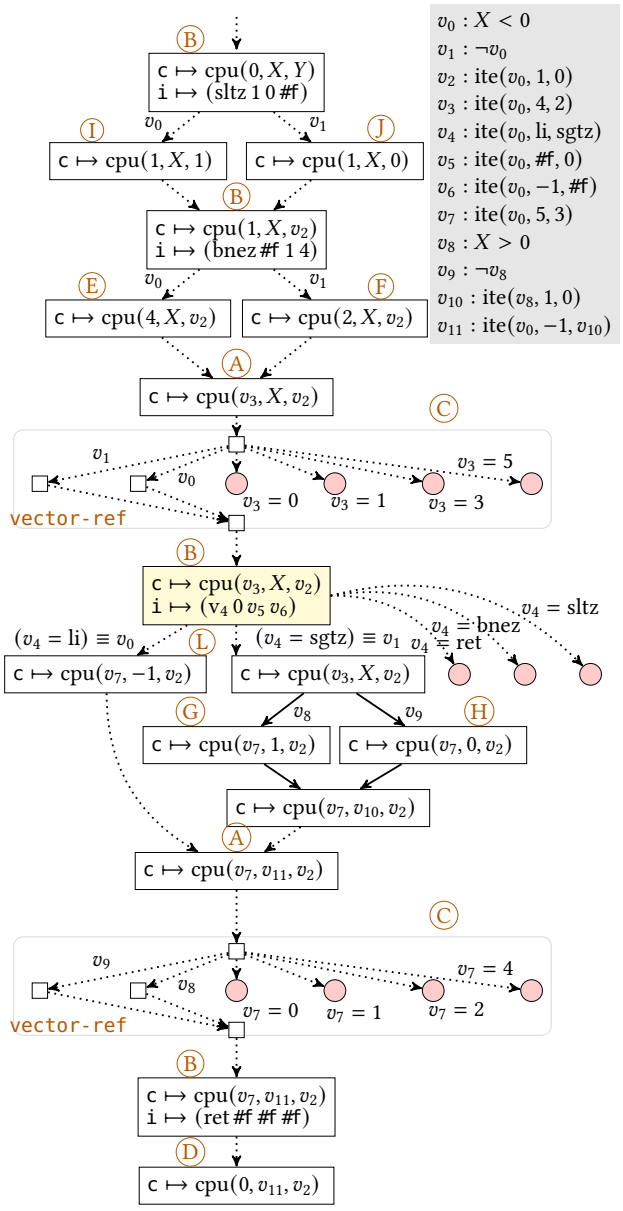
The ToyRISC interpreter (lines 1–48) implements a simple recursive procedure for executing a ToyRISC program from a given CPU state. The state consists of a program counter and vector of two registers, r_0 and r_1, both holding integers. A program is a sequence of instructions that manipulate the state. An instruction is a list of four values, `(opcode rd rs imm)`, specifying the instruction's opcode, destination and source registers, and the immediate constant. Unused arguments are denoted by `#f`; for example, the return instruction takes no arguments, denoted by `(ret #f #f #f)`. In addition to the return instruction, which halts the execution (line 10), the language also includes instructions for conditional branching (`bnez`), loading values into registers (`li`), and comparing register values to zero (`sgtz` and `sltz`). The example ToyRISC program, `sgnt`, uses these instructions to compute the sign of the value in register r_0, storing the result back into r_0 and using r_1 as a scratch register.

The `sgnt` program is correct if it produces the same result as the host sign procedure, `sgn`, for all valid CPU states. To verify `sgnt`, we first use Rosette's `define-symbolic` form to create two fresh symbolic integers, X and Y, and bind them to the variables `X` and `Y` (line 59). Next, we use these variables to create a CPU state `c` with the program counter set to 0 and registers set to X and Y (line 60). The symbolic state `c` represents all valid concrete CPU states. Finally, we interpret `sgnt` on `c` and use Rosette's `verify` query to search for a counterexample to the assertion that register r_0 holds the sign of X. A counterexample to this query would bind the symbolic values X and Y to concrete integers that trigger the assertion failure. But since `sgnt` is correct, the query returns `(unsat)` to indicate the absence of counterexamples.

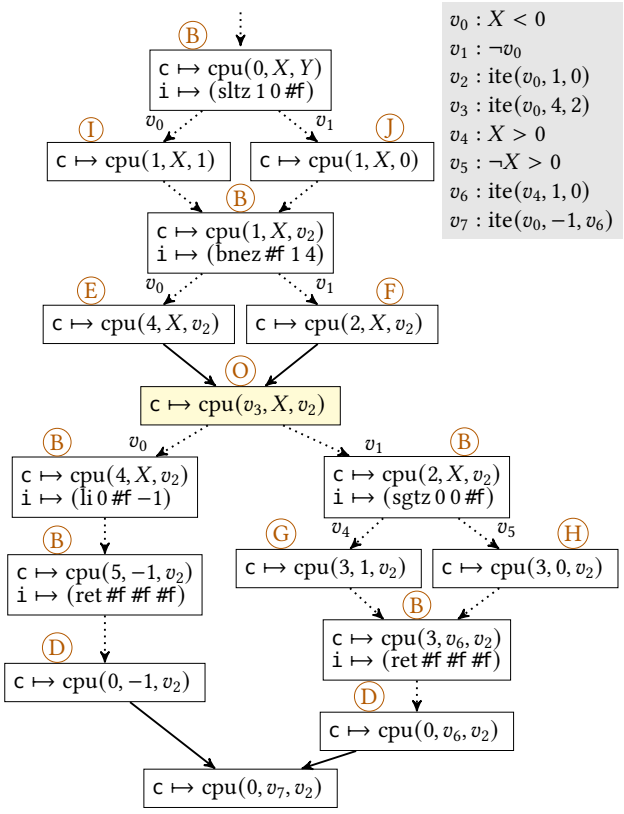
Symbolic evaluation and profiling. When interpreting `sgnt` on the symbolic state `c`, Rosette evaluates all paths through the interpreter code and reduces their meaning to symbolic expressions over X and Y. For example, after the call to the interpreter (line 61), register r_0 of `c` holds the symbolic value $\text{ite}(X < 0, -1, \text{ite}(0 < X, 1, 0))$, which encodes the meaning of `sgnt`. This value is part of the *symbolic heap* that Rosette generates while exploring the interpreter's *symbolic evaluation graph* [14] (Figure 3.2a). The heap consists of all symbolic values created during evaluation. The graph is a DAG over program states and guarded transitions between states, and its shape reflects the evaluator's strategy for path splitting and state merging. The symbolic heap and evaluation graph characterize the behavior of solver-aided code under every (forward) symbolic evaluation strategy, and controlling their complexity is key to good performance [14].

To help with this task, Rosette provides a *symbolic profiler*, SymPro, that monitors the heap and the graph to identify performance bottlenecks. SymPro computes summary statistics about the effect of each procedure call on these structures, such as the number of symbolic values added to the heap, and the number of path splits and state merges added to the graph. It then uses these statistics to rank the calls to suggest likely bottlenecks. When applied to ToyRISC, SymPro identifies the calls to `execute` at line 9 and `vector-ref` at line 16 as the likely bottlenecks. But how does one diagnose and fix these bottlenecks?

Manually repairing bottlenecks. The authors of ToyRISC reasoned [60] that the first location returned by SymPro "is not surprising since `execute` implements the core functionality, but `vector-ref` is a red flag." Examining the merging statistics for `vector-ref`, they concluded that `vector-ref` is being invoked with a symbolic program counter to produce a "merged symbolic instruction" (highlighted in Figure 3.2a), which represents a set of concrete instructions, only some of which are feasible. Since `execute` consumes this symbolic instruction (line 9), its evaluation involves exploring infeasible paths, leading to degraded performance on our example and



(a) Original (Figure 3.1)



(b) Repaired (Figure 3.3)

Figure 3.2: Simplified symbolic evaluation heap and graph for the original (a) and repaired (b) ToyRISC code. Heaps are shown in gray boxes. Nodes in a symbolic evaluation graph are program states, and edges are guarded transitions between states, labeled with the condition that guards the transition. Edges ending at pink circles denote infeasible transitions. Dotted edges indicate elided parts of the graph. Circled letters are program locations, included for readability.

```
(define (interpret c program)       ; A
  (serval:split-pc [cpu pc] c       ; O
    (define i (fetch c program))    ; B
    (match i
      [(list opcode rd rs imm)
      (execute c opcode rd rs imm)
      (when (not (equal? opcode 'ret))
        (interpret c program))])))
```

(a) Manual repair [60]

```
(define (interpret c program)       ; A
  (split-all (c)                    ; O
    (define i (fetch c program))    ; B
    (match i
      [(list opcode rd rs imm)
      (execute c opcode rd rs imm)
      (when (not (equal? opcode 'ret))
        (interpret c program))])))
```

(b) Generated repair

Figure 3.3: Manual and SymFix repair for ToyRISC code.

non-termination on more complex ToyRISC programs.

Having diagnosed the problem, the authors of ToyRISC then reasoned that the fix should force Rosette to split the evaluation into separate paths that keep the program counter concrete. Such a fix can be implemented through *symbolic reflection* [93], a set of constructs that allow programmers to control Rosette's splitting and merging behavior. In this case, ToyRISC authors used symbolic reflection and metaprogramming with macros (which Rosette inherits from Racket) to create a custom construct, `split-pc`, that forces a path split on CPU states with symbolic program counters. Applying `split-pc` to the body of `interpret` (Figure 3.3a) fixes this bottleneck (Figure 3.2b)—and ensures that symbolic evaluation terminates on all ToyRISC programs. But while simple to implement, this fix is hard won, requiring manual analysis of symbolic profiles, diagnosis of the bottleneck, and, finally, repair with a custom construct based on symbolic reflection.

Repairing bottlenecks with SymFix. SymFix lowers this burden by automatically repairing common performance bottlenecks in solver-aided code. The core idea is to view the repair problem (Section 3.3) as search for a sequence of semantics-preserving repairs that transform an input program into an equivalent program with minimal symbolic cost—a value computed from the program's symbolic profiling metrics. To realize this approach, SymFix solves two core technical challenges (Section 3.4): (1) developing a small set of generic repairs that can be combined into useful and general repair sequences for common bottlenecks, and (2) developing a search strategy

that discovers good fixes quickly and best fixes eventually.

SymFix can repair complex bottlenecks in real code as well as or better than experts (Section 3.5). It can also repair ToyRISC, finding the fix in Figure 3.3b. This fix has the same effect as the expert `split-pc` fix but uses a generic `split-all` construct. The construct forces a split on the value stored in a variable depending on its type: if the value is a `struct`, the split is performed on all of its fields that hold symbolic values. The `split-all` construct can be soundly applied to any bound occurrence of a local variable in a program, leading to intractable search spaces even for small programs. For example, there are 55 bound local variables in ToyRISC, so the `split-all` repair alone can be used to transform ToyRISC into 2^{55} syntactically distinct programs. SymFix is able to navigate this large search space effectively, matching the expert fix in a few seconds.

3.3 Symbolic Performance Repair

As Section 3.2 illustrates, performance bottlenecks in solver-aided code are difficult to repair manually. This section presents a new formulation of this problem that enables automatic solving. Our formulation is based on two core concepts: repairs and fixes. A *repair* is a semantics-preserving transformation on programs. A *fix* combines a sequence of repair steps, with the goal of reducing the *cost* of a program under symbolic evaluation. The *symbolic performance repair problem* is to find a fix, drawn from a finite set of repairs, that minimizes this cost. We describe repairs and fixes first, present the symbolic performance repair problem next, and end with a discussion of key properties of repairs that are sufficient for solving the repair problem in principle and helpful for solving it in practice.

3.3.1 Repairs, fixes, and symbolic cost

Repairs. A *repair* transforms a program to a set of programs that are syntactically different but semantically equivalent (Definition 3.1 and 3.2). A repair operates on programs represented as abstract syntax trees (ASTs). It takes as input an AST and a node in this AST, and produces an ordered set of ASTs that transform the input program at the given node or one of its ancestors. This interface generalizes classic program transformations by allowing repairs to produce multiple ASTs. The classic interface is often implemented by heuristically choosing one of many possible outputs that an underlying rewrite rule can generate. Our interface externalizes this choice, while still letting repairs provide heuristic knowledge in the order of the generated ASTs, as illustrated in Example 3.1. This enables an external algorithm to drive the search for fixes, with advice from the repairs.

Definition 3.1 (Program) *A* program *is an abstract syntax tree (AST) in a language $\mathcal{P}$, consisting of labeled nodes and edges. A program $P \in \mathcal{P}$ denotes a function $[\![P]\!] : \Sigma \to \Sigma$ on* program states*, which map program variables to values. Programs P and P' are syntactically equivalent if their trees consist of identically labeled nodes, connected by identically labeled edges. They are semantically equivalent iff $[\![P]\!]^{\mathcal{P}}\sigma \equiv [\![P']\!]^{\mathcal{P}}\sigma$ for all program states $\sigma \in \Sigma$, where $[\![\cdot]\!]^{\mathcal{P}} : \mathcal{P} \to \Sigma \to \Sigma$ denotes the concrete semantics of $\mathcal{P}$.*

Definition 3.2 (Repair) *A* repair *$R : \mathcal{P} \to \mathcal{L} \to 2^{\mathcal{P}}$ is a function that maps a program and a location to an ordered finite set of programs. A location $l \in \mathcal{L}$ identifies a node in an AST. The set $R(P, l)$ is empty if the repair R is not applicable to P at the location l. Otherwise, each program $P_i \in R(P, l)$ satisfies two properties. First, P and P_i differ in a single subtree rooted at l or an ancestor of l in P. Second, P and P_i are semantically equivalent.*

Example 3.1 *Consider a repair R_1 that performs the rewrite $e * 2 \longrightarrow e \ll 1$ on integer expressions. There are three ways to apply this rewrite to the program $P =$* `1 + (a * 2) * 2` *at the node* `a` *or its ancestors, and R_1 orders them as follows:*

```
1 + (a << 1) << 1 ; 0: Apply the rewrite exhaustively.
1 + (a << 1) * 2  ; 1: Apply the rewrite just to a's parent.
1 + (a * 2) << 1  ; 2: Apply the rewrite just to a's grandparent.
```

The order of the generated ASTs suggests that applying the rewrite exhaustively is most useful, followed by applying it from the inside out.

Fixes. A *fix* composes a sequence of *repair steps* into a function from programs to programs (Definition 3.3 and 3.4). A repair step $\langle R, l, i\rangle$ specifies the repair R to apply to a program, the location l at which to apply it, and the index i of the program to select from the resulting ordered set of programs. In essence, a repair step turns a repair into a classic program transformation by choosing one of the repair's outputs, and fixes can compose these steps to create new transformations, as illustrated in Example 3.2.

Definition 3.3 (Repair step) *A* repair step $\langle R, l, i\rangle$ *consists of a repair R, program location l, and non-negative integer i. A step denotes the function $[\![\langle R, l, i\rangle]\!] : \mathcal{P} \cup \{\bot\} \to \mathcal{P} \cup \{\bot\}$ as follows: $[\![\langle R, l, i\rangle]\!]P = R(P, l)[i]$ if $P \neq \bot$ and $|R(P, l)| > i$; otherwise the result is $\bot$. We write $R(P, l)[i]$ to mean the i^{th} program in the ordered set $R(P, l)$.*

Definition 3.4 (Fix) *A* fix $F = [\langle R_1, l_1, i_1\rangle, \ldots, \langle R_n, l_n, i_n\rangle]$ *is a finite sequence of one or more repair steps. A fix F denotes the function that composes the repair steps of F, i.e., $[\![F]\!] = [\![\langle R_n, l_n, i_n\rangle]\!] \circ \ldots \circ [\![\langle R_1, l_1, i_1\rangle]\!]$. We say that fix is* successful *for a program P if $[\![F]\!]P \neq \bot$.*

Example 3.2 *Consider the fix $F = [\langle R_1, a, 0\rangle, \langle R_2, a, 0\rangle]$, where R_1 is the repair from Example 3.1 and R_2 performs the rewrite $(e \ll 1) \ll 1 \longrightarrow e \ll 2$. Applying F to the program P from Example 3.1 produces the program `1 + (a « 2)`: $[\![\langle R_2, a, 0\rangle]\!]([\![\langle R_1, a, 0\rangle]\!]P) = [\![\langle R_2, a, 0\rangle]\!]$`(1 + (a « 1) « 1)` $=$ `1 + (a « 2)`. In other words, the fix F composes its repair steps to rewrite the second subexpression of P using the rule $(e * 2) * 2 \longrightarrow e \ll 2$.*

Cost. There are many ways to combine repairs into fixes for a given program, even when the program is small and repairs are few (Example 3.3). To choose a fix that is *useful* for improving the performance of a program under *symbolic evaluation*, we need a way to measure the *cost* of symbolic evaluation (Definition 3.5). We address this challenge by building on the observation that the behavior of symbolic evaluators is characterized by two structures: the symbolic heap and the symbolic evaluation graph. Our framework defines symbolic evaluation as a function from programs and program states to these structures (Definition 3.6), and the cost of symbolic evaluation as a function from these structures to (natural) numbers (Definition 3.7). The details of the cost function are not important for the framework, although they are important in practice: the symbolic cost should correlate with concrete metrics that are meaningful to developers (e.g., end-to-end running time), and SymFix uses a cost function (Section 3.4) that is simple but effective (Section 3.5). What matters, however, is that the symbolic evaluator is a total function, which means that we consider only finite computations. In particular, we make the standard assumption that programs $P \in \mathcal{P}$ are free of input-dependent loops, and are therefore guaranteed to terminate under symbolic evaluation, ensuring that we can compute the cost for every fix.

Definition 3.5 (Useful fix) *A fix F is* useful *for a program $P \in \mathcal{P}$, program state $\sigma \in \Sigma$, symbolic evaluator $S : \mathcal{P} \to \Sigma \to \mathcal{G}$, and cost $c : \mathcal{G} \times \mathcal{H} \to \mathbb{N}$, if $[\![F]\!]P \neq \bot$ and $c(S([\![F]\!]P, \sigma)) < c(S(P, \sigma))$.*

Definition 3.6 (Symbolic evaluator) *A* symbolic evaluator *$S : \mathcal{P} \to \Sigma \to \mathcal{G} \times \mathcal{H}$ is a function that takes as input a program $P \in \mathcal{P}$ and program state $\sigma \in \Sigma$, and outputs a pair $\langle G, H \rangle$, where $G \in \mathcal{G}$ is a symbolic evaluation graph and $H \in \mathcal{H}$ is a symbolic heap [14]. A* symbolic heap *$H = (V_H, E_H)$ is a directed acyclic graph (DAG) with labeled nodes and edges. Heap nodes are* symbolic values*, and heap edges connect compound symbolic values to the symbolic or concrete values from which they are built. A* symbolic evaluation graph *$G = (V_G, E_G)$ is a DAG where nodes $V_G \subseteq \Sigma$ are program states and edges are transitions between states, each labeled with a (symbolic*

or concrete) boolean value that guards the transition and a program location in P that caused the transition. The graph G has $\sigma \in V_G$ as its sole source node. The heap H contains all symbolic values that appear in G as part of a program state or as an edge label. If $H = (\emptyset, \emptyset)$ is empty, then G consists of a single path from σ to $[\![P]\!]^{\mathcal{P}}\sigma$, where all edges are labeled with $\top$.

Definition 3.7 (Symbolic cost) *A* symbolic cost *function $c : \mathcal{G} \times \mathcal{H} \to \mathbb{N}$ assigns a cost, expressed as a natural number, to the results of symbolic evaluation.*

Example 3.3 *Consider again the fix F, repairs R_1 and R_2, and program P from Example 3.1 and 3.2. In addition to F, there are seven different ways to compose repair steps over R_1 and R_2 into fixes for P; two are equivalent to F and five to the outputs of R_1 on P. Intuitively, F produces the best program for all workloads, and in this case, the intuition is captured by a simple cost function that measures the size of the symbolic heap, i.e., $c(\langle G, H\rangle) = |V_H|$. For example, letting $\sigma = \{\mathsf{a} \mapsto A\}$, where A is a symbolic integer, we can compute the cost of P, the output of the fix $[\langle R_1, \mathsf{a}, 0\rangle]$, and the output of the fix F as follows:*

$$c(S(\texttt{1 + (a * 2) * 2}, \sigma)) = |\{v_0 : A * 2, v_1 : v_0 * 2, v_2 : 1 + v_1\}| = 3$$

$$c(S(\texttt{1 + (a « 1) « 1}, \sigma)) = |\{v_0 : A \ll 1, v_1 : v_0 \ll 1, v_2 : 1 + v_1\}| = 3$$

$$c(S(\texttt{1 + (a « 2)}, \sigma)) = |\{v_0 : A \ll 2, v_1 : 1 + v_0\}| = 2$$

As expected, the program produced by F has the lowest cost.

3.3.2 The symbolic performance repair problem

The *symbolic performance repair problem* is to find a fix, drawn from a finite set of repairs, that minimizes the symbolic cost of a program on a given workload (Definition 3.8). To make this problem solvable in principle, it is sufficient to ensure that the set of repairs is *terminating* [33], preventing the repairs from being indefinitely applicable to any program (Definition 3.9). To help solve the repair problem in practice, we can use a general property of repairs, *precedence*, to prune fixes during

search without missing any programs (Definition 3.10). A partial order $\preceq_{\mathsf{R}}$ is a precedence relation on a set of repairs R if every successful fix over R can be turned into an equivalent fix by permuting its repair steps to respect $\preceq_{\mathsf{R}}$. To search for a fix over R with $\preceq_{\mathsf{R}}$, it is sufficient to explore successful fixes that order all repair steps according to $\preceq_{\mathsf{R}}$. Example 3.4 illustrates these definitions, and we use them in the next section to develop the SymFix algorithm for solving the repair problem.

Definition 3.8 (Symbolic performance repair) *Let $P \in \mathcal{P}$ be a program, $\sigma \in \Sigma$ a program state,* R *a finite set of repairs for $\mathcal{P}$, S a symbolic evaluator for $\mathcal{P}$, and c a symbolic cost function for S. The* symbolic performance repair problem *is to find a useful fix F over* R *that minimizes the cost of evaluating P on σ; i.e., for all useful fixes $F' \neq F$ over* R, $c(S(\llbracket F \rrbracket P, \sigma)) \leq c(S(\llbracket F' \rrbracket P, \sigma))$.

Definition 3.9 (Terminating repair set) *Let* R *be a finite set of repairs for the language $\mathcal{P}$. We say this set is* terminating *if for every program $P \in \mathcal{P}$, there is an upper bound on the length of every successful fix for P drawn from* R.

Definition 3.10 (Repair precedence) *Let* R *be a finite set of repairs and $\preceq_{\mathsf{R}}$ a partial order on* R. *Let* spine *be a function that projects out the repairs from a fix, i.e.,* $\mathrm{spine}(F) = [R_1, \ldots, R_n]$ *for* $F = [\langle R_1, l_1, i_1 \rangle, \ldots, \langle R_n, l_n, i_n \rangle]$. *We say that $\preceq_{\mathsf{R}}$ is a* precedence *on* R *if for every program P and every successful fix F for P drawn from* R, *there is a fix F' such that $\llbracket F \rrbracket P = \llbracket F' \rrbracket P$ and* $\mathrm{spine}(F')$ *permutes* $\mathrm{spine}(F)$ *to respect $\preceq_{\mathsf{R}}$, i.e.,* $\forall i, j.\, \mathrm{spine}(F')[i] \preceq_{\mathsf{R}} \mathrm{spine}(F')[j] \implies i \leq j$.

Example 3.4 *Recall the program P, repairs R_1 and R_2, fix F, and cost c from Examples 3.1–3.3. The repair set* $\mathsf{R} = \{R_1, R_2\}$ *is terminating; $R_1 \preceq_{\mathsf{R}} R_2$; and F is a solution to the symbolic performance repair problem for P,* R, *and c.*

3.4 The SymFix Algorithm and Repairs

This section presents the SymFix system for solving the symbolic performance repair problem. SymFix consists of two components: an anytime algorithm for searching the space of fixes drawn from a terminating set of repairs, and a set of three generic repairs for functional solver-aided languages with symbolic reflection. We present the algorithm first and prove its correctness and optimality (Section 3.4.1). We then describe the repairs and a total precedence relation on them, and argue that they form a terminating set (Section 3.4.2). We end by highlighting the key details of our implementation of SymFix for the Rosette language (Section 3.4.3).

3.4.1 Profile-guided search for fixes

The SymFix algorithm (Figure 3.4) solves the symbolic performance repair problem for a cost function based on symbolic profiling. As shown in prior work [14], the metrics computed by a symbolic profiler closely reflect the overall running time of solver-aided code (i.e., symbolic evaluation together with solving time), and reducing these metrics is key to improving performance. In addition to computing these metrics, which measure the size and shape of the symbolic heap and evaluation graph, a symbolic profiler M also ranks all locations in a program from most to least expensive to evaluate. The SymFix algorithm uses both of these outputs: it searches for a fix that minimizes the sum of the profiling metrics for a given program and workload, and the search is guided by the profiling ranks.

The algorithm relies on the SEARCH procedure to explore the space of fixes for a program P_{in} and a terminating set of repairs R. SEARCH performs exhaustive (rather than greedy) best-first search over this space. It starts by initializing the work set W with the input program P_{in}; the *info* map with a binding from P_{in} to its profiling metrics, cost, and the empty fix; and the minimum cost *minCost* with the cost of P_{in}. The main search loop then picks a program P from the work set,

```
1: function SymFix(P_in, σ, S, M, R, ≤_R)
2:     function Info(P, F)                                  ▷ Symbolic profile sorts P's locations from most to least
3:         ⟨⌊P, m_P⟩ ← M(S(P, σ))                           ▷ likely bottlencks & collects k profiling metrics.
4:         c_P ← Σ_{0≤i<k} m_i                              ▷ P's cost is the sum of its profiling metrics.
5:         return {P ↦ {cost ↦ c_P, locs ↦ ⌊P, fix ↦ F}}
6:     end function
7:     function Next(P, info)                               ▷ Picks a successor of P, if any, with an extra repair.
8:         F ← info[P][fix]                                 ▷ Get the fix that generated P.
9:         for R in R do                                    ▷ Iterate over the repairs in R that do not precede
10:            if ⋀_{R_i ∈ spine(F)} R_i = R ∨ R ⋠_R R_i then   ▷ any repairs in P's fix,
11:                for l in info[P][locs] do                ▷ then over the ranked locations in P,
12:                    for P_j ∈ R(P, l) do                 ▷ and then over the ordered results
13:                        if info[P_j] = ⊥ then            ▷ to find a new program P_j.
14:                            return ⟨P_j, append(F, ⟨R, l, j⟩)⟩   ▷ Return P_j and its fix.
15:                        end if
16:                    end for
17:                end for
18:            end if
19:        end for
20:        return ⟨⊥, ⊥⟩                                    ▷ No new programs can be obtained from P.
21:    end function
22:    function Search(P_in)
23:        W, info ← {P_in}, Info(P_in, [ ])                ▷ Initialize the work set and info map.
24:        minCost ← info[P_in][cost]                       ▷ Set P's cost as current best cost.
25:        while W ≠ ∅ do
26:            P ← min(W, λP.info[P][cost])                 ▷ Choose the cheapest P ∈ W to work on.
27:            ⟨P′, F′⟩ ← Next(P, info)                     ▷ Get a successor P′ of P and its fix.
28:            if ⟨P′, F′⟩ ≠ ⟨⊥, ⊥⟩ then                    ▷ If P′ exists,
29:                W, info ← W ∪ {P′}, info ∪ Info(P′, F′)   ▷ add P′ to W and info;
30:                if info[P′][cost] < minCost then         ▷ and if P′ is best so far,
31:                    minCost ← info[P′][cost]             ▷ update minCost and
32:                    print P′                             ▷ output P′.
33:                end if
34:            else
35:                W ← W \ {P}                              ▷ No new programs can be obtained from P.
36:            end if
37:        end while
38:    end function
39:    Search(P_in)
40: end function
```

Figure 3.4: The SymFix search algorithm takes as input a program P_{in} in a language $\mathcal{P}$, a workload σ, a symbolic evaluator S for $\mathcal{P}$, a symbolic profiler M for S, a terminating set of repairs R for $\mathcal{P}$, and a precedence relation $\preceq_{\text{R}}$ on R. It searches the space of fixes drawn from R to find a program that is equivalent to P_{in} and minimizes the cost of symbolic evaluation on σ according to the profiler M.

applies one repair step to P to get a new program P' (corresponding to a fix F' that extends P's fix by one step), and adds P' to both W and *info*. If the new program P' has lower cost than *minCost*, SEARCH prints it and updates *minCost* accordingly. But if no new programs can be obtained from P by applying a repair from R, then P has no more children in the underlying search graph, and SEARCH removes it from the work set W. The search continues as long as there are programs in W, so the entire search graph is eventually explored.

To make the algorithm practically useful, SEARCH employs the procedure NEXT to explore the most promising fixes first and to prune the search space without losing completeness. SEARCH selects the cheapest fix F to extend (line 26), and NEXT constructs the repair step $\langle R, l, j\rangle$ to add to F. To construct $\langle R, l, j\rangle$, NEXT first choses a repair R that does not strictly precede any of the repairs in F, according to the precedence relation $\preceq_{\mathsf{R}}$. Then, it uses profiling rankings and the repair's ordering heuristics to select the location l and the result index j. This ensures that SEARCH explores only fixes that respect $\preceq_{\mathsf{R}}$, and that it tries to repair most likely bottlenecks first.

The SymFix algorithm is sound, complete, and optimal (Theorem 3.1). It produces correct fixes that are semantically equivalent to the input program (soundness). It always finds a useful fix if one exists in the space defined by the given set of repairs (completeness). And it eventually finds the best such fix that minimizes the symbolic profiling cost on the given workload (optimality).

Theorem 3.1 *Let P_{in} be a program, σ a workload,* R *a terminating set of repairs, and $\preceq_{\mathsf{R}}$ a precedence relation on* R. *Then* SYMFIX$(P_{in}, \sigma, S, M, \mathsf{R}, \preceq_{\mathsf{R}})$ *terminates and satisfies the following conditions. (1) If* SEARCH *produces a program at line 32, then every such program P' is semantically equivalent to P_{in} (soundness). (2) For every cost $C < info[P_{in}][cost]$, if there is a fix over* R *with cost C, then line 32 will produce a program P' with $info[P'][cost] \leq C$ (completeness and optimality).*

Proof sketch 3.1 *First, note that SymFix explores a search graph where nodes are programs; two nodes are related by a repair step drawn from* R*; and a path in the graph corresponds to a fix over* R *that*

respects $\preceq_{\mathsf{R}}$. *All paths through this graph are finite because* R *is terminating (Definition 3.9). There are also finitely many such paths because each node has finitely many outgoing edges (repair steps), which follows from the finite number of repairs, locations in a program, and repair outputs. So, (1) the underlying search graph is finite, and (2) by definition of* $\preceq_{\mathsf{R}}$ *(Definition 3.10), it contains the same programs (nodes) as the search graph that includes all fixes (paths) over* R. *Next, note that (3) SymFix adds each program in this graph to the work set* W *exactly once, and (4) each added program is removed after all of its children have been visited, i.e., added to the info map. These facts (1–4) imply that the algorithm terminates after visiting each program in the space defined by* R. *Completeness and optimality then follow from lines 24, 30–32, and soundness follows from the definition of repairs (Definition 3.2).*

3.4.2 Effective repairs for functional hosts with symbolic reflection

The effectiveness of SymFix hinges on the choice of the repair set R. An ideal repair set includes a few key repairs that can be combined into useful fixes for most common performance bottlenecks. This section presents three such repairs for solver-aided languages with functional programming primitives and symbolic reflection. We use Rosette to illustrate these repairs, but they are applicable to any solver-aided language or framework with similar features (e.g., [94, 79, 20]).

Deforestation. Higher-order combinators (e.g., `map`, `fold`, and `filter`) are commonly used to operate on lists. Using these combinators generates intermediate lists that are immediately consumed and discarded, slowing down concrete evaluation by a constant factor. Under symbolic evaluation, however, the resulting slow down is asymptotically worse, as the following example demonstrates.

```
(define (sum-slow xs)                ; Sums the positive numbers in xs using an
  (foldr + 0 (filter positive? xs)))  ; intermediate list (the result of filter).

(define (sum-fast xs)                ; Sums the positive numbers in xs without
  (foldr (lambda (e acc)             ; creating any intermediate lists.
           (if (positive? e)
               (+ e acc)
               acc))
```

```
        0 xs))

> (define-symbolic xs integer? [100])  ; xs is a list of 100 symbolic integers.

> (time (sum-slow xs))                 ; Adds 520,000 values to the symbolic heap.
cpu time:  5119 real time:  4954 gc time:  2194

> (time (sum-fast xs))                 ; Adds 100 values to the symbolic heap.
cpu time:  3 real time:  3 gc time:  0    ; Times are given in milliseconds.
```

Deforestation [95] is a classic program transformation that eliminates intermediate lists produced by list combinators. As such, it makes a powerful repair for performance bottlenecks in solver-aided code. In the above example, it automatically transforms `sum-slow` into `sum-fast`, avoiding the expensive call to `filter` that creates a symbolic intermediate list when the input `xs` is symbolic. Many variants of deforestation exist for different functional languages; for Rosette, SymFix uses a repair based on build/foldr fusion [44]. This repair applies deforestation exhaustively at a given location and outputs at most one program.

Path splitting. Deforestation changes the behavior of a program under symbolic evaluation by restructuring its implementation. But if the host language supports symbolic reflection [93], we can control the evaluation more directly, by using dedicated constructs to force path splitting [90] (or state merging [78]) at specific program locations. We have seen an example of this in Section 3.2, where SymFix used a path splitting construct to fix the ToyRISC interpreter. In Rosette, this construct takes the form `(split-all (`x`) `E`)`, where x is an identifier and E an expression over x. If x is bound to a symbolic value that ranges over a small finite set of concrete values, $\{v_1, \ldots, v_n\}$, then `split-all` splits the evaluation of E into n paths, one for each value that x can take, i.e., $x = v_i \vdash$ `(let ([`x v_i`]) `E`)` for $1 \leq i \leq n$. Otherwise, `split-all` acts as the identity transformation on E. Because path splitting increases the number of paths that are evaluated, it must be applied carefully to avoid path explosion—a task we delegate to automated search.

The SymFix path splitting repair works as follows. Given a program location l in a procedure body P, it outputs all valid ways to insert `(split-all (`x`) `E`)` into P, so that E contains the

location l, x is bound in E's context, and there is no other split on x in E or its context. So, nested splits on the same identifier, `(split-all (`x`) (...(split-all (`x`) ...)`, are disallowed. The resulting set of transformed programs is finite but large, and the repair heuristically prefers splits with broadest scope (i.e., where E is the highest ancestor of l in P).

Value splitting. Path splitting allows programs to exert local control on the symbolic evaluation strategy, by concretizing a specific symbolic value at a specific program location. In principle, it is possible to combine many path splitting repairs to implement a global change in the evaluation strategy, such as concretizing every operation on a given user-defined type. In practice, however, this would require prohibitively long and complex fixes. We therefore develop a global value splitting repair that assumes the host language provides a mechanism for controlling how all values of a given type are merged and split. In Rosette, this is done with a *transparency* annotation, illustrated in the following example.

```
(require rosette/lib/match)

@(struct Cell (v) #:transparent)@

; Return a new Cell that doubles
; the value v of c.
(define (twice c)
  (match c
    [(Cell v) (Cell (+ v v))]))

; Create a symbolic Cell.
(define-symbolic b boolean?)
(define c (if b (Cell 1) (Cell 0)))

; Fields of transparent structs are merged,
; so 'twice' works on symbolic values.
> c
(Cell (ite b 1 0))
> (twice c)
(Cell (+ (ite b 1 0) (ite b 1 0)))

; The symbolic heap now contains 4 values:
; b, ¬b, ite(b, 1, 0), ite(b, 1, 0) + ite(b, 1, 0).
```

```
(require rosette/lib/match)

@(struct Cell (v))@ ; Opaque struct.

; Return a new Cell that doubles
; the value v of c.
(define (twice c)
  (match c
    [(Cell v) (Cell (+ v v))]))

; Create a symbolic Cell.
(define-symbolic b boolean?)
(define c (if b (Cell 1) (Cell 0)))

; Fields of opaque structs are not merged,
; so 'twice' works on concrete values.
> c
{[b ⊢ (Cell 1)] [(! b) ⊢ (Cell 0)]}
> (twice c)
{[b ⊢ (Cell 2)] [(! b) ⊢ (Cell 0)]}

; The symbolic heap now contains 2 values,
; b, ¬b, but the graph has more paths.
```

The SymFix value splitting repair toggles the transparency annotation on user-defined structures in a way that preserves soundness. Under Rosette semantics, it is sound to make structs less transparent (i.e., the transparency annotation can be removed) but not more. So given a location within a struct declaration, the value splitting repair produces at most one program. Like path

splitting, this repair creates more opportunities for concrete evaluation, at the cost of adding more paths to the symbolic evaluation graph.

Termination and precedence. The SymFix repairs form a terminating set with a total precedence relation $R_V \preceq_{\mathsf{R}} R_D \preceq_{\mathsf{R}} R_P$ that orders value splitting first, deforestation second, and path splitting last. To see this, first note that value splitting applies to structs, while neither of the other repairs does, so R_V can be freely reordered with R_D and R_P. Next, observe that if deforestation R_D follows path splitting R_P, then either they were applied to disjoint locations, or R_P was applied to an expression that is moved but not transformed by deforestation (e.g., `xs` in the `sum-slow` example). In either case, the same effect can be achieved by applying R_P after R_D (though not vice versa). Finally, note that R_V and R_D can be applied to the same location at most once, and R_P can be applied at most N times, where N is the number of bound identifiers in the enclosing context. Hence, the set $\{R_V, R_D, R_P\}$ is terminating.

3.4.3 Implementation

We implemented the SymFix algorithm and repairs for Rosette. All three repairs require side effect analysis to preserve soundness, and we implement a simple conservative analysis that allows repairs only on expressions built out of procedures and constructs known to be safe. Because the repairs are totally ordered, we apply them in stages so that all of our fixes are of the form $R_V^* R_D^* R_P^*$. While our repair framework assumes that programs have no unbounded loops, Rosette places no bounds on loops by design [93], so it is possible to write a Rosette program that does not terminate under symbolic evaluation. Our implementation deals with diverging and slow executions with timeouts.

3.5 Evaluation

To evaluate the effectiveness of SymFix, we address three research questions:

RQ1 : Can SymFix repair the performance of state-of-the-art solver-aided tools, and how do its fixes compare to those written by experts?

RQ2 : Do the fixes found by SymFix generalize to different workloads?

RQ3 : How important is SymFix's search strategy for finding useful fixes?

All results in this section were collected using an Intel Core i7-7700K at 4.20 GHz with 16 GB of RAM, running Racket v7.4. Each timing result is the average of 10 executions of the corresponding experiment.

3.5.1 Can SymFix repair the performance of state-of-the-art solver-aided tools, and how do its fixes compare to experts'?

To demonstrate that SymFix is effective on state-of-the-art solver-aided tools, we collected a suite of 15 tools [1, 12, 13, 15, 17, 19, 23, 50, 74, 70, 93, 100, 101] built in Rosette from a prior literature survey [14], together with 3 more recent tools [60, 64, 73]. For each of these Rosette programs, we applied SymFix to identify and repair performance bottlenecks.

Figure 3.5 shows the results. For each program, we report the original running time in seconds, and the cost of the original program as estimated by SymFix. We report three sources of repairs: fixes found by SymFix, fixes found by a baseline greedy algorithm discussed in Section 3.5.3, and manual fixes from prior work [14, 60]. We used a one-hour timeout for all experiments. For each fix, we report the relative speedup and cost decrease compared to the original run time and cost. A dash "–" indicates the absence of data due to timeouts or the lack of known manual fixes. One

		Original		SymFix				Greedy		Manual	
Program	LoC	Time	Cost	Time	Cost	$\|F\|$	#	Time	Cost	Time	Cost
Bagpipe	3317	17 s	6.0e4	1.0×	1.0×	1	6	–	–	–	–
Bonsai[§]	641	27 s	1.5e6	1.3×	1.3×	2	21	1.1×	1.1×	1.0×	0.9×
Cosette[‖]	2709	–	–	21 s	6.8e5	3	33	–	–	15 s	7.4e5
Ferrite	350	13 s	9.8e5	2.8×	3.8×	4	11	–	–	1.6×	1.1×
Fluidics	145	10 s	6.5e5	1.9×	1.7×	1	1	1.9×	1.7×	2.1×	1.8×
FRPSynth	304	3 s	2.3e4	3.1×	1.6×	4	93	1.4×	1.3×	–	–
GreenThumb	934	1179 s	2.0e5	1.3×	1.1×	1	1	1.3×	1.1×	–	–
IFCL	574	53 s	6.2e5	–	–	–	–	–	–	–	–
Memsynth	3362	15 s	2.0e6	1.1×	1.1×	1	2	1.0×	1.1×	–	–
Neutrons	37317	29 s	5.6e6	2.0×	2.3×	3	5	2.0×	2.3×	193.7×	869.9×
Nonograms	6693	8 s	1.5e5	1.1×	1.4×	7	46	–	–	–	–
Quivela	5946	47 s	2.9e6	91.7×	218.4×	6	7	90.1×	187.3×	86.1×	218.5×
RTR	2007	282 s	1.6e7	–	–	–	–	–	–	7.2×	4.1×
Serval[¶]	8641	116 s	7.3e6	6.2×	80.7×	1	1	–	–	6.2×	80.7×
Swizzle	1240	7 s	3.1e5	1.8×	1.3×	2	18	–	–	–	–
SynthCL	3732	16 s	7.5e5	–	–	–	–	–	–	–	–
Wallingford	3866	2 s	8.5e3	1.0×	1.0×	1	2	–	–	–	–
WebSynth	2057	7 s	1.0e6	–	–	–	–	–	–	–	–

[§] The manual repair was made unnecessary by a subsequent Rosette improvement.
[‖] The repair by SymFix involves independent changes from users.
[¶] The repair by SymFix uses user-supplied repairs.

Figure 3.5: Summary of fixes found by SymFix, a baseline greedy search, and experts. "LoC" is the number of lines of code in a given benchmark; "Cost" is SymFix's cost function for search; "$|F|$" is the number of repair steps in the fix found by SYMFIX; and "#" is the number of fixes explored in one hour before the reported best one is found.

original program (Cosette) does not terminate within an hour, so we report only its repaired running times and costs.

SymFix finds fixes that improve the performance of 12 programs, with the improvements ranging from 1.1× to 91.7×. SymFix also finds 2 fixes that lower the symbolic cost and runtime only slightly, marked as 1.0× in Figure 3.5. The "#" column reports the number of iterations of SymFix's search procedure needed to find the fix, and "$|F|$" reports the number of repair steps in the fix. Most fixes are found in fewer than 10 iterations, and most have up to 2 repair steps.

Of the 15 benchmarks from prior work, 7 were manually fixed by the authors of that paper. For two of these benchmarks (Neutrons and RTR), the expert finds a significantly better fix than SymFix or finds some fix while SymFix finds none. Overall, SymFix matches or outperforms experts on 4 benchmarks, and it finds fixes for 5 benchmarks with no expert fix. We inspected

all the fixes manually, and discuss interesting cases below.

For **Bonsai** (a synthesis tool for checking type-system soundness [19]), **Neutrons** (a verifier for safety-critical systems [70]), and **RTR** (a refinement type checker for Ruby), the manual fixes were sound but not semantics-preserving, so SymFix cannot discover them. For Bonsai, the manual fix was made unnecessary by a subsequent Rosette improvement, but SymFix still discovers a new repair that improves the performance further. For Neutrons, SymFix cannot recover the manual fix but does find a concretization opportunity offering a 2.0× speedup. For RTR, SymFix does not find a useful fix, suggesting future opportunities to exploit *conditional* repairs that are only sound under certain preconditions [81].

For **Cosette**, an automated prover for deciding the equivalence of two SQL queries [23], the original implementation did not terminate within one hour. The expert fix allowed Cosette to terminate in 15 seconds. Because SymFix needs to execute the original program during the search for repairs, we imposed a timeout of 60 seconds per execution. SymFix finds a fix that reduces Cosette's run time to 21 seconds, comparable to the manual fix. This new fix combines path splitting and deforestation of the map–reduce pattern Cosette uses to filter SQL tables. Finding the deforestation repair required converting Cosette's recursive implementation of this pattern into a higher-order version, but the Cosette developers made this change independently to implement the manual fix; SymFix exploited this new structure to find another fix that allows Cosette to terminate in seconds.

For **Fluidics**, a tool for synthesizing programs that control a digital microfluidics array [101], the expert-written fix involves a change to the core data structure the tool uses to represent the array. This change is outside the scope of SymFix's search space. However, SymFix instead discovers a different fix that uses path splitting and requires no changes to the data structure. This fix offers a 1.9× speedup instead of 2.1× for the manual one, but it is made automatically and allows the tool's developers to retain their preferred data structure.

For **Ferrite**, a tool for checking file-system crash consistency [12], SymFix improves upon the

		Original		SymFix	
Program	Input	Time (s)	Cost	Time	Cost
Bonsai	nanodot	17 s	7.8e5	1.2×	1.1×
Cosette	q2	1 s	4.6e4	2.2×	9.8×
Cosette	q3	–	–	33 s	1.3e6
Ferrite	chrome	99 s	2.1e7	16.2×	15.6×
FRPSynth	program0	2 s	1.9e4	0.8×	0.8×
Quivela	test-etm-10	19 s	6.7e5	1.8×	1.8×
Serval	enosys	105 s	8.0e5	1.8×	11.3×
Swizzle	stencil	6 s	2.1e5	1.1×	1.1×
Swizzle	aos-sum	5 s	5.4e4	1.1×	1.0×

Figure 3.6: Effectiveness of SymFix's repairs from Figure 3.5 on alternative workloads.

expert-written fix by finding additional opportunities for concretization through path and value splitting. These changes make Ferrite close to 2× faster than the expert-repaired version.

For **GreenThumb**, a tool for developing superoptimizers [74], SymFix finds a concretization opportunity that the expert did not. The concretization both improves symbolic evaluation and alters the shape of the SMT formula so that SMT solving is 1.1× faster. SymFix also finds previously unknown concretization opportunities for **FRPSynth**, a tool for synthesizing functional reactive programs [64], and **Swizzle**, a tool for synthesizing GPU kernels [73], leading to a 3.1× and 1.8× speed-up, respectively.

For **Serval**, a toolset for automatic verification of systems software [60], SymFix does not discover a significant fix using its built-in repairs. But Serval comes with its own set of *symbolic optimizations*, which were originally designed for manual application [60]. Using these optimizations as repairs, SymFix discovers the manual fix, showing that its algorithm works well with a variety of repairs.

3.5.2 Do the fixes found by SymFix generalize to different workloads?

SymFix generates each of the fixes in Figure 3.5 using a single input to the respective program. To determine whether discovered repairs generalize to *different* program inputs, we identified the

programs in Figure 3.5 that have alternative inputs available and executed the repaired versions on them.

Figure 3.6 shows the performance of each program on alternative inputs, both before and after the fix that SymFix discovered in Figure 3.5. In all but one case, the fix generalizes to the new input and improves the program's performance. The relative performance improvement varies from Figure 3.5 due to different problem sizes; for example, the new Ferrite input is much larger than the original and so spends comparatively less time in the fixed procedure. The one exception is the "program0" input to FRPSynth, which is 20% slower than the original version. Manual inspection of this fix shows that the last of its 4 repair steps overfits to the initial input, and the first 3 steps improve the performance on both inputs.

3.5.3 How important is SymFix's search strategy for finding fixes?

SymFix employs a complete form of best-first search, guided by symbolic profiling ranks. To evaluate the importance of these design choices, we consider two alternative algorithms without them:

Random implements a complete best-first search that is not guided by profiling ranks, and instead chooses a location randomly at line 11 in Figure 3.4; and

Greedy implements the standard greedy best-first search, which applies only the first repair produced by Next at line 27 and never backtracks (by removing P from W unconditionally at line 35).

The **Random** algorithm discovers no useful fix for any benchmark within a one hour timeout. This is not surprising since the space of fixes is exponential in the number of potential repair locations, and there are thousands of such locations in each benchmark. The results for the **Greedy**

algorithm are reported in the last two columns of Figure 3.5. **Greedy** finds a useful fix for only half (7) of the benchmarks repaired by SymFix, and none of its fixes are better than those found by SymFix. These results show that the key features of the SymFix algorithm are vital for fixing performance bottlenecks in real solver-aided tools.

3.6 Related Work

Profile-guided optimization. Compilers often support *profile-guided* optimization, in which the compiler uses profile data to guide its optimization phases (see Gupta et al. [45] for a survey). For example, the efficacy of inlining depends on factors including cache size and access patterns that are best determined by executing the program in the intended environment. Pettis and Hansen [71] introduce a profile-guided code layout algorithm that tries to position commonly used code together in memory to improve spatial locality. As another example, many JIT compilers for both static and dynamic languages will *specialize* methods based on type information observed at run time [42, 69] (e.g., specializing virtual calls for a particular concrete receiver). SymFix takes inspiration from these approaches, using profile data to guide the application of semantics-preserving repairs. But unlike them, SymFix focuses on optimizing a program's symbolic evaluation strategy rather than its utilization of machine resources.

Not all profile-guided optimization techniques are automated. Optimization coaching [84] is an interactive tool that gives programmers feedback about the optimizations a compiler applied to their program, and optimizations that it tried unsuccessfully. SymFix does not provide interactivity, but because its repairs are high level, it can follow the optimization coach approach of reporting them to the programmer as syntactic changes to their input program.

Symbolic profiling. Because SymFix uses profile data to guide the search for fixes, its effectiveness depends on high quality profiles. SymFix builds on *symbolic profiling* [14], a technique for profiling the behavior of symbolic evaluation engines. Symbolic profiling generalizes across a spectrum of symbolic evaluation techniques, and so SymFix's approach could generalize to other engines that support symbolic profiling (e.g., Crucible [43]). Other profiling techniques measure different aspects of automated tools. The Z3 Axiom Profiler [4] measures axiom instantiations in the Z3 [31] SMT solver's quantifier theory module. It can be used to detect optimization opportunities at the SMT level. Using such profilers to extend SymFix to the SMT level is an interesting direction for future work.

Optimizing symbolic evaluation. A number of approaches exist for interactively improving the performance of tools based on symbolic evaluation. Wagner et al. [97] introduce a configuration for optimizing compilers to prioritize generating code that is amenable to symbolic execution. Cadar [18] develops a suite of compiler optimizations that make code easier to evaluate symbolically. Nelson et al. [60] develop a set of custom symbolic optimizations that can be manually applied to build scalable verifiers for low-level languages (e.g., RISC-V [98], LLVM [54], x86 [47], and eBPF [40]) on top of a generic verification framework. SymFix is complimentary to these approaches: it can automatically apply custom optimizations to verifiers for low-level code, and these verifiers can further benefit from the custom compiler optimizations applied to their input programs.

3.7 Conclusion

This chapter presented a new approach to repairing performance bottlenecks in code under symbolic evaluation. Our approach rests on three technical contributions. We formulate the symbolic

performance repair problem as combinatorial search for a fix that applies a sequence of semantics-preserving repairs to a program and a workload; the resulting fixed program is guaranteed to be equivalent to the input program, and to have minimal symbolic evaluation cost on the input workload. To solve this repair problem, we develop SymFix, a system with two key components: (1) a small set of generic repairs based on deforestation and symbolic reflection, and (2) an anytime search algorithm that uses symbolic profiling to guide the exploration of this space. Our evaluation shows that SymFix can discover useful fixes for state-of-the-art verification and synthesis tools, matching or outperforming experts, and that the fixed programs continue to work well across different workloads. As more programmers employ symbolic evaluation to automate verification and synthesis tasks for new domains, SymFix can help them build better tools more easily.

Chapter 4

An expressive pretty printer

4.1 Introduction

General-purpose pretty printers (or, simply, *printers*) are widely used to convert structured data—typically an AST—into human-readable text. In the context of solver-aided programming, printers have been employed for tasks that require human interaction, such as generating synthesized code that must be reviewed by humans, generating code in human-in-the-loop program synthesis framework, and displaying complex values for debugging purposes. These printers take as inputs (1) a document in a pretty printing language (*PPL*), which encodes the structured data along with formatting choices, and (2) a page width limit. Choices in the document can yield (exponentially) many possible layouts. The task of the printers then is to efficiently choose an *optimal* layout from all possible layouts. Existing printers use a variety of built-in optimality objectives. A good objective reflects the informal notion of "prettiness," such as not having an overflow over the page width limit whenever possible while having as few lines as possible.

Different printers make different trade-offs in the *expressiveness* of the PPL, the *optimality* objective, and the *performance*. With the current landscape of pretty printing, programmers

```
text "function append(first,second,third){"
<> nest 4 (
  let f = text "first +" in
  let s = text "second +" in
  let t = text "third" in
  nl <> text "return " <>
  group (nest 4 (f <> nl <> s <> nl <> t))
) <> nl <> text "}"
```

```
1 function append(first,second,third){
2     return first +
3         second +
4         third
5 }
```

```
1 function append(first,second,third){
2     return first + second + third
3 }
```

(a) A document in the traditional PPL and its corresponding layouts. The **nest** construct increments the current indentation level by some specified amount, causing **nl** (newline) to insert indentation spaces. **<>** is the unaligned concatenation operator, which places the right sub-layout after the left sub-layout on the current indentation level. Lastly, the **group** construct creates two formatting choices: one where the sub-layouts are left alone and one where the sub-layouts are flattened by replacing newlines and indentation spaces due to **nl**s in the group with single spaces.

```
text "function append(first,second,third){"
<$>
( let f = text "first +" in
  let s = text "second +" in
  let t = text "third" in
  let sp = text " " in
  let indentation = text "    " in
  let ret = text "return " in
  indentation <+>
  (((ret <+> text "(") <$>
    (indentation <+> (f <$> s <$> t)) <$>
    text ")") <|>
   (ret <+> f <+> sp <+> s <+> sp <+> t)))
<$> text "}"
```

```
1 function append(first,second,third){
2     return (
3         first +
4         second +
5         third
6     )
7 }
```

```
1 function append(first,second,third){
2     return first + second + third
3 }
```

(b) A document in the arbitrary-choice PPL and its corresponding layouts. **<|>** is the arbitrary-choice operator, which per its namesake, creates two formatting choices from layouts by arbitrary sub-document. **<$>** is the vertical concatenation operator, which joins two sub-layouts with a newline. Lastly, **<+>** is the aligned concatenation operator, which joins two sub-layouts horizontally, aligning the whole right sub-layout at the column where it is to be placed in.

Figure 4.1: The traditional and arbitrary-choice PPLs, embedded in the host language OCaml. Colored regions in a document and corresponding layouts indicate the correspondence between the colored sub-documents and the colored sub-layouts. We use the **let** construct to make the documents easier to read, even though it is usually not a part of PPLs. Dotted lines illustrate different page width limits at 22 and 36 characters.

who wish to use expressive features from various PPLs, optimize certain objectives, and achieve practical running time are left with two possibilities. They can write an ad-hoc printer, which is time-consuming and error-prone. Alternatively, they can pick an existing general-purpose printer, potentially giving up some desired features. This is further complicated by some common misunderstandings about the three aspects of these existing printers.

This chapter presents a printer that we call Π_e. It targets Σ_e, a PPL that is strictly more expressive than all published PPLs. This can be shown via our formal framework for reasoning about the expressiveness of PPLs. Π_e is parameterized by a *cost factory*, which enables programmers to specify an optimality objective for Π_e to minimize. The cost factory is versatile. For example, it can express non-linear costs and define concepts such as soft page width limits [104]. As a result, the optimal layout that Π_e chooses can have higher quality compared to existing printers. The time complexity of Π_e is $O(nW^4)$, where n is the size of the document and W is the computation width limit (defined in Section 4.5). This is better than the time complexity of many printers in the literature, and it is improved to $O(nW^3)$ when Π_e is restricted to process documents in some well-known but less expressive PPLs. We prove the correctness of Π_e in the Lean theorem prover [59], ensuring the validity and optimality of the output layout, and demonstrate Π_e's efficiency by evaluating our implementation of Π_e, which we call SnowWhite. We believe these attributes make Π_e not only a good printer by itself, but also a good building block to construct other derived printers.

A survey of printers in the wild To evaluate Π_e, we conducted a broad survey of the literature on pretty printing. Most PPLs, embedded in a host programming language, provide a small set of core constructs that allow programmers to create a document with text, concatenate documents together, set indentation level, and express formatting choices. High-level constructs can then be built on top of the core constructs. The details of these core constructs can differ from PPL to PPL. We found that there are two main schools of PPLs in the wild, which we call the *traditional*

and *arbitrary-choice* PPLs. The traditional PPL centers around manipulation of `nl`s (newlines) and current indentation level, while the arbitrary-choice PPL is characterized by the ability to express arbitrary formatting choices and the use of aligned concatenation to supplant the concept of indentation level. Figure 4.1 illustrate documents in both PPLs that pretty-print the function definition `append` in a hypothetical programming language with slightly different styling.

Expressiveness The literature contains informal claims about the expressiveness of PPLs [96, 21, 75]. We develop two formal notions of expressiveness: the ability to *express layouts* and the ability to *express features*. The former reflects the functionality of a PPL, while the latter reflects the ease of document construction. Using our framework, we can show that neither the traditional PPL nor the arbitrary-choice PPL is more expressive than the other. For example, the set of layouts in Figure 4.1b cannot be expressed by any document in the traditional PPL. This is because all layouts due to a particular document in the traditional PPL must be the same modulo whitespace, but one of the layouts in the figure has an extra pair of parentheses.[1] As another example, the document in Figure 4.1b is awkwardly constructed. It would be more natural to use unaligned concatenation, but the feature cannot be expressed by any combination of features in the arbitrary-choice PPL.[2] To that end, we develop a PPL called Σ_e that is provably strictly more expressive than both the traditional and arbitrary-choice PPLs, facilitating both functionality and ease of document construction.

[1] Languages such as Python require an extra pair of parentheses around an expression that spans multiple lines [87]. Similarly, some styles prefer adding an extra comma (also known as trailing comma) when a function call spans multiple lines [34]. Hence, the ability to express layouts with differing content is desirable.

[2] Different programming language styles prefer different concatenation operators. C-like languages heavily use unaligned concatenation, while aligned concatenation has been used for Haskell, Lisp, R, and Julia. However, there are instances where C-like languages would benefit from aligned concatenation, and Haskell would benefit from unaligned concatenation.

Optimality The optimality objective of a printer indicates what it optimizes for when resolving choices. Most printers targeting the traditional PPL minimizes overflow over the page width limit line-by-line, preferring a longer line when there is no overflow. For example, given the document in Figure 4.1a, the first layout is optimal when the page width limit is 22 (red dotted line), while the second layout is optimal when the page width limit is 36 (green dotted line). Contrary to prior claims [96, 21], we discovered that this strategy guarantees neither the absence of overflow whenever possible nor the minimality of the number of lines. In contrast, most printers targeting the arbitrary-choice PPL minimizes the number of lines among layouts with no overflow. However, they *error* when all possible layouts have an overflow, resulting in a poor user experience (e.g., when the width limit is 22 in Figure 4.1b). Recognizing that unavoidable overflows do occur in practice, we introduce the concept of a *cost factory*. The factory allows programmers to choose a desired objective permitted by its interface, including an objective that tolerates overflow gracefully.

Performance Printing proceeds in two phases: resolving choices and rendering the optimal choice to text (although many printers fuse these two phases together). Time complexity of printers is best measured against the resolving phase[3], and it is usually specified with two parameters: the size of the document n and the width limit W, with the preference that the time complexity be polynomial in W and linear in n. Most printers in the literature leave their time complexity unanalyzed, instead opting to show experimental results that their implementations are efficient in practice. We analyze these printers and demonstrate documents that trigger worse than linear time behavior (in n) on some printers. Further complication arises in printers with the arbitrary choice feature, which gives rise to documents that are structured as DAGs as opposed to trees. We show that many printers that treat the input document as a tree suffer from a combinatorial explosion as the DAG structure is unfolded, resulting in exponential time complexity. With a

[3]This formulation allows us to talk about "linear-time" printers, even though there are, e.g., documents whose size is $O(n)$, but its optimal layout has $O(n^2)$ characters.

combination of proof and experimental results, we show that the time complexity of Π_e is linear in the DAG size of the document and that it runs fast in practice.

In summary, this chapter makes the following contributions:

- A new PPL called Σ_e that is strictly more expressive than all published PPLs. Constructs in Σ_e are not new, but packaging them all in a single PPL has never been done before.
- A printer Π_e targeting Σ_e that utilizes a *cost factory* to allow a variety of optimality objectives.
- A proof of correctness for Π_e, formalized in the Lean theorem prover. To our knowledge, this is the first time that a printer has been formally verified.
- A framework to formally reason about the expressiveness of PPLs.
- A survey of printers and an analysis that dispels common misunderstandings about them.
- An implementation of Π_e, SNOWWHITE, and an evaluation that shows its effectiveness.

The rest of this chapter is structured as follows. Section 4.2 surveys the related work. Section 4.3 presents the semantics of Σ_e. Section 4.4 introduces a framework to reason about the expressiveness of PPLs. Section 4.5 presents Π_e and its analysis. Section 4.6 discusses SNOWWHITE, an implementation of Π_e. Section 4.7 presents an evaluation of SNOWWHITE that demonstrates its effectiveness. Section 4.9 shows applications of Π_e in solver-aided programming. Lastly, Section 4.10 concludes the chapter.

4.2 Related work

To understand the trade-off space of printer designs, we conduct a comprehensive analysis of related work in the literature. This section provides our analysis of the printers, grouped by the

Table 4.1: A comparison of existing printers. n and $\hat{n}$ are the DAG size and tree size of the input document (where $\hat{n}$ in the worst case is exponential in n). W is the width limit.

	Expressiveness		Optimality	Performance
Printer	Choice	Concatenation	Minimization objective	Time complexity
Oppen [67]	Group	Unaligned	Lexicographic overflow	$O(n)$
Hughes [46]	Group	Aligned	Lexicographic overflow	$O(n^2)$
Wadler [96]	Group	Unaligned	Lexicographic overflow	$O(n^2)$
Leijen [55]	Group	Both	Lexicographic overflow	$O(n^2)$
Chitil [21]	Group	Unaligned	Lexicographic overflow	$O(n)$
Kiselyov et al. [52]	Group	Unaligned	Lexicographic overflow	$O(n)$
Swierstra et al. [85]	Arbitrary	Aligned	Height†	Exp. in n
Podkopaev and Boulytchev [75]	Arbitrary	Aligned	Height†	$O(\hat{n}W^4)$
Yelland [104]	Arbitrary	Aligned	Linear cost	$O(\hat{n}^{3/2})$
Bernardy [8]	Arbitrary	Aligned	Height†	$O(nW^6)$
Π_e	Both	Both	Cost (from the cost factory)	$O(nW^4)$
Π_e (aligned only)	Both	Aligned	Cost (from the cost factory)	$O(nW^3)$

† only consider layouts without an overflow over W.

$$
\begin{array}{lll}
d \in \mathcal{D} ::= & \texttt{text}\ s & \text{text} \\
\mid & \texttt{nl} & \text{newline} \\
\mid & d\ \texttt{<>}\ d & \text{unaligned concatenation} \\
\mid & \texttt{nest}\ n\ d & \text{increase indentation level} \\
\mid & \texttt{group}\ d & \text{create two choices where one flattens layouts}
\end{array}
$$

(a) A variant of traditional PPL from Wadler [96].

$$
\begin{array}{lll}
d \in \mathcal{D} ::= & \texttt{text}\ s & \text{text} \\
\mid & d_a\ \texttt{<+>}\ d_b & \text{aligned concatenation} \\
\mid & d_a\ \texttt{<\$>}\ d_b & \text{vertical concatenation} \\
\mid & d_a\ \texttt{<|>}\ d_b & \text{create two arbitrary choices}
\end{array}
$$

(b) A variant of arbitrary-choice PPL from Podkopaev and Boulytchev [75].

Figure 4.2: A comparison between the traditional and arbitrary-choice PPLs. s denotes a string without newline, and n denotes a natural number.

```
group (text "AAA" <> nl) <>
nest 5 (group (text "B" <> nl <>
               text "B" <> nl <> text "B"))
```

```
1 AAA
2 B B B
```

```
1 AAA B
2     B
3     B
```

Figure 4.3: A document in the traditional PPL and two of their corresponding layouts. Under the width limit of 5, the first layout is optimal—it does not overflow and occupies minimal number of lines. In contrast, the second layout, which is produced by Wadler's printer, overflows and does not occupy minimal number of lines.

```
let rec quadratic (n : int): doc =
  if n = 0 then text "line"
  else group (quadratic (n - 1) <> nl <> text "line")
```

Figure 4.4: The function `quadratic` generates a document of size $O(n)$ that Wadler's algorithm takes $O(n^2)$ to print at any width limit, due to repeated flattening.

expressiveness of their public interface[4]. The summary is presented in Table 4.1. We then compare and contrast our printer Π_e against them.

4.2.1 Traditional printers

Pretty printing has a long history. Oppen [67] first introduced a general-purpose printer, written in the imperative style. Oppen pioneered the PPL that we call the traditional PPL, shown in Figure 4.2a. Instead of representing an input document as a tree, as commonly done in subsequent work, Oppen represents the document as a stream of "instruction tokens." The algorithm's time complexity is $O(n)$, where n is the length of the stream. Furthermore, the algorithm is *bounded*, requiring a limited look-ahead into the stream. As with other printers in the family, the printer greedily minimizes overflow over the page width limit, which neither avoids overflow whenever possible nor minimizes number of lines, as discussed in the paper.

[4]In practice, printers include extensions that increase their expressiveness. A printer may even have different expressiveness across different versions. This section focuses on the core features of these printers as specified in their publications.

Wadler [96] designed a printer that targets the traditional PPL. It is used in many real world applications, such as an industrial code formatter [77], and as a basis for much pretty printing research [21, 52]. The printer aims to be a rewrite of Oppen's printer using the functional style employed by Hughes (described later). The printer is claimed [96, 21] to produce an output layout that does not exceed the width limit whenever possible, and minimizes the number of lines. However, this is not the case, as shown in Figure 4.3. The time complexity of the printer is claimed to be $O(n)$ where n is the size of document [96], but it is in fact $O(n^2)$ in the worst case, as demonstrated in Figure 4.4, although this worst case behavior is unlikely to occur in practice.

Chitil [21] improved Wadler's printer so that it is as efficient as Oppen's, $O(n)$, by using lazy dequeues. Kiselyov et al. [52] similarly improved Wadler's printer via their generator framework.

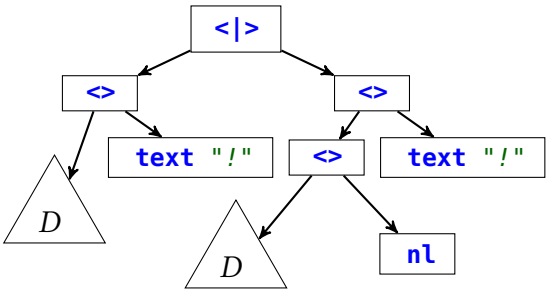
Compared to traditional printers, Π_e is more expressive as it allows arbitrary choices and aligned concatenation. Furthermore, Π_e can produce an output layout that minimizes number of lines when the output layout does not exceed the page width limit, and does not exceed the page width limit whenever possible. The tradeoff is that Π_e is less space efficient and slower than traditional printers. The space complexity of traditional printers is sub-linear in the size of document, which was especially important decades ago when memory is scarce. The space complexity of Π_e is $O(nW^3)$ in the worst case (or $O(nW^2)$ when targeting some PPLs). We find that on modern machines, the added memory consumption and performance overhead are rarely an issue in practice (Section 4.7).

4.2.2 Arbitrary-choice printers

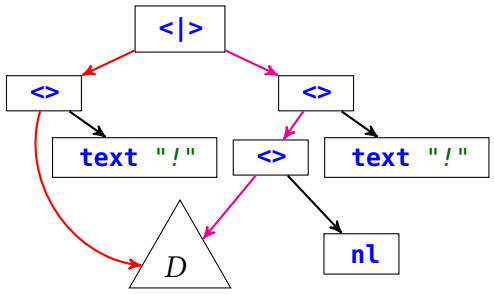
Azero Alcocer and Swierstra [2] introduced a printer that supports arbitrary choices with aligned concatenation, starting the line of work that targets the arbitrary-choice PPL, shown in Figure 4.2b. The printer's optimality objective is to avoid overflow whenever possible and produce a minimal

```
let shared := D in (shared <> text "!") <|> ((shared <> nl) <> text "!")
```

(a) A document that encodes (at least) two possible layouts. D is an arbitrary sub-document.



(b) A tree representation of Figure 4.5a. D contributes to the size twice.



(c) A DAG representation of Figure 4.5a. D contributes to the size only once.

Figure 4.5: An example document that shows the importance of treating document as a DAG rather than a tree. The red and pink paths illustrate that the DAG is properly shared, as will be discussed in Section 4.5.5.

number of lines. However, it does not have the ability to cope with unavoidable overflow. This printer was soon superseded by Swierstra et al. [85], which improves its performance via heuristics and adds the capability to *share* a sub-document across choices by deeply embedding the (equivalent of the) **let** construct in the PPL. As a result, the later printer can process documents that are structured as DAGs rather than trees, as shown in Figure 4.5. Nonetheless, the time complexity of both printers is exponential in n [75].

Podkopaev and Boulytchev [75] improved upon Swierstra et al.'s work by formulating the problem as dynamic programming. The formulation fixes the exponential blowup in the prior work, but treats the document as a tree, making its time complexity $O(\hat{n}W^4)$, where $\hat{n}$ is the tree size of the document, which could be exponentially larger than its DAG size. The paper acknowledges the problem and surmises that memoization may be able to address it.

The paper by Bernardy [8] is the main inspiration for our work. The printer uses Pareto frontiers to find an optimal layout. By shallowly embedding the PPL (in Haskell), computations on sub-documents are effectively shared for free. However, as presented in the paper, the printer

```
text "xxxxxx" <$>
((text "aaa" <+> text "bbb") <|>
 (text "aaa" <$> text "bbb"))
```

```
1 xxxxxx
2 aaa
3 bbb
```

```
1 xxxxxx
2 aaabbb
```

Figure 4.6: A document in the arbitrary-choice PPL and two of their corresponding layouts. Under the width limit of 5, the first layout minimally overflows. In contrast, the second layout, which is produced by Bernardy's practical implementation, overflows than necessary.

requires the page width limit to be hard-coded. In the actual implementation [7], the page width limit is customizable, accomplished by threading the value through functions. But this change destroys the shared computations, leading to exponential running time. Compared to Podkopaev and Boulytchev [75]'s work, Bernardy [8]'s approach can exploit sparseness to improve practical efficiency, but the use of an inefficient algorithm makes the time complexity of the printer $O(nW^6)$ in the worst case. While the paper does not handle unavoidable overflow, the implementation does by automatically scaling up the page width limit (or equivalently, minimizing the maximum overflow). This, however, allows avoidable overflow elsewhere, as shown in Figure 4.6, which is undesirable. Later on, Bernardy abandoned the arbitrary-choice operator, noting that it could trigger the exponential behavior [6].

Yelland [104] similarly targeted the arbitrary-choice PPL. However, the paper took a very different approach. The core printer restricts the use of aligned concatenation by requiring the left sub-document to be a `text` syntactically. This restriction allows the core printer to utilize the concept of "piecewise linear cost function" to seemingly boost the performance. To achieve the expressiveness of the arbitrary-choice PPL, the printer employs rewriting rules to transform the original document into the restricted document. While the printer is careful to avoid exponential blowup by sharing documents in the resulting restricted document, it does not necessarily preserve the sharing structure of the original document, as demonstrated in Figure 4.7. Compound this with the lack of computation width limit, the number of piecewise linear cost functions under consideration could be as large as $O(\hat{n}^{1/2})$, making the time complexity $O(\hat{n}^{3/2})$ in total, as shown in Figure 4.8.

```
let rec mk (n : int): doc =
  if n = 0 then text "X" <|> text "XX"
  else let subdoc = mk (n - 1) in (chr n <> subdoc <> chr n) <|> subdoc
```

(a) The function mk generates a document whose DAG size is $O(n)$. $\mathrm{chr}(n)$ denotes a **text** whose content is a string of length one that contains the nth character.

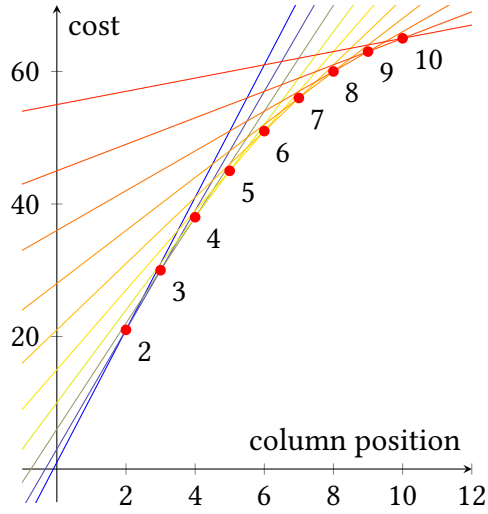
$$
\begin{aligned}
C'[D_n, Z_{[]}] &= C'[\mathrm{chr}(n) \texttt{<>} D_{n-1} \texttt{<>} \mathrm{chr}(n), Z_{[]}] \texttt{<|>} C'[D_{n-1}, Z_{[]}] \\
&= C'[\mathrm{chr}(n), C'[D_{n-1}, C'[\mathrm{chr}(n), Z_{[]}]]] \texttt{<|>} C'[D_{n-1}, Z_{[]}] \\
&= C'[\mathrm{chr}(n), C'[D_{n-1}, Z_{[n]}]] \texttt{<|>} C'[D_{n-1}, Z_{[]}] \\
C'[D_{n-1}, Z_{[]}] &= C'[\mathrm{chr}(n-1), C'[D_{n-2}, Z_{[n-1]}]] \texttt{<|>} C'[D_{n-2}, Z_{[]}] \\
C'[D_{n-1}, Z_{[n]}] &= C'[\mathrm{chr}(n-1), C'[D_{n-2}, C'[\mathrm{chr}(n-1), Z_{[n]}]]] \texttt{<|>} C'[D_{n-2}, Z_{[n]}] \\
&= C'[\mathrm{chr}(n-1), C'[D_{n-2}, Z_{[n-1,n]}]] \texttt{<|>} C'[D_{n-2}, Z_{[n]}]
\end{aligned}
$$

(b) Let D_n denote $\mathrm{mk}(n)$. Yelland's C' function would transform the original document D_n into a restricted document where every aligned concatenation has a **text** as its left subdocument. However, the above derivation shows that the transformation has a combinatorial explosion. Define $Z_{[]}$ to be ■ in Yelland's paper and $Z_{[x,x_1,\ldots,x_n]}$ to be $C'[\mathrm{chr}(x), Z_{[x_1,\ldots,x_n]}]$. The derivation shows that D_{n-k} is recursively transformed in 2^k different contexts.

Figure 4.7: A family of documents that illustrates how the transformation C' in Yelland's algorithm does not necessarily preserve the sharing structure in the original document.

```
(* make an empty document of size n; n >= 1 *)
let rec make_dummy (n : int): doc =
  if n = 1 then text ""
  else text "" <+> make_dummy (n - 1)

(* make n lines; n >= 1 *)
let rec make_lines (n : int): doc =
  if n = 1 then text ""
  else text "" <$> make_lines (n - 1)

(* nth triangle number *)
let tri (n : int): int = n * (n + 1) / 2

let make_choices (k : int): doc =
  let rec loop (i : int): doc =
    let doc =
      (make_lines i) <+>
        text (String.make (tri (k - i + 1)) 'a')
    in if i = 1 then doc else doc <|> loop (i - 1)
  in loop k

let rec example (k : int): doc =
  let dummy = make_dummy (k * k) in
  let giant = make_choices k in
  dummy <+> giant
```

(a) The function `example` produces a document that triggers the worst-case time complexity of Yelland's algorithm (that we are aware of). For a fixed k, `giant` is a document with k choices, where the i-th choice has i lines and $\mathrm{tri}(k-i+1)$ characters (tri is the triangle number function). Thus, its document tree size is $O(k^2)$. By concatenating `giant` with `dummy`, which is an "empty" document of size $O(k^2)$, the total document tree size is still $O(k^2)$. `giant` is designed so that it has k segmented linear cost functions. Thus, the aligned concatenation of `dummy` and `giant` takes $O(k^3)$. By normalizing the document size to $\hat{n}$, we obtain that the time complexity of the printer is $O(\hat{n}^{3/2})$.



(b) A plot of the piecewise linear cost function (lines along the red dots) for `giant` in Figure 4.8a with $k = 10$. The x-axis is column positions that `giant` will be placed. The y-axis is costs due to the placement. The plot consists of $O(k)$ segmented cost functions, where each segment is a linear function. For simplicity, we assume that (1) the page width limit is 0; (2) there is no cost for newlines; and (3) the cost for each character past the page width limit is 1. Let $\overline{d}_i$ be the i-th choice in `giant`. The cost function for $\overline{d}_i$ then is $C_{\overline{d}_i}(c) = ic + \mathrm{tri}(k-i+1)$. These cost functions intersect at $c = 2, \ldots, k$. Thus, the cost function for `giant` is unable to prune any segments away.

Figure 4.8: In Yelland's algorithm, every choiceless document (in the arbitrary-choice PPL) $\overline{d}$ has an associated *piecewise linear* cost function $C_{\overline{d}}$, where $C_{\overline{d}}(c)$ determines the cost of placing $\overline{d}$ at the column position c. A general document d similarly has an associated piecewise linear cost function C_d, which takes the minimum of the cost functions from all choiceless documents d generates. The algorithm appears to be efficient at first glance, since taking the minimum can prune away many segmented linear cost functions. However, we are able to construct a document `giant` of size $O(\hat{n})$ whose cost function has $O(\sqrt{\hat{n}})$ segmented linear cost functions, where $\hat{n}$ is the tree size of the document. As the time complexity of the printer is $O(\hat{n}M)$ where M is the maximum number of piecewise linear cost functions in a cost function, we obtain $O(\hat{n}^{3/2})$.

Another aspect to consider is the printer's optimality objective, which is restricted to minimizing a linear combination of quantities like the number of lines and overflow. Hence, the printer will not technically avoid overflow whenever possible (although the overflow coefficient can be made arbitrarily large to arbitrarily discourage overflow). On the other hand, this optimality objective can support unique features, such as incorporating the costs due to multiple soft page width limits.

Compared to arbitrary-choice printers, Π_e is more expressive as it allows unaligned concatenation. Π_e is also asymptotically faster than most arbitrary-choice printers, as it treats a document as a DAG rather than a tree. Like Yelland's printer, for each layout under consideration, Π_e keeps track of two quantities: cost and last line length. This is different from most printers in the family which keep track of three quantities: height, width, and last width. The dimension reduction further makes Π_e more efficient. The concept of cost also allows Π_e to decouple the page width limit and computation width limit, which allows graceful overflow handling. Π_e, unlike Yelland's printer, is parameterized by a cost factory, which supports a variety of optimality objectives without requiring a modification to the core printer. This includes not only the linear optimality objectives that Yelland's printer supports, but also non-linear optimality objectives that can properly avoid overflows.

4.2.3 Other printers

Hughes [46] brought a general-purpose printer to the functional world. The printer pioneers using combinators to construct a document, which is now a standard practice. The printer targets a PPL that is neither the traditional nor arbitrary-choice PPL, but somewhere in-between. In particular, it only supports aligned concatenation and does not provide the arbitrary-choice operator in the public interface. The printer is more similar to the traditional printers in how it makes choices greedily, which minimizes neither overflow nor number of lines. The combination of greedy choice making and aligned concatenation makes some documents print very poorly [8]. Furthermore,

Peyton-Jones [72] identified quadratic time complexity in the printer.

Leijen [55] implemented Wadler's printer in Haskell and added support for aligned concatenation via the inclusion of `align`, becoming the first printer that supports both aligned and unaligned concatenation. However, similar to Hughes' printer, the printer can produce very poor output [8].

4.3 The syntax and semantics of Σ_e

This section presents Σ_e, an expressive PPL. We begin this section by describing *layouts*, which are the textual outputs. Then, we describe the syntax of Σ_e and its informal semantics, which is determined by the evaluation of a document in Σ_e to layouts. Lastly, we formally describe the semantics of Σ_e.

4.3.1 Layout

A *layout* $l \in \mathcal{L}$ is a textual output. In our work, we represent a layout as a non-empty, finite list of lines (implicitly joined by newlines), where each line is a string without the newline character. This allows us to easily reason about the number of lines and the length of each line.

A layout can be *placed* at a column position c. This placing puts the first line of the layout at the column position c, and puts the rest of the lines at the column position 0. While the final layout to present to users is placed at the column position 0, a (sub)layout may be placed at a non-zeroth column position during the rendering process.

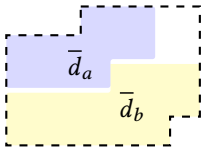
Example 4.1 *The following illustration displays a layout* `["[|", " 42,", " 1729,", "|]"]` *placed at the column position* 5. *The pink area shows the general shape of layout "boundaries", where the first line could be "indented" due to the placing on a non-zeroth column position, and the last line covers its length exactly. The shape will be useful to understand how layouts are composed together.*

Document $d \in \mathcal{D}_e ::=$ **text** s | **nl** | d **<>** d | **nest** n d | **align** d | **flatten** d | d **<|>** d

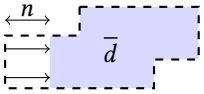
String without newline $s, t, \ldots \in \text{Str}$

Natural number $n \in \mathbb{N}$

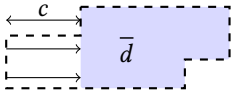
Figure 4.9: Syntax for Σ_e



(a) An unaligned concatenation of $\overline{d}_a$ and $\overline{d}_b$

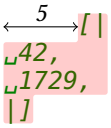


(b) A **nest** of $\overline{d}$ to increase the indentation level by n



(c) An **align** of $\overline{d}$ when placed at the column position c

Figure 4.10: Illustrations of constructs in Σ_e. The area with dashed borders is the resulting boundaries.



4.3.2 The syntax and the informal semantics of Σ_e

Figure 4.9 shows the syntax of document in Σ_e. Each construct is from either the traditional or the arbitrary-choice PPLs, except the flatten construct **flatten** (which is internally used in Wadler [96]'s printer) and the align construct **align** (which is from Leijen [55]'s printer). Similar to layouts, documents can be placed at a column position. As traditionally done, we for now ignore the choice operator. A document without the arbitrary-choice operator is called a *choiceless document*, which can be *rendered* to a single layout. We denote a choiceless document with $\overline{d} \in \overline{\mathcal{D}_e}$. The informal semantics of choiceless document are as follows:

text s renders to a layout with a single line s.

nl normally renders to a layout with two lines. The first line is empty, and the second line consists of i spaces where i is the *indentation level*. **nl** interacts with *flattening*, which reduces it to just a single space.

$\overline{d}_a$ `<>` $\overline{d}_b$	renders to a layout that concatenates the layout of $\overline{d}_a$ and the layout of $\overline{d}_b$ without alignment. The rendering of $\overline{d}_b$ is dependent on the rendering of $\overline{d}_a$, because $\overline{d}_b$ will be placed at a column position after the last character of $\overline{d}_a$. Figure 4.10a illustrates this.
`nest` n $\overline{d}$	renders to a layout like $\overline{d}$, but with the indentation level *increased* by n relative to the current indentation level. Figure 4.10b roughly illustrates this.
`align` $\overline{d}$	renders to a layout like $\overline{d}$, but with the indentation level *set* (not relatively increased) to the column position that `align` $\overline{d}$ is being placed in. Figure 4.10c roughly illustrates this.
`flatten` $\overline{d}$	renders to a layout just like $\overline{d}$, but with all newlines and indentation spaces due to `nl`s flattened to single spaces.

While Figure 4.10 provides a rough illustration that should be helpful to understand the semantics of choiceless document, it could be misleading.

Example 4.2 *The document* `text "a" <> (nest 42 (align (text "b" <> nl <> text "c")))` *is rendered to* `["ab", " c"]`. *The nesting doesn't visibly increase the indentation level by 42, because the alignment on the inner document overrides the indentation level. This example shows the importance of the indentation level, and why it must be specifically tracked.*

This concludes our informal description of how a choiceless document renders to a layout. General documents, by contrast, can contain the arbitrary-choice operator `<|>`, which encodes layouts from two sub-documents into one document. Thus, unlike choiceless documents, which render to a single layout, general documents will *evaluate* to a non-empty, finite set of layouts. Our approach is to first *widen* a document into a set of choiceless documents, then render each choiceless document in the set to produce a set of layouts.

$$\textsc{Text}\ \frac{}{\langle \texttt{text}\ s, c, i, f\rangle \Downarrow_{\mathcal{R}} [s]} \qquad \textsc{Flatten}\ \frac{\langle \overline{d}, c, i, \top\rangle \Downarrow_{\mathcal{R}} [s]}{\langle \texttt{flatten}\ \overline{d}, c, i, f\rangle \Downarrow_{\mathcal{R}} [s]}$$

$$\textsc{LineNoFlatten}\ \frac{}{\langle \texttt{nl}, c, i, \bot\rangle \Downarrow_{\mathcal{R}} [\epsilon, \texttt{" "} \times i]} \qquad \textsc{LineFlatten}\ \frac{}{\langle \texttt{nl}, c, i, \top\rangle \Downarrow_{\mathcal{R}} [\texttt{" "}]}$$

$$\textsc{ConcatOne}\ \frac{\langle \overline{d}_a, c, i, f\rangle \Downarrow_{\mathcal{R}} [s] \quad \langle \overline{d}_b, c + |s|, i, f\rangle \Downarrow_{\mathcal{R}} [t, t_1, \ldots, t_n]}{\langle \overline{d}_a \texttt{<>}\ \overline{d}_b, c, i, f\rangle \Downarrow_{\mathcal{R}} [s +\!\!+ t, t_1, \ldots, t_n]} \qquad \textsc{ConcatMult}\ \frac{\langle \overline{d}_a, c, i, f\rangle \Downarrow_{\mathcal{R}} [s_1, \ldots^+, s_n, s] \quad \langle \overline{d}_b, |s|, i, f\rangle \Downarrow_{\mathcal{R}} [t, t_1, \ldots, t_m]}{\langle \overline{d}_a \texttt{<>}\ \overline{d}_b, c, i, f\rangle \Downarrow_{\mathcal{R}} [s_1, \ldots^+, s_n, s +\!\!+ t, t_1, \ldots, t_m]}$$

$$\textsc{Nest}\ \frac{\langle \overline{d}, c, i + n, f\rangle \Downarrow_{\mathcal{R}} [s_1, \ldots^+, s_m]}{\langle \texttt{nest}\ n\ \overline{d}, c, i, f\rangle \Downarrow_{\mathcal{R}} [s_1, \ldots^+, s_m]} \qquad \textsc{Align}\ \frac{\langle \overline{d}, c, c, f\rangle \Downarrow_{\mathcal{R}} [s_1, \ldots^+, s_n]}{\langle \texttt{align}\ \overline{d}, c, i, f\rangle \Downarrow_{\mathcal{R}} [s_1, \ldots^+, s_n]}$$

$$\textsc{TextWiden}\ \frac{}{\texttt{text}\ s \Downarrow_{\mathcal{W}} \{\texttt{text}\ s\}} \qquad \textsc{LineWiden}\ \frac{}{\texttt{nl} \Downarrow_{\mathcal{W}} \{\texttt{nl}\}}$$

$$\textsc{ConcatWiden}\ \frac{d_a \Downarrow_{\mathcal{W}} \overline{D}_a \quad d_b \Downarrow_{\mathcal{W}} \overline{D}_b}{d_a \texttt{<>}\ d_b \Downarrow_{\mathcal{W}} \{\overline{d}_a \texttt{<>}\ \overline{d}_b \mid \overline{d}_a \in \overline{D}_a, \overline{d}_b \in \overline{D}_b\}}$$

$$\textsc{NestWiden}\ \frac{d \Downarrow_{\mathcal{W}} \overline{D}}{\texttt{nest}\ n\ d \Downarrow_{\mathcal{W}} \{\texttt{nest}\ n\ \overline{d} \mid \overline{d} \in \overline{D}\}} \qquad \textsc{AlignWiden}\ \frac{d \Downarrow_{\mathcal{W}} \overline{D}}{\texttt{align}\ d \Downarrow_{\mathcal{W}} \{\texttt{align}\ \overline{d} \mid \overline{d} \in \overline{D}\}}$$

$$\textsc{FlattenWiden}\ \frac{d \Downarrow_{\mathcal{W}} \overline{D}}{\texttt{flatten}\ d \Downarrow_{\mathcal{W}} \{\texttt{flatten}\ \overline{d} \mid \overline{d} \in \overline{D}\}} \qquad \textsc{UnionWiden}\ \frac{d_a \Downarrow_{\mathcal{W}} \overline{D}_\alpha \quad d_b \Downarrow_{\mathcal{W}} \overline{D}_\beta}{d_a \texttt{<|>}\ d_b \Downarrow_{\mathcal{W}} \overline{D}_\alpha \cup \overline{D}_\beta}$$

Figure 4.11: Semantics for Σ_e. "ϵ" is the empty string. "$s \times i$" is the notation for replicating the string s for i times. "$s +\!\!+ t$" is a string concatenation of s and t. Lastly, "$s_1, \ldots, s_n$" and "$s_1, \ldots^+, s_n$" indicate n lines, where $n \geq 0$ and $n \geq 1$ respectively.

4.3.3 The formal semantics of Σ_e

The formal semantics of Σ_e, consisting of two relations, is given in Figure 4.11. The judgment $\langle \overline{d}, c, i, f\rangle \Downarrow_{\mathcal{R}} l$ states that the choiceless document $\overline{d} \in \overline{\mathcal{D}_e}$ placed at column position $c \in \mathbb{N}$ with indentation level $i \in \mathbb{N}$ and flattening mode $f \in \mathbb{B}$, will render to the layout $l \in \mathcal{L}$. The flattening mode f, which indicates whether newlines should be replaced with spaces, can be either on ($\top$) or off ($\bot$). Another judgment $d \Downarrow_{\mathcal{W}} \overline{D}$ states that a document $d \in \mathcal{D}_e$ is widened to a finite, non-empty set of choiceless documents $\overline{D} \in 2^{\overline{\mathcal{D}_e}}$. We sometimes call a combination of c and i (and possibly f) a *printing context*. Now, we elaborate some interesting rules in the figure.

Rendering text The Text rule states that the rendering of a text placement `text` s contains a layout with a single line of the text s. The printing context is completely ignored: indentation level and flattening mode do not affect the rendering of text, and we have already assumed that s will be placed at the column position.

Rendering newline When the flattening mode is off, the LineNoFlatten rule states that the rendering of `nl` results in a layout with two lines. The first line is empty, while the second line is indented by i spaces. On the other hand, when the flattening mode is on, the LineFlatten rule states that the rendering of the newline results in a layout with a single line of a single space.

Rendering unaligned concatenation In the rendering of $\overline{d}_a$ `<>` $\overline{d}_b$, we recursively render $\overline{d}_a$ and $\overline{d}_b$, but the rendering of $\overline{d}_b$ is dependent on the rendering of $\overline{d}_a$. Let l_a be the rendering result of $\overline{d}_a$. The ConcatOne rule handles the case where l_a has a single line, and the ConcatMult rule handles the case where l_a has multiple lines.

- If l_a has only a single line s, the column position of $\overline{d}_b$'s rendering needs to be after the string s is placed, i.e. at $c + |s|$. In such case, let l_b be the rendering result of $\overline{d}_b$. The first line of the resulting layout is the concatenation of s and the first line of l_b. The rest of the lines are from the rest of l_b.
- On the other hand, if l_a has multiple lines, the column position of d_b's rendering is simply the column position after the last line is placed. In such case, let l_b be the rendering result of $\overline{d}_b$, the resulting layout contains all but the last line of l_a, a concatenation of the last line of l_a and the first line of l_b, and the rest of l_b.

Widening choice The UnionWiden rule states that the widening of d_a `<|>` d_b is the union of widen d_a and widen d_b

Both $\Downarrow_{\mathcal{R}}$ and $\Downarrow_{\mathcal{W}}$ are deterministic and total. Thus, we can define $\text{eval}_e(d) = \{l : \langle \overline{d}, 0, 0, \bot \rangle \Downarrow_{\mathcal{R}} l, \overline{d} \in \overline{D}, d \Downarrow_{\mathcal{W}} \overline{D}\}$ as the evaluation function for Σ_e, which consumes a document, widens it, and produces a set of layouts.

4.4 A framework to reason about expressiveness

In previous sections, we informally made claims about expressiveness of PPLs. This section presents a framework to formally reason about it, based on two notions: *functional completeness* and *definability*. We first define the semantics of the traditional and arbitrary-choice PPLs. Then, we define our framework, and show that Σ_e is strictly more expressive than both the traditional and arbitrary-choice PPLs.

4.4.1 The extended semantics

To reason about the traditional and arbitrary-choice PPLs, we need to precisely define their semantics. To do so, we construct a PPL Σ_{all} that contains all constructs from Σ_e, traditional, and arbitrary-choice PPLs by extending Figure 4.11 with the rules below (with the straightforward widening rules). Note that we follow Wadler [96]'s approach by treating `group` d as a syntactic sugar for d `<|> flatten` d. As `<|>` and `flatten` are already in Σ_{all}, we do not need to adjust anything further.

$$\textsc{VertConcatNoFlatten}\ \frac{\langle \overline{d}_a, c, i, \bot \rangle \Downarrow_{\mathcal{R}} [s_1, \ldots^+, s_n] \quad \langle \overline{d}_b, i, i, \bot \rangle \Downarrow_{\mathcal{R}} [t_1, \ldots^+, t_m]}{\langle \overline{d}_a \texttt{ <\$> } \overline{d}_b, c, i, \bot \rangle \Downarrow_{\mathcal{R}} [s_1, \ldots^+, s_n, t_1, \ldots^+, t_m]}$$

$$\textsc{VertConcatFlatten}\ \frac{\langle \overline{d}_a, c, i, \top \rangle \Downarrow_{\mathcal{R}} [s] \quad \langle \overline{d}_b, c + 1 + |s|, i, \top \rangle \Downarrow_{\mathcal{R}} [t]}{\langle \overline{d}_a \texttt{ <\$> } \overline{d}_b, c, i, \top \rangle \Downarrow_{\mathcal{R}} [s +\!\!+ \texttt{" "} +\!\!+ t]}$$

$$\textsc{AlignedConcatOne}\ \frac{\langle \overline{d}_a, c, i, f \rangle \Downarrow_{\mathcal{R}} [s] \quad \langle \overline{d}_b, c + |s|, c + |s|, f \rangle \Downarrow_{\mathcal{R}} [t, t_1, \ldots, t_n]}{\langle \overline{d}_a \texttt{ <+> } \overline{d}_b, c, i, f \rangle \Downarrow_{\mathcal{R}} [s +\!\!+ t, t_1, \ldots, t_n]}$$

$$\textsc{AlignedConcatMult}\ \frac{\langle \overline{d}_a, c, i, f\rangle \Downarrow_{\mathcal{R}} [s_1, \ldots^+, s_n, s] \quad \langle \overline{d}_b, |s|, |s|, f\rangle \Downarrow_{\mathcal{R}} [t, t_1, \ldots, t_m]}{\langle \overline{d}_a \texttt{<+>} \overline{d}_b, c, i, f\rangle \Downarrow_{\mathcal{R}} [s_1, \ldots^+, s_n, s +\!\!+ t, t_1, \ldots, t_m]}$$

The semantics of the traditional and arbitrary-choice PPLs are then the restricted semantics of Σ_{all} that only allows constructs from the traditional and the arbitrary-choice PPLs, respectively. Throughout this section, we will assume that any PPL is similarly a sublanguage of Σ_{all}, whose semantics is well-defined and consistent with Σ_{all}.

The extended semantics are still deterministic and total. However, it is worth noting that there are many different ways to specify rules in a way that is consistent with the intended semantics of the arbitrary-choice PPL. For instance, an invariant in the arbitrary-choice PPL is that $c = i$ throughout the rendering process. As a result, we could substitute the VertConcatNoFlatten rule with VertConcatNoFlatten*, which is like VertConcatNoFlatten but with a modification to use $\langle \overline{d}_b, c, c, \bot\rangle \Downarrow_{\mathcal{R}} [t_1, \ldots^+, t_m]$ as a premise instead of $\langle \overline{d}_b, i, i, \bot\rangle \Downarrow_{\mathcal{R}} [t_1, \ldots^+, t_m]$, without changing the semantics of the arbitrary-choice PPL. This change could affect the semantics of Σ_{all} and subsequent theorems in this section. We pick VertConcatNoFlatten over VertConcatNoFlatten* because it seemingly integrates better with other features in Σ_{all}.

4.4.2 Functional completeness

In Section 4.1, we claimed that the traditional PPL cannot express the two layouts in Figure 4.1b, as one layout has an extra pair of parentheses. The question that we may want to ask in general then is, given a PPL Σ and a non-empty set of layouts L, is it possible to construct a document in Σ that evaluates to L? This motivates us to define the notion of functional completeness for PPLs.

Definition 4.1 *A PPL Σ with an evaluation function* $\text{eval}(\cdot)$ *is* functionally complete *if for any non-empty set of layouts L, there exists a document d in Σ such that* $\text{eval}(d) = L$.

With this definition, we can formally reason about some PPLs that we have previously seen.

Lemma 4.1 *The arbitrary-choice and Σ_e PPLs are functionally complete.*

Proof sketch 4.1 *For the arbitrary-choice PPL with the evaluation function* $\mathsf{eval}(\cdot)$*, let L be any non-empty set of layouts. For each $l_i \in L$ where $l_i = [s^i_1, \ldots, s^i_{|l_i|}]$, let d_i be* `text` s^i_1 `<$>` $\ldots$ `<$>` `text` $s^i_{|l_i|}$*. Finally, we construct d to be d_1* `<|>` $\ldots$ `<|>` $d_{|L|}$*. We can see that* $\mathsf{eval}(d) = L$*. The proof for Σ_e PPL is similar, but we would use* `nl` *and* `<>` *instead of* `<$>`*, where d_a* `<$>` *d_b are replaced by d_a* `<> nl <>` *d_b*

Lemma 4.2 *The traditional PPL is* not *functionally complete.*

Proof sketch 4.2 *It is not possible to construct a document in the traditional PPL that evaluates to the set of layouts $E = \{[\texttt{"a"}], [\texttt{"b"}]\}$. To see why, let* $\mathsf{rmspace} : \mathcal{L} \to \mathsf{Str}$ *be a function that joins all lines in a layout into a single line, with all whitespaces removed, and we lift* rmspace *to work on a set of layouts (i.e.,* $\mathsf{rmspace}(L) = \{\mathsf{rmspace}(l) : l \in L\}$*. Let* $\mathsf{eval}(\cdot)$ *be the evaluation function for the traditional PPL. We can prove by induction that* $\mathsf{rmspace}(\mathsf{eval}(d))$ *is a singleton set for any document d. In other words, all layouts in* $\mathsf{eval}(d)$ *are the same, modulo whitespaces. However,* $\mathsf{rmspace}(E) = \{\texttt{"a"}, \texttt{"b"}\}$*, which is not a singleton set. Hence, by congruence, no document can render to E.*

Note that there are other sets of layouts that are the same modulo whitespaces, but can't be evaluated to by the traditional PPL. This is due to how the `group` *construct is the only way to express choices, but a choice that it creates is very simple: collapsing everything into a single line. Therefore,* synchronized *differences of spacing across multiple lines can't be accounted for.*

Lemma 4.3 *For each construct* **F** *in* {`text`, `<>`, `nl`, `<|>`}*, Σ_e without* **F** *is not functionally complete.*

Proof sketch 4.3 *It is not possible to construct a document in each language in question that evaluates to the following set of layouts*

Σ_e **without** `text` $\{[\texttt{"a"}]\}$*, because all we can produce is whitespaces.*

Σ_e **without** `<>` {["a", "b", "c"]}, *because all we can produce is at most two lines.*

Σ_e **without** `nl` {["a", "b"]}, *because all we can produce is a single line.*

Σ_e **without** `<|>` {["a"], ["b"]}, *because all we can produce is a single layout.*

If we limit the notion of expressiveness to only functional completeness, then all functionally complete PPLs would be equally expressive. However, intuitively this is clearly not the case. The proof of Lemma 4.1 shows that it suffices for a PPL to only have `text`, `<$>`, and `<|>` for functional completeness, yet such a PPL would not be pleasant to use compared to Σ_e, because of the lack of features to, e.g., adjust indentation level. In a sense, functional completeness for PPLs is similar to Turing completeness for programming languages, which similarly does not fully capture expressiveness for programming languages. The next subsection presents a more fine-grained notion of expressiveness, based on the ability to define features.

4.4.3 Definability

The proof of Lemma 4.1 shows that while Σ_e doesn't have `<$>`, we can simply expand d_a `<$>` d_b to d_a `<> nl <>` d_b, which are in Σ_e, to perform the same functionality. In other words, the construct `<$>` is already *definable* by `<>` and `nl`. Thus, adding `<$>` to Σ_e doesn't increase its expressiveness. In contrast, `<>` is not definable by any combination of features in the arbitrary-choice PPL. To achieve the same functionality of `<>`, it would require a non-local restructuring of the document, making it difficult to construct the document in natural way. In this sense, the inability to define a construct in a PPL means that adding the construct to the PPL increases its expressiveness.

More concretely, consider the arbitrary-choice document in Figure 4.1b. The document is awkwardly constructed. The `return` keyword must be distributed to combine with a first line of the

returned expression, due to the undefinability of unaligned concatenation. This creates a disconnection between the document structure and the underlying AST structure, making it more tedious and error-prone to construct documents. In contrast, the following document is a rewrite of Figure 4.1b to utilize the full expressiveness of Σ_e in a natural way. The sub-document colored blue fully corresponds to the "returned expression," allowing users to recursively construct documents naturally.

```
text "function append(first,second,third){" <> nest 4 (
  let f = text "first +" in let s = text "second +" in let t = text "third" in
  nl <> text "return " <>
  ((text "(" <> (nest 4 (nl <> (f <> nl <> s <> nl <> t))) <> nl <> text ")") <|>
    let sp = text " " in (f <> sp <> s <> sp <> t))
) <> nl <> text "}"
```

The notion of definability (also known as expressibility) for programming languages was first developed by Felleisen [35], and we adapt it for PPLs through a series of definitions as follows:

Definition 4.2 *A PPL Σ consists of:*

- *a set of constructs $\Sigma = \{\mathbf{F}_1, \mathbf{F}_2, \ldots\}$. There could potentially be infinite constructs. Each construct may have different arity and sort, whose arguments may also have different sort.*
- *a non-empty set of documents $\mathcal{D}$, which are of sort* Doc*, generated from Σ.*
- *an evaluation function* eval $: \mathcal{D} \to 2^{\mathcal{L}}$.

Example 4.3 *Σ_e contains* `nest` $\square_{\mathbb{N}}$ $\square_{\text{Doc}}$*, which is a construct with arity 2 of sort* Doc*. The first argument to* `nest` *has sort $\mathbb{N}$ and the second argument has sort* Doc*. Σ_e also contains all natural numbers and strings with no newline, which are constructs with arity 0 of sort $\mathbb{N}$ and* Str *respectively. The evaluation function for Σ_e is* eval_e *from Section 4.3.3.*

Henceforth, unless indicated otherwise, $\mathcal{D}_X$ and eval_X are the set of documents and the evaluation function for the PPL Σ_X.

Definition 4.3 *A* syntactic abstraction $\mathbf{M}(\alpha_1, \ldots, \alpha_n)$ *of arity* n *for a PPL* Σ *is a document in* $\Sigma \cup \{\alpha_1, \ldots, \alpha_n\}$ *where* $\alpha_1, \ldots, \alpha_n$ *are metavariables (nullary constructors) of some sorts. An* instance $\mathbf{M}(e_1, \ldots, e_n)$ *is a document in* Σ *that substitutes* α_i *with* e_i *in* $\mathbf{M}(\alpha_1, \ldots, \alpha_n)$ *for all* $1 \leq i \leq n$*, where* e_i *and* α_i *must have a compatible sort.*

Example 4.4 $\mathbf{M}(\alpha_1, \alpha_2) = \alpha_1$ `<> nl <>` α_2 *is a syntactic abstraction for* Σ_e*, where* α_1 *and* α_2 *have sort* Doc*. On the other hand,* $\mathbf{M}'(\alpha_1) =$ `nest` α_1 `nl <>` α_1 ***is not*** *a syntactic abstraction because the first occurrence of* α_1 *requires it to have sort* $\mathbb{N}$*, but the second occurrence requires it to have sort* Doc*. An instance* $\mathbf{M}$(`text "a", text "b"`) *is the document* `text "a" <> nl <> text "b"`*, but* $\mathbf{M}$(`text "a",` 1) *is not an instance due to the incompatible sort.*

Definition 4.4 *Let* Σ_{base} *be a PPL and* $\Sigma_{\text{extended}} = \Sigma_{\text{base}} \cup \{\mathbf{F}\}$ *where* $\mathbf{F}$ *has sort* Doc*. A* syntactic expansion $\text{expand}^{\mathbf{M}}_{\mathbf{F}}(d)$ *from* Σ_{extended} *to* Σ_{base} *is a function from* $\mathcal{D}_{\text{extended}}$ *to* $\mathcal{D}_{\text{base}}$ *that replaces every occurrence of* $\mathbf{F}(e_1, \ldots, e_n)$ *with an instance* $\mathbf{M}(e_1, \ldots, e_n)$ *in* d*, where* $\mathbf{F}$ *and* $\mathbf{M}$ *must have compatible arity and sort arguments.*

Example 4.5 $\text{expand}^{\mathbf{M}}_{\texttt{<\$>}}(\cdot)$ *is a syntactic expansion from* $\Sigma_e \cup \{$ `<$>`$\}$ *to* Σ_e*, where* $\mathbf{M}$ *is from Example 4.4. Hence,* $\text{expand}^{\mathbf{M}}_{\texttt{<\$>}}$(`text "a" <$> text "b"`) = `text "a" <> nl <> text "b"`.

We are now ready to define definability.[5]

Definition 4.5 *Let* Σ_{base} *be a PPL and* $\Sigma_{\text{extended}} = \Sigma_{\text{base}} \cup \{\mathbf{F}\}$*. We say that* Σ_{base} *can* define $\mathbf{F}$ *(alternatively,* $\mathbf{F}$ *is* definable *by* Σ_{base}*) if there exists a syntactic abstraction* $\mathbf{M}$ *from* Σ_{extended} *to* Σ_{base} *such that for every document* $d \in \mathcal{D}_{\text{extended}}$*,* $\text{eval}_{\text{extended}}(d) = \text{eval}_{\text{base}}(\text{expand}^{\mathbf{M}}_{\mathbf{F}}(d))$.

We can now present one of our main results:

[5] One important distinction of this definition and Felleisen's counterpart is that PPLs are *total*. Hence, observing the termination behavior, as done in Felleisen's work, is not feasible in our formulation.

Theorem 4.1 *Σ_e can define every construct in the traditional and arbitrary-choice PPLs.*

Proof sketch 4.4 *Following syntactic abstractions can be used to define the constructs:*

- ***group*** *is definable by* $\mathbf{M}(\alpha_1) = \alpha_1$ **<|>** ***flatten*** α_1
- **<$>** *is definable by* $\mathbf{M}(\alpha_1, \alpha_2) = \alpha_1$ **<>** ***nl*** **<>** α_2.
- **<+>** *is definable by* $\mathbf{M}(\alpha_1, \alpha_2) = \alpha_1$ **<>** ***align*** α_2.

The rest of the constructs is already in Σ_e.

Despite the result, we might wonder if Σ_e is actually needed. Could it be that the arbitrary-choice PPL can already define every construct in the traditional PPL? As we foreshadowed, the answer to this question is negative. However, we must first develop tools that allow us to answer the question, again following the development in Felleisen's work.

Definition 4.6 *A context $C(\alpha)$ for Σ is a unary syntactic abstraction for Σ where α has sort* Doc.

Definition 4.7 *Given a PPL Σ and a relation $R \subseteq 2^{\mathcal{L}} \times 2^{\mathcal{L}}$, $E_R^{\Sigma}(d_1, d_2)$ is a relation that holds if and only if for all contexts C for Σ, $R(\mathsf{eval}(C(d_1)), \mathsf{eval}(C(d_2)))$ holds*[6].

Example 4.6 *Let* $\mathsf{width} : \mathcal{L} \to \mathbb{N}$ *be a function that computes the maximum length across all lines in the input layout, and we lift* width *to work on any set of layouts. Furthermore, let* $R = \{(L_1, L_2) : \mathsf{width}(L_1) = \mathsf{width}(L_2)\}$.

- $E_R^{\Sigma_e}$(***text*** `"a"`, ***text*** `"b"`) *holds by induction. Intuitively, this is because (1) if we only observe the width, the textual content doesn't matter, and (2) there is no construct in Σ_e that allows us to lay out differently in a way that would affect the width based on the textual content.*

[6]The relation E_R^{Σ} is a generalization of the operational equivalence relation in Felleisen's work.

- *On the other hand,* $\neg E_R^{\Sigma_e}(\texttt{text "a"}, \texttt{text "aa"})$. *For example, with* $C(\alpha) = \alpha$, *we have that* $\mathsf{width}(\mathsf{eval}_e(C(\texttt{text "a"}))) = \{1\}$, *but* $\mathsf{width}(\mathsf{eval}_e(C(\texttt{text "aa"}))) = \{2\}$.

The following theorem provides a tool to prove that a construct is not definable in a PPL.

Theorem 4.2 *Given a PPL* Σ *and a construct* $\mathbf{F}$, *if there exists two documents* d_1 *and* d_2 *in* Σ *and a relation* R *such that* $E_R^{\Sigma}(d_1, d_2)$, *but* $\neg E_R^{\Sigma\cup\{\mathbf{F}\}}(d_1, d_2)$, *then* $\mathbf{F}$ *is not definable in* Σ.

Proof sketch 4.5 *Let* $\mathsf{eval}_a(\cdot)$ *and* $\mathsf{eval}_b(\cdot)$ *denote the evaluation functions for* Σ *and* $\Sigma \cup \{\mathbf{F}\}$, *respectively. We prove the contraposition. Assuming that* $\mathbf{F}$ *is definable in* Σ, *we need to prove that for any* d_1, d_2, *and* R, $E_R^{\Sigma}(d_1, d_2)$ *implies* $E_R^{\Sigma\cup\{\mathbf{F}\}}(d_1, d_2)$. *Let* d_1, d_2, *and* R *be arbitrary. We suppose that for all context* C *in* Σ, $R(\mathsf{eval}_a(C(d_1)), \mathsf{eval}_a(C(d_2)))$ *holds, and need to prove that for all context* C *in* $\Sigma \cup \{\mathbf{F}\}$, $R(\mathsf{eval}_b(C(d_1)), \mathsf{eval}_b(C(d_2)))$ *holds.*

Let C *be a context in* $\Sigma \cup \{\mathbf{F}\}$. *Because* $\mathbf{F}$ *is definable in* Σ, *we can perform a syntactic expansion on* C *to obtain a context* C^* *in* Σ *such that* $\mathsf{eval}_a(C^*(d)) = \mathsf{eval}_b(C(d))$ *for all document* d *in* Σ. *Hence, it suffices to prove that* $R(\mathsf{eval}_a(C^*(d_1)), \mathsf{eval}_a(C^*(d_2)))$ *holds, but this is our hypothesis (instantiated with* C^**).*

With this tool, we are able to prove the following:

Theorem 4.3 *The following is true:*

- `<>` *is not definable in the arbitrary-choice PPL.*
- `nest` *is not definable in the arbitrary-choice PPL.*
- `group` *is not definable in the arbitrary-choice PPL.*
- `<+>` *is not definable in the traditional PPL.*

Proof sketch 4.6 *In each proof, we need to show that* **F** *is not definable in* Σ*, where* **F** *and* Σ *are the construct and the PPL in question. We do so by providing a counterexample (which is initially discovered by using Rosette [76, 93]), which consists of documents* d_1 *and* d_2*, and the relation* R*. By induction, it can be shown that* $E_R^{\Sigma}(d_1, d_2)$*. We will further provide a counterexample context to show that* $\neg E_R^{\Sigma \cup \{\mathbf{F}\}}(d_1, d_2)$*. By Theorem 4.2, this suffices to show that* **F** *is not definable in* Σ*.*

`<>` is not definable in the arbitrary-choice PPL *Given* width *from Example 4.4, the counterexample is* $d_1 =$ `text "a" <$> text "bb"`, $d_2 =$ `text "aa" <$> text "bb"`, *and* $R = \{(L_a, L_b) : \mathsf{width}(L_a) = \mathsf{width}(L_b)\}$*. In particular, with* $C(\alpha) =$ `text "c" <>` α*, we have that* $\mathsf{width}(\mathsf{eval}(d_1)) = \{2\}$*, but* $\mathsf{width}(\mathsf{eval}(d_2)) = \{3\}$*.*

`nest` is not definable in the arbitrary-choice PPL *Given* width *from Example 4.4, the counterexample is* $d_1 =$ `text "bb" <$> text "a"`, $d_2 =$ `text "cc" <$> text "bb" <$> text "a"`, *and* $R = \{(L_a, L_b) : \mathsf{width}(L_a) = \mathsf{width}(L_b)\}$*. In particular, with* $C(\alpha) =$ `nest` 1 α*, we have that* $\mathsf{width}(\mathsf{eval}(d_1)) = \{2\}$*, but* $\mathsf{width}(\mathsf{eval}(d_2)) = \{3\}$*.*

`group` is not definable in the arbitrary-choice PPL *Let* maxa $: \mathcal{L} \to \mathbb{N}$ *finds the maximum number of the character "a" in lines of the layout, and we lift* maxa *to work on a set of layouts. The counterexample is* $d_1 =$ `text "a" <$> text "a"`, $d_2 =$ `text "a" <$> text "a" <$> text "a"`, *and* $R = \{(L_a, L_b) : \mathsf{maxa}(L_a) = \mathsf{maxa}(L_b)\}$*. In particular, with* $C(\alpha) =$ `group` α*, we have that* $\mathsf{maxa}(\mathsf{eval}(d_1)) = \{1, 2\}$*, but* $\mathsf{maxa}(\mathsf{eval}(d_2)) = \{1, 3\}$*.*

`<+>` is not definable in the traditional PPL *Let* spaces $: \mathcal{L} \to \mathbb{N}$ *counts the number of spaces in a layout (not counting newlines), and we lift* spaces *to work on a set of layouts. The counterexample is* $d_1 =$ `text "a"`, $d_2 =$ `text "aa"`, *and* $R = \{(L_a, L_b) : \mathsf{spaces}(L_a) = \mathsf{spaces}(L_b)\}$*. In particular, with* $C(\alpha) = \alpha$ `<+> (text "b" <> nl <> text "c")`*, we have that that* $\mathsf{spaces}(\mathsf{eval}(d_1)) = \{1\}$*,*

but $\text{spaces}(\text{eval}(d_2)) = \{2\}$.

Next, we show a relationship between functional completeness and definability.

Lemma 4.4 *If* Σ *is not functionally complete, but* $\Sigma \cup \{\mathbf{C}\}$ *is, then* $\mathbf{C}$ *is not definable in* Σ.

Proof sketch 4.7 *Because* Σ *is not functionally complete, there is a set of layouts* L^* *that can't be evaluated to by any document in* Σ. *Since* $\Sigma \cup \{\mathbf{C}\}$ *is functionally complete, there is a document* d^* *(which necessarily contains* $\mathbf{C}$*) that evaluates to* L^*. *Let* d_1 *and* d_2 *be any document in* Σ, *and* $R = (2^{\mathcal{L}} \times 2^{\mathcal{L}}) \setminus \{(L^*, L^*)\}$. *Then* $E_R^{\Sigma}(d_1, d_2)$ *holds trivially. However, with* $C(\alpha) = d^*$, *we have that* $\neg E_R^{\Sigma \cup \{\mathbf{C}\}}(d_1, d_2)$. *This concludes the proof that* $\mathbf{C}$ *is not definable in* Σ.

Lastly, we present our final result for this section: Σ_e is *minimal* in a sense that each of its constructs is not definable by Σ_e without it. This is what the design of Σ_e strives to achieve, so that the size of primitive constructs is small.

Theorem 4.4 *For any construct* $\mathbf{F}$ *of* Σ_e, $\mathbf{F}$ *is not definable in* $\Sigma_e \setminus \{\mathbf{F}\}$.

Proof sketch 4.8 *The proofs for* `text`, `nl`, `<>`, *and* `<|>` *are applications of Lemma 4.1, Lemma 4.3, and Lemma 4.4. The proofs for* `nest`, `flatten`, *and* `align` *are just like how we proved Theorem 4.3 for* `nest`, `group`, *and* `<+>`.

4.5 Our Printer, Π_e

In this section, we describe our printer, Π_e, which targets the PPL Σ_e presented in Section 4.3. Π_e is parameterized by a *cost factory*, allowing users to customize the optimality objective within the resource budget that they find acceptable. We start with an overview of Π_e. Then, we describe the cost factory interface. Next, we define *measure*, which is an output from the core printer that

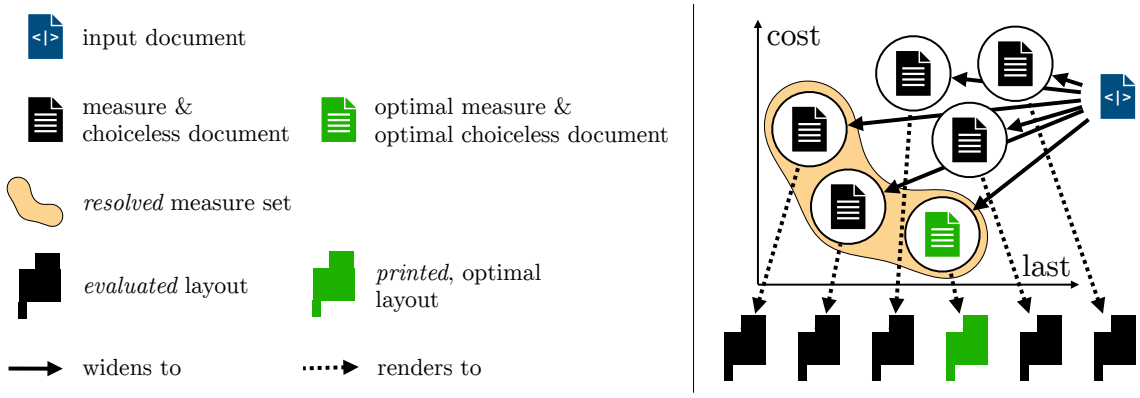


Figure 4.12: Π_e resolves the document to a set of measures, and chooses an optimal measure from it to render.

allows us to record a cost and at the same time avoid a full-blown, expensive rendering. After that, we describe the requirements of the input document structure, which will become important when we analyze the complexity of the printer. Then, we present the Π_e's printing algorithm, which utilizes the cost factory to achieve an optimal and efficient printing. Finally, we show our analysis of Π_e.

4.5.1 Overview

So far we have defined the *evaluation* of a document, which produces the set of possible layouts. But when we *print* a document, we wish to output only a single, most optimal layout.

A naïve approach is to evaluate the input document, via widening and rendering, to all possible layouts, determine costs of these layouts according to a given optimality objective, and then pick one with the least cost as the optimal layout. However, this approach is not practical for two reasons. First, widening could produce exponentially many choiceless documents. Second, rendering non-optimal choiceless documents is unnecessary and wasteful.

A better approach would utilize early pruning to reduce the search space, and avoid rendering until an optimal choiceless document is first identified. The need to prune early motivates us to

Cost type τ	
$\leq_{\mathcal{F}} : \tau \to \tau \to \mathbb{B}$	$\leq_{\mathcal{F}}$ must be a total ordering (transitive, antisymmetric, and total)
$+_{\mathcal{F}} : \tau \to \tau \to \tau$	$\forall C_1, C_2, C_3, C_4 \in \tau.\ [C_1 \leq_{\mathcal{F}} C_2 \to C_3 \leq_{\mathcal{F}} C_4 \to C_1 +_{\mathcal{F}} C_3 \leq_{\mathcal{F}} C_2 +_{\mathcal{F}} C_4]$
$\mathsf{text}_{\mathcal{F}} : \mathbb{N} \to \mathbb{N} \to \tau$	$\forall c, c', l \in \mathbb{N}.\ [c \leq c' \to \mathsf{text}_{\mathcal{F}}(c, l) \leq_{\mathcal{F}} \mathsf{text}_{\mathcal{F}}(c', l)]$
$\mathsf{newline}_{\mathcal{F}} : \mathbb{N} \to \tau$	$\forall i, i' \in \mathbb{N}.\ [i \leq i' \to \mathsf{newline}_{\mathcal{F}}(i) \leq_{\mathcal{F}} \mathsf{newline}_{\mathcal{F}}(i')]$
$W_{\mathcal{F}} : \mathbb{N}$	$W_{\mathcal{F}}$ has no contract

Figure 4.13: The cost factory interface. Users need to supply the cost type τ and implement the operations satisfying the contracts indicated in the interface.

devise the notion of *cost factory*, which allows us to incrementally compute costs to be used for pruning decision. At the same time, we design the cost factory to support a variety of optimality objectives. Since we wish to avoid full-blown rendering, we will instead operate on *measures* [8], which are limited views of choiceless document rendering under a printing context. This allows us to record the cost of a layout without expensive rendering.

The workflow of Π_e is shown in Figure 4.12. The printer first resolves choices, with early pruning, to produce a small set of measures that contain the optimal measure. The set in particular forms a Pareto frontier in the cost and last line length trade-off (Section 4.5.3 and Section 4.5.4). We could then pick the optimal measure from the set and render its choiceless document to produce an optimal layout.

4.5.2 The cost factory

The cost factory interface is presented in Figure 4.13. It allows users to define a cost type τ and implement operations on the cost type. $C_1 \leq_{\mathcal{F}} C_2$ tests if the cost C_1 is less than or equal to C_2. $C_1 +_{\mathcal{F}} C_2$ adds the cost C_1 and C_2 together. $\mathsf{text}_{\mathcal{F}}(c, l)$ computes a cost due to a placement of a string without a newline of length l at the column position c. $\mathsf{newline}_{\mathcal{F}}(i)$ is the cost due to a newline with i indentation spaces. The implemented operations must satisfy the indicated contracts in the interface, which are required for the core printer to function correctly. However, in practice, more constraints are recommended so that the printer is not brittle in presence of rewriting rules. We dis-

```
let args = nl <> text "arg1," <> nl <> text "arg2"
in
text "func(" <>
(group (nest 2 args <> nl)) <>
text ")"
```

```
1 func( arg1, arg2 )

1 func(
2   arg1,
3   arg2
4 )
```

Figure 4.14: An example document to illustrate how the cost factory computes a cost. The dotted lines show the width limit of 8 and 10.

cuss more about this topic in Section 4.8. Lastly, $W_{\mathcal{F}}$ is the computation width limit. When printing a document d, Π_e only provides the optimality guarantee among layouts in $\text{eval}_e(d)$ whose column position or indentation level during the printing does not exceeds $W_{\mathcal{F}}$. To illustrate how a cost factory is used in Π_e, we show a printing of the document in Figure 4.14 with a concrete cost factory.

Example 4.7 *Consider an optimality objective that minimizes the sum of* overflows*, which are the numbers of characters that exceed a given* page width limit w *in each line, and* then *minimizes the* height*, which is the total number of newline characters (or equivalently, the number of lines minus one). This objective is thus able to avoid the excessive overflow problem in Bernardy's printer described in Section 4.2.*

More formally, given a layout, a cost is a pair of the overflow sum and the height, where the lexicographic order determines which cost is less. With $w = 8$, the first layout in Figure 4.14 has the cost $(10, 0)$, whereas the second layout has the cost $(0, 3)$. Thus, the second layout is the optimal layout that Π_e should pick.

We implement the optimality objective with the following cost factory $\mathcal{F}$.

$$\tau = \mathbb{N} \times \mathbb{N} \quad \leq_{\mathcal{F}} = \leq_{\text{lex}} \quad (o_a, h_a) +_{\mathcal{F}} (o_b, h_b) = (o_a + o_b, h_a + h_b)$$

$$\text{text}_{\mathcal{F}}(c, l) = (\max(c + l - \max(w, c), 0), 0) \quad \text{newline}_{\mathcal{F}}(i) = (\max(i - w, 0), 1)$$

Each text and newline placement would query the cost factory to compute its cost. For example, `"func("` *of length 5 is placed at the column position 0 in both layouts, so the cost is* $\text{text}_{\mathcal{F}}(0, 5) = (0, 0)$.

In the first layout, `"arg1,"` *of length 5 is placed at the column position 6, so the cost is* $\mathsf{text}_{\mathcal{F}}(6,5) = (3,0)$. *On the other hand, in the second layout,* `"arg1,"` *is placed at the column position 2, so the cost is* $\mathsf{text}_{\mathcal{F}}(2,5) = (0,0)$. *Spaces due to newlines when the flattening mode is on are considered text.* $\mathsf{newline}_{\mathcal{F}}$ *is never used for the first layout, but the second layout would query* $\mathsf{newline}_{\mathcal{F}}$ *twice with indentation level 2 and once with indentation level 0. When we sum the costs up using* $+_{\mathcal{F}}$*, we obtain that the first layout has the cost* $(10,0)$*, whereas the second layout has the cost* $(0,3)$ *as expected.* $\leq_{\mathcal{F}}$ *could then be used to conclude that the second layout is optimal.*

We have been ignoring the computation width limit $W_{\mathcal{F}}$ *until now. When a column position or indentation level exceeds the computation width limit during a printing, the result is* tainted. *For example, with* $W_{\mathcal{F}} = 10$*, the first layout would be tainted, while the second layout would not. Tainted layouts can usually be discarded right away, except when every possible layout is tainted. In such case,* Π_e *keeps one tainted layout so that it can still output a layout, but provide no guarantee that the layout will be optimal. The tainting system allows us to bound the computation so that the algorithm is efficient.*

The cost factory interface is versatile. The above example shows that Π_e does not need to take a page width limit as an input, because the concept of page width limit can already be defined by users via $\mathsf{text}_{\mathcal{F}}$. It is also possible to, for example, implement the concept of *soft* width limit, compute a linear combination of height and overflow in the style of Yelland [104], compute a sum of squared overflow, and compute a maximum of overflow. Furthermore, the cost factory plays a central role that enables Π_e to be efficient. In the subsequent subsections, every definition is implicitly parameterized by a cost factory $\mathcal{F}$.

4.5.3 Measure

As presented earlier, the resolving phase computes *measure*s. Each measure is a limited view of a choiceless document rendering that incorporates cost. Presented in Figure 4.15, a measure consists

$$\text{Measure } m \in \mathcal{M} = \langle l, C, \overline{d}, x, y \rangle_{\mathcal{M}}$$

$$\begin{array}{llll} \text{last} : \mathcal{M} \to \mathbb{N} & \text{last}(\langle l, C, \overline{d}, x, y \rangle_{\mathcal{M}}) = l & \text{cost} : \mathcal{M} \to \tau & \text{cost}(\langle l, C, \overline{d}, x, y \rangle_{\mathcal{M}}) = c \\ \text{doc} : \mathcal{M} \to \overline{\mathcal{D}_e} & \text{doc}(\langle l, C, \overline{d}, x, y \rangle_{\mathcal{M}}) = \overline{d} & & \\ \text{maxx} : \mathcal{M} \to \mathbb{N} & \text{maxx}(\langle l, C, \overline{d}, x, y \rangle_{\mathcal{M}}) = x & \text{maxy} : \mathcal{M} \to \mathbb{N} & \text{maxy}(\langle l, C, \overline{d}, x, y \rangle_{\mathcal{M}}) = y \end{array}$$

$$\begin{array}{ll} \circ : \mathcal{M} \to \mathcal{M} \to \mathcal{M} & \langle l_a, C_a, \overline{d}_a, x_a, y_a \rangle_{\mathcal{M}} \circ \langle l_b, C_b, \overline{d}_b, x_b, y_b \rangle_{\mathcal{M}} = \\ & \quad \langle l_b, C_a +_{\mathcal{F}} C_b, \overline{d}_a \texttt{ <> } \overline{d}_b, \max(x_a, x_b), \max(y_a, y_b) \rangle_{\mathcal{M}} \\ \text{adjustNest} : \mathbb{N} \to \mathcal{M} \to \mathcal{M} & \text{adjustNest}(n, \langle l, C, \overline{d}, x, y \rangle_{\mathcal{M}}) = \langle l, C, \texttt{nest } n\ \overline{d}, x, y \rangle_{\mathcal{M}} \\ \text{adjustAlign} : \mathbb{N} \to \mathcal{M} \to \mathcal{M} & \text{adjustAlign}(i, \langle l, C, \overline{d}, x, y \rangle_{\mathcal{M}}) = \langle l, C, \texttt{align } \overline{d}, x, \max(y, i) \rangle_{\mathcal{M}} \\ \leq : \mathcal{M} \to \mathcal{M} \to \mathbb{B} & \langle l_a, C_a, \overline{d}_a, x, y \rangle_{\mathcal{M}} \leq \langle l_b, C_b, \overline{d}_b, x, y \rangle_{\mathcal{M}} = l_a \leq l_b \wedge C_a \leq_{\mathcal{F}} C_b \end{array}$$

Figure 4.15: Measure and operations on measures

$$\textsc{TextM}\ \frac{}{\langle \texttt{text } s, c, i \rangle \Downarrow_{\mathbb{M}} \langle c + |s|, \text{text}_{\mathcal{F}}(c, |s|), \texttt{text } s, c + |s|, i \rangle_{\mathcal{M}}}$$

$$\textsc{LineM}\ \frac{}{\langle \texttt{nl}, c, i \rangle \Downarrow_{\mathbb{M}} \langle i, \text{newline}_{\mathcal{F}}(i), \texttt{nl}, \max(c, i), i \rangle_{\mathcal{M}}} \qquad \textsc{ConcatM}\ \frac{\langle \overline{d}_a, c, i \rangle \Downarrow_{\mathbb{M}} m_a \quad \langle \overline{d}_b, \text{last}(m_a), i \rangle \Downarrow_{\mathbb{M}} m_b}{\langle \overline{d}_a \texttt{ <> } \overline{d}_b, c, i \rangle \Downarrow_{\mathbb{M}} m_a \circ m_b}$$

$$\textsc{NestM}\ \frac{\langle \overline{d}, c, i + n \rangle \Downarrow_{\mathbb{M}} m}{\langle \texttt{nest } n\ \overline{d}, c, i \rangle \Downarrow_{\mathbb{M}} \text{adjustNest}(n, m)} \qquad \textsc{AlignM}\ \frac{\langle \overline{d}, c, c \rangle \Downarrow_{\mathbb{M}} m}{\langle \texttt{align } \overline{d}, c, i \rangle \Downarrow_{\mathbb{M}} \text{adjustAlign}(i, m)}$$

Figure 4.16: Measure computation from a choiceless document in a printing context

of five components: length of last line (l), cost (C), choiceless document (d), max column position (x), and max indentation (y). We gray out the last two components because they are ghosted [68], only needed for the correctness theorem, and not required in the actual implementation.

Example 4.8 *Given* $\overline{d}$ `= text "= {|" <> nest 1 (nl <> text "1234") <> nl <> text "|}"` *being placed at the column position* 0*, with indentation level* 0*. The choiceless document would render to a layout* `["= {|", " 1234", "|}"]`*. With the cost factory in Example 4.7, the cost of the layout is* 0*. Thus, the measure is* $\langle 2, 0, \overline{d}, 5, 1 \rangle_{\mathcal{M}}$.

Figure 4.16 shows rules that define measure computation. The judgment $\langle d, c, i \rangle \Downarrow_{\mathbb{M}} m$ states that when we compute the measure of $\overline{d} \in \overline{\mathcal{D}_e}$ placed at the column position $c \in \mathbb{N}$ with indentation level $i \in \mathbb{N}$, the resulting measure is $m \in \mathcal{M}$. To simplify the core printer, we (temporarily) remove

flatten from Σ_e. This allows us to eliminate the flattening mode parameter, which implicitly defaults to $\bot$. Toward the end of this section, we will show how to add support for **flatten** back.

The rules are largely standard. They reflect the actual rendering defined by $\Downarrow_{\mathcal{R}}$, and utilize the cost factory in a straightforward way. The rules use a helper operator function $\circ$ to concatenate two measures, and helper functions adjustNest and adjustAlign to construct a correct measure for **nest** and **align**. These functions are defined in Figure 4.15. Notably, the LineM rule creates a measure whose maxc is $\max(c, i)$ because before placing the newline, the column position is c, and after placing the newline, the column position is i. The AlignM rule creates a measure whose maxi is $\max(y, i)$ where y is obtained via the recursive computation. This is because the recursive computation discards the current indentation level, so we need to specifically record the information.

$\Downarrow_{\mathbb{M}}$ is deterministic and total. It is also correct with respect to $\Downarrow_{\mathcal{R}}$.

Theorem 4.5 *For any $\overline{d} \in \overline{\mathcal{D}_e}$ and $c, i \in \mathbb{N}$, there exists a cost C and max indentation y such that*

- *if $\langle \overline{d}, c, i, \bot \rangle \Downarrow_{\mathcal{R}} [s]$, then $\langle \overline{d}, c, i \rangle \Downarrow_{\mathbb{M}} \langle c + |s|, C, \overline{d}, c + |s|, y \rangle_{\mathcal{M}}$.*
- *if $\langle \overline{d}, c, i, \bot \rangle \Downarrow_{\mathcal{R}} [s, s_1, \ldots, s_n, t]$, then $\langle \overline{d}, c, i \rangle \Downarrow_{\mathbb{M}} \langle |t|, C, \overline{d}, \max(c + |s|, |s_1|, \ldots, |s_n|, |t|), y \rangle_{\mathcal{M}}$*

So far, we only consider the measure computation for a choiceless document. When we take the choice operator into account, there could be multiple measures under the same printing context. The main operation that we can perform on these measures is finding domination $\preceq$, also presented in Figure 4.15. $m_a \preceq m_b$ when both the cost and the last length of m_a are no worse than those of m_b. The fact that $m_a \preceq m_b$ is useful because it allows us to prune m_b away immediately.

$$
\begin{array}{rcll}
\text{Measure set } S \in \mathcal{S} & ::= & \texttt{Tainted}(\hat{m}) & \text{where } \hat{m} \text{ is a promise that can be forced to a measure} \\
& | & \texttt{Set}([m_1, \ldots^+, m_n]) & \text{where } \mathrm{last}(m_1) > \ldots > \mathrm{last}(m_n) \text{ and } \forall i \neq j, \neg(m_i \preceq m_j \vee m_j \preceq m_i)
\end{array}
$$

$$
\begin{array}{rl}
\mathrm{taint} : \mathcal{S} \to \mathcal{S} & \mathrm{taint}(\texttt{Tainted}(m)) = \texttt{Tainted}(m) \\
& \mathrm{taint}(\texttt{Set}([m_0, m_1, \ldots, m_n])) = \texttt{Tainted}(m_0) \\
\mathrm{lift} : \mathcal{S} \to (\mathcal{M} \to \mathcal{M}) \to \mathcal{S} & \mathrm{lift}(\texttt{Tainted}(m), f) = \texttt{Tainted}(f(m)) \\
& \mathrm{lift}(\texttt{Set}([m_1, \ldots^+, m_n]), f) = \texttt{Set}([f(m_1), \ldots^+, f(m_n)]) \\
\mathrm{dedup} : \overrightarrow{\mathcal{M}} \to \overrightarrow{\mathcal{M}} & \mathrm{dedup}([m, m', m_1, \ldots, m_n]) = \mathrm{dedup}([m', m_1, \ldots, m_n]) \quad \text{if } m' \preceq m \\
& \mathrm{dedup}([m, m', m_1, \ldots, m_n]) = [m]@\mathrm{dedup}([m', m_1, \ldots, m_n]) \quad \text{if } m' \not\preceq m \\
& \mathrm{dedup}([m]) = [m] \\
\uplus : \mathcal{S} \to \mathcal{S} \to \mathcal{S} & S \uplus \texttt{Tainted}(m) = S \\
& \texttt{Tainted}(m) \uplus \texttt{Set}([m_1, \ldots^+, m_n]) = \texttt{Set}([m_1, \ldots^+, m_n]) \\
& \texttt{Set}([m_1, \ldots^+, m_n]) \uplus \texttt{Set}([m'_1, \ldots^+, m'_{n'}]) = \texttt{Set}([m_1, \ldots^+, m_n] \uplus [m'_1, \ldots^+, m'_{n'}]) \\
\uplus : \overrightarrow{\mathcal{M}} \to \overrightarrow{\mathcal{M}} \to \overrightarrow{\mathcal{M}} & [] \uplus [m_1, \ldots^+, m_n] = [m_1, \ldots^+, m_n] \\
& [m_1, \ldots^+, m_n] \uplus [] = [m_1, \ldots^+, m_n]
\end{array}
$$

$$
[m_0, m_1, \ldots, m_n] \uplus [m'_0, m'_1, \ldots, m'_{n'}] = \begin{cases}
[m_0, m_1, \ldots, m_n] \uplus [m'_1, \ldots, m'_{n'}] & \text{if } m_0 \preceq m'_0 \\
[m_1, \ldots, m_n] \uplus [m'_0, m'_1, \ldots, m'_{n'}] & \text{if } m'_0 \preceq m_0 \\
[m_0]@([m_1, \ldots, m_n] \uplus [m'_0, m'_1, \ldots, m'_{n'}]) & \text{if } \mathrm{last}(m_0) > \mathrm{last}(m'_0) \\
[m'_0]@([m_0, m_1, \ldots, m_n] \uplus [m'_1, \ldots, m'_{n'}]) & \text{otherwise}
\end{cases}
$$

Figure 4.17: Measure set and the merge operation on measure sets. @ denotes a list concatenation. We treat a promise $\hat{m}$ and a measure m interchangeably, as they can be straightforwardly casted to each other.

4.5.4 Measure set

Resolving a document (in a printing context) produces a small set of measures. To accommodate taintedness mentioned in Section 4.5.2, Figure 4.17 defines a measure set to be either a non-empty `Set` of untainted measures where no measure dominates the other, or a `Tainted` singleton set of a promise $\hat{m}$ that can be forced to a measure. The `Set`, by definition, forms a Pareto frontier. To aid computation, we represent the `Set` with a list ordered by the cost in the strict ascending order (and therefore the last length in the strict descending order). We are able to do so because in a Pareto frontier, all last and cost values must be distinct.

The main operation that we can perform on measure sets is merging two measure sets ($\uplus$), shown in Figure 4.17, where we prefer a `Set` over a `Tainted`. The merge operation maintains the Pareto frontier invariant, by doing the merge in the style of the merge operation in merge sort, although the Pareto frontier merging can also prune measures away during the operation. One important "quirk" of this merge operation is that it is *left-biased* in presence of taintedness. If two tainted measure sets are merged, the result is always the left one. This means the order of arguments to the merge operation is important, as we will see in the next subsections.

Other operations on measure sets which are used in next subsections are taint, lift, and dedup. taint taints a measure set. When tainting a `Set`, we choose to pick the first measure from the `Set` because it has the least cost, which is a greedy heuristic. lift adjusts measures in a measure set. Lastly, dedup prunes measures that are sorted by last in the strict decreasing order and by cost in the non-strict increasing order, so that the result conforms the Pareto frontier invariant.

4.5.5 The document structure

Section 4.2.2 has shown that we need to handle document sharing by treating the input document as a DAG. However, documents cannot be arbitrarily shared, as the following example shows:

Example 4.9 *The following document* mk(n) *has the DAG size of* $O(n)$*. However, resolving it necessitates* $O(2^n)$ *units of computation, as the printing contexts are all different. This is bad news because it means resolving could take exponential time in the input size.*

```
let rec mk (n : int): doc =
  if n = 0 then text "x"
  else let shared = mk (n - 1) in shared <> shared
```

However, we argue that the above document is not *properly shared*, because the sub-documents are not shared *across choices*, which is how sharing is employed in practice. The corresponding properly shared document should have $O(2^n)$ DAG size, so $O(2^n)$ units of computation are still linear in the input size. To make this precise, we provide the following definitions:

Definition 4.8 *Given a document* $d \in \mathcal{D}_e$*,* $G(d)$ *is a DAG rooted at* d *whose edge in the graph connects a document to its direct subdocuments.*

Definition 4.9 *A document* $d \in \mathcal{D}_e$ *is* properly shared *if for any two vertices* d_a *and* d_b *in* $G(d)$*, if* p_1 *and* p_2 *are two distinct paths from* d_a *to* d_b*, then there exists a common document* d' *such that (1)* d' *is a* `<|>`*; (2)* d' *occurs in both* p_1 *and* p_2*; and (3)* d' *is not* d_b*.*

Figure 4.5c shows a properly shared document (assuming that D is properly shared). It illustrates two paths where d_a is the root node, d_b is D, and d' is d_a. In practice, non-properly shared documents can still be processed by Π_e, and in fact can even make resolving faster when a shared document is resolved under the same printing context. However, this shared document would be effectively duplicated when it is resolved in different contexts. For simplicity, we only consider properly shared documents as the input to Π_e in this work.

4.5.6 The resolver

We now formally define the core of Π_e, which is the resolver. It is described in Figure 4.18, which is a fusion of widening in Figure 4.11 and measure computation in Figure 4.16, with early

$$\textsc{TextRSSet}\frac{c + |s| \leq W_{\mathcal{F}} \quad i \leq W_{\mathcal{F}} \quad \langle \textbf{text}\ s, c, i\rangle \Downarrow_{\mathbb{M}} m}{\langle \textbf{text}\ s, c, i\rangle \Downarrow_{\mathbb{RS}} \mathtt{Set}([m])} \qquad \textsc{LineRSSet}\frac{c \leq W_{\mathcal{F}} \quad i \leq W_{\mathcal{F}} \quad \langle \textbf{nl}, c, i\rangle \Downarrow_{\mathbb{M}} m}{\langle \textbf{nl}, c, i\rangle \Downarrow_{\mathbb{RS}} \mathtt{Set}([m])}$$

$$\textsc{TextRSTnt}\frac{c + |s| > W_{\mathcal{F}} \vee i > W_{\mathcal{F}} \quad \langle \textbf{text}\ s, c, i\rangle \Downarrow_{\mathbb{M}} m}{\langle \textbf{text}\ s, c, i\rangle \Downarrow_{\mathbb{RS}} \mathtt{Tainted}(m)} \qquad \textsc{LineRSTnt}\frac{c > W_{\mathcal{F}} \vee i > W_{\mathcal{F}} \quad \langle \textbf{nl}, c, i\rangle \Downarrow_{\mathbb{M}} m}{\langle \textbf{nl}, c, i\rangle \Downarrow_{\mathbb{RS}} \mathtt{Tainted}(m)}$$

$$\textsc{NestRS}\frac{\langle d, c, i + n\rangle \Downarrow_{\mathbb{RS}} S}{\langle \textbf{nest}\ n\ d, c, i\rangle \Downarrow_{\mathbb{RS}} \mathsf{lift}(S, \mathsf{adjustNest}(n))} \qquad \textsc{AlignRS}\frac{i \leq W_{\mathcal{F}} \quad \langle d, c, c\rangle \Downarrow_{\mathbb{RS}} S}{\langle \textbf{align}\ d, c, i\rangle \Downarrow_{\mathbb{RS}} \mathsf{lift}(S, \mathsf{adjustAlign}(i))}$$

$$\textsc{UnionRS}\frac{\langle d_a, c, i\rangle \Downarrow_{\mathbb{RS}} S_a \quad \langle d_b, c, i\rangle \Downarrow_{\mathbb{RS}} S_b}{\langle d_a\ \textbf{<|>}\ d_b, c, i\rangle \Downarrow_{\mathbb{RS}} S_a \uplus S_b} \qquad \textsc{AlignRSTnt}\frac{i > W_{\mathcal{F}} \quad \langle d, c, c\rangle \Downarrow_{\mathbb{RS}} S}{\langle \textbf{align}\ d, c, i\rangle \Downarrow_{\mathbb{RS}} \mathsf{lift}(\mathsf{taint}(S), \mathsf{adjustAlign}(i))}$$

$$\textsc{ConcatRS}\frac{\langle d_a, c, i\rangle \Downarrow_{\mathbb{RS}} \mathtt{Set}([m_1, \ldots, m_n]) \quad \langle m_1, d_b, i\rangle \Downarrow_{\mathbb{RSC}} S_1 \quad \ldots \quad \langle m_n, d_b, i\rangle \Downarrow_{\mathbb{RSC}} S_n}{\langle d_a\ \textbf{<>}\ d_b, c, i\rangle \Downarrow_{\mathbb{RS}} S_1 \uplus \ldots \uplus S_n}$$

$$\textsc{ConcatRSTnt}\frac{\langle d_a, c, i\rangle \Downarrow_{\mathbb{RS}} \mathtt{Tainted}(m_a) \quad \langle d_b, \mathsf{last}(m_a), i\rangle \Downarrow_{\mathbb{RS}} S \quad \mathsf{taint}(S) = \mathtt{Tainted}(m_b)}{\langle d_a\ \textbf{<>}\ d_b, c, i\rangle \Downarrow_{\mathbb{RS}} \mathtt{Tainted}(m_a \circ m_b)}$$

$$\textsc{RSCSet}\frac{\langle d_b, \mathsf{last}(m_a), i\rangle \Downarrow_{\mathbb{RS}} \mathtt{Set}([m_1, \ldots^+, m_n])}{\langle m_a, d_b, i\rangle \Downarrow_{\mathbb{RSC}} \mathtt{Set}(\mathsf{dedup}([m_a \circ m_1, \ldots^+, m_a \circ m_n]))} \qquad \textsc{RSCTnt}\frac{\langle d_b, \mathsf{last}(m_a), i\rangle \Downarrow_{\mathbb{RS}} \mathtt{Tainted}(m_b)}{\langle m_a, d_b, i\rangle \Downarrow_{\mathbb{RSC}} \mathtt{Tainted}(m_a \circ m_b)}$$

Figure 4.18: The resolver

pruning inherent in the merge operation and extra bookkeeping for taintedness. The judgment $\langle d, c, i\rangle \Downarrow_{\mathbb{RS}} S$ states that a properly shared document $d \in \mathcal{D}_e$ at a column position $c \in \mathbb{N}$ with an indentation level $i \in \mathbb{N}$ resolves to a measure set S.

Resolving text If placing the text would exceed $W_{\mathcal{F}}$ or the indentation level is beyond $W_{\mathcal{F}}$, the TEXTRSTNT rule returns a `Tainted`. Otherwise, the TEXTRS rule returns a singleton `Set`.

Resolving newline Resolving a **`nl`** is similar to resolving a **`text`**, but we only need to consider the current column position and indentation level, as resolving the newline does not change the column position. The LINERSTNT and LINERS rules cover these two cases.

Resolving nest Resolving a **`nest`** is handled by the NESTRS rule, which recursively resolves its sub-document, with the indentation level changed. The recursive resolving would determine whether the measure set would be a `Set` or `Tainted`. In all cases, the result is adjusted to construct correct choiceless documents.

Resolving alignment Resolving an **`align`** is similar to resolving **`nest`**. However, because the recursive resolving would discard the current indentation level, which could exceed $W_{\mathcal{F}}$, we need to taint the measure set when the indentation level is beyond $W_{\mathcal{F}}$. The ALIGNRSTNT rule handles such case, and the ALIGNRS rule handles other possibilities.

Resolving choice The UNIONRS rule recursively resolves its two sub-documents, and then merges the measure sets. As mentioned in Section 4.5.4, the merge operation is left-biased. Therefore, the left sub-document will be preferred over the right sub-document if exceeding $W_{\mathcal{F}}$ is unavoidable. It is possible to employ a heuristic to remove this bias, as discussed in Section 4.8.

Resolving unaligned concatenation Resolving a <> is done through ConcatRSTnt and ConcatRS rules, which handle the two possibilities of measure set types obtained from the left sub-document's recursive resolving. Notably, the ConcatRS rule employs $\Downarrow_{\mathbb{RSC}}$ to help us concatenates a left measure from the left measure set with a right measure set. The series of merges should be done with the right fold ordering to avoid unnecessary list traversal.

$\Downarrow_{\mathbb{RS}}$ is deterministic and total. This allows us to define the top-level printer as $\Pi_e(d) = l$ where $\langle \mathrm{doc}(m_0), 0, 0, \bot \rangle \Downarrow_{\mathcal{R}} l$ and $\langle d, 0, 0 \rangle \Downarrow_{\mathbb{RS}} [m_0, m_1, \ldots, m_n]$, which consumes a properly shared document d, resolves it to a set of measures, picks the measure with the least cost, and simply renders the associated choiceless document to produce a layout (although our implementation further fuses resolving and rendering together, as described in Section 4.8).

While the rules above are enough for correctness, implementing these rules requires further consideration. As we will see in Lemma 4.6, any resolving beyond $W_{\mathcal{F}}$ would eventually result in a tainted measure set. Hence, Π_e should *immediately* delay the computation for any resolving beyond $W_{\mathcal{F}}$. Π_e should also *memoize* the computation, so that on identical document and printing context within $W_{\mathcal{F}}$, the result of the previous computation is reused.

We claim that $\Pi_e(d)$ consumes a properly shared document d in Σ_e and produces an optimal layout among $\mathrm{eval}_e(d)$ within $W_{\mathcal{F}}$. We will prove the claim in the next subsection.

4.5.7 Correctness of Π_e

$\Downarrow_{\mathbb{P}}$ is correct with respect to $\Downarrow_{\mathbb{M}}$. Two theorems govern the correctness. The first theorem states that the core printer returns a measure set that contains a measure that is no worse than any measure within the computation width limit from all possible measures.

Theorem 4.6 (Optimality) *For any $d \in \mathcal{D}_e$, $c \in \mathbb{N}$, $i \in \mathbb{N}$, if the following conditions hold*

- $\langle d, c, i\rangle \Downarrow_{\mathbb{RS}} S$
- $\overline{d} \in \overline{D}$
- $\mathsf{maxx}(m) \leq W_{\mathcal{F}}$
- $d \Downarrow_{\mathcal{W}} \overline{D}$
- $\langle \overline{d}, c, i\rangle \Downarrow_{\mathbb{M}} m$
- $\mathsf{maxy}(m) \leq W_{\mathcal{F}}$

then $S = \texttt{Set}([m_1, \ldots^+, m_n])$. *Furthermore, there exists* i *such that* $m_i \preceq m$.

The second theorem states that measures in the resulting measure set are valid.

Theorem 4.7 (Validity) *For any* $d \in \mathcal{D}_e$, $c \in \mathbb{N}$, $i \in \mathbb{N}$ *with* $d \Downarrow_{\mathcal{W}} \overline{D}$, *if it is the case that* $\langle d, c, i\rangle \Downarrow_{\mathbb{RS}} \texttt{Set}([m_1, \ldots^+, m_n])$, *then for each* i, *there exists* $\overline{d}$ *such that* $\overline{d} \in \overline{D}$ *and* $\langle \overline{d}, c, i\rangle \Downarrow_{\mathbb{M}} m_i$. *Likewise, if* $\langle d, c, i\rangle \Downarrow_{\mathbb{RS}} \texttt{Tainted}(m_0)$, *then there exists* $\overline{d}$ *such that* $\overline{d} \in \overline{D}$ *and* $\langle \overline{d}, c, i\rangle \Downarrow_{\mathbb{M}} m_0$.

The correctness of Π_e follows immediately.

While the above theorems guarantee the correctness of the result that the printer produces, it does not concern with efficiency. The following lemmas provide some properties of the printer that allow us to reason about the efficiency.

Lemma 4.5 *For any* $d \in \mathcal{D}_e$, $c \leq W_{\mathcal{F}}$, $i \leq W_{\mathcal{F}}$, *if* $\langle d, c, i\rangle \Downarrow_{\mathbb{RS}} \texttt{Set}([m_1, \ldots, m_n])$, *then* $n \leq W_{\mathcal{F}} + 1$.

Lemma 4.6 *For any* $d \in \mathcal{D}_e$, *if* $c > W_{\mathcal{F}}$ *or* $i > W_{\mathcal{F}}$ *and* $\langle d, c, i\rangle \Downarrow_{\mathbb{RS}} S$, *then* S *is a* `Tainted`.

We now informally prove the efficiency of Π_e that we claimed in Section 4.1.

Theorem 4.8 *The time complexity of* Π_e *is* $O(nW_{\mathcal{F}}^4)$ *where* n *is the DAG size of the document.*

Proof sketch 4.9 *The most expensive operation in the printer is concatenation (via* ConcatRSSet*). The operation resolves the left sub-document, resulting in a measure set whose size is at most* $W_{\mathcal{F}}$ *according to Lemma 4.5. It then resolves the right sub-document in at most* $W_{\mathcal{F}}$ *different contexts. Thus, there are at most* $W_{\mathcal{F}}^2$ *different measures from the right sub-document that the printer needs to concatenate and prune.*

Consider $\langle d, c, i\rangle \Downarrow_{\mathbb{RS}} S$. d can range over n different values. c and i can range over $W_{\mathcal{F}}$ different values that are under $W_{\mathcal{F}}$. Hence, there are $O(nW_{\mathcal{F}}^2)$ different contexts under the computation width limit. Multiplying this with the maximum units of computation in the previous paragraph, we obtain that the time complexity due to resolving within $W_{\mathcal{F}}$ is $O(nW_{\mathcal{F}}^4)$, assuming that the resolver reuses memoized measure set under the same context.

When d is printed beyond $W_{\mathcal{F}}$, however, it can be fully resolved for at most once, because:

1. *While we would resolve both sub-documents of choice nodes, they would be all tainted, due to Lemma 4.6. Because all tainted measure sets are promises, all computations are delayed. The merge operation then chooses only one tainted measure set as the result, discarding the other one.*

2. *The document is properly shared, so under a path a document is encountered at most once.*

As a result, the time complexity due to printing over $W_{\mathcal{F}}$ is simply $O(n)$. Combining both parts, we obtain that the time complexity of Π_e is $O(nW_{\mathcal{F}}^4)$.

Theorem 4.9 *If a document d is in the arbitrary-choice PPL, Π_e can print d in $O(nW_{\mathcal{F}}^3)$.*

Proof sketch 4.10 *In the arbitrary-choice PPL, $c = i$ is (mostly) maintained throughout the printing. Hence, there is one less dimension to consider, leading to the time complexity of $O(nW_{\mathcal{F}}^3)$.*

4.5.8 Handling flattening

To support `flatten`, we make it a function that walks its sub-document and replaces all `nl` with `text " "`. The walk is memoized and preserves the original identity of the document whenever possible (i.e. if nothing is flatten in sub-documents, then the document itself is returned unchanged without creating a new document). Thus, each document can be flattened at most once. This

flattening creates at most $O(n)$ new documents without destroying the shared structure in the original document. We therefore achieve the functionality of `flatten` without affecting the time complexity of the printer.

4.6 Implementation

We implement Π_e in OCaml and Racket. The printer, which we call SNOWWHITE, is further refined to be more efficient and practical. Due to the space limit, we describe these refinements in Section 4.8. The OCaml SNOWWHITE, as a reference implementation, is used for comparing against other printers in Section 4.7. The Racket SNOWWHITE has even more features, and it has been used to implement the code formatter for the Racket programming language.

SNOWWHITE provides a pre-defined cost factory that minimizes the sum of squared overflows over the page width limit w (without considering indentation spaces) and then height. The text placement formula is derived from the identity $(a+b)^2 - a^2 = b(2a+b)$ where in each text placement, a is the starting position count past the page width limit and b is the overflow length. With this cost factory and $w = 8$, the first layout in Figure 4.14 has the cost $(10^2, 0)$ whereas the second layout has the cost $(0, 3)$.

$$\text{text}_{\mathcal{F}}(c,l) = \begin{cases} (b(2a+b),0) & \text{if } c+l > w \\ (0,0) & \text{otherwise} \end{cases} \quad \text{where} \quad \begin{array}{l} a = \max(w,c) - w \\ b = c + l - \max(w,c) \end{array}$$

4.7 Evaluation

This evaluates the performance and optimality of SNOWWHITE. The evaluation consists of two parts. First, we compare SNOWWHITE against Wadler/Leijen [55] and Bernardy [7]'s printers, which are popular practical printers with capabilities from the traditional and arbitrary-choice

PPLs. Second, we evaluate the Racket code formatter, which uses SnowWhite as its foundation. The evaluation aims to answer the following questions:

1. Does SnowWhite run fast in practice?

2. Does SnowWhite produce pretty layouts in practice?

All experiments are performed on a (non-poisoned) Apple M1 MacBook Pro with 16GB of RAM. We describe the experiments and benchmarks in Section 4.7.1 and Section 4.7.2, and discuss the results in Section 4.7.3.

4.7.1 Comparison of printers

We compare OCaml SnowWhite against the latest version (1.2.1) of Wadler/Leijen's printer, and the "camera ready version" of Bernardy's printer[7]. This "camera ready version" consists of two printers: the "naïve" variant, which is presented in the paper, and the "practical" implementation, which has more features (such as unavoidable overflow handling) but suffers from the exponential time complexity when the DAG structure unfolds, as discussed in Section 4.2. We manually remove the capability to customize the width limit from the latter to avoid the issue. Both variants are used for the evaluation, since the naïve variant does not have necessary features for some benchmarks.

SnowWhite is instantiated with the cost factory in Section 4.6, with the page width limit of 80 (unless indicated otherwise). We run SnowWhite twice with different computation width limits (once with $W_{\mathcal{F}} = 100$, unless indicated otherwise, and once with $W_{\mathcal{F}} = 1000$), in order to observe the effect of the tainting system and how it affects the performance.

[7]We also tried other versions of Bernardy's printer, such as the commit 006fa0e8, which is the version right before the **<|>** operator was removed, and supposedly more optimized than the camera ready version. Unfortunately, we find that it has a severe performance deficiency. When attempting to replicate the experiments in Bernardy [8], we find that formatting the 10k-line-JSON file takes about 80 seconds, which is much slower than the 145 milliseconds reported in the paper.

Table 4.2: Comparison between SNOWWHITE in different configurations and other printers. For each printer and configuration, the first column reports the running time, and the second column reports the line count of the output layout. SNOWWHITE has an additional third column, where ✓ indicates that the output layout fits $W_{\mathcal{F}}$ and ✗ indicates that the output layout is tainted. "N/A" means the benchmark is not applicable. ⏱ indicates that running the benchmark exceeds the timeout of 60 seconds. "-" means the data is not collected. A grayed row indicates an output mismatch among the printers/configurations. The bolded line count signals that in our manual inspection, the associated layout is the prettiest.

Benchmark	SNOWWHITE default $W_{\mathcal{F}}$ (usually 100)			SNOWWHITE $W_{\mathcal{F}} = 1000$			Wadler/Leijen		Bernardy Naïve		Bernardy Practical	
Concat10k	0.89 ms	1	✗	0.91 ms	1	✗	2.05 ms	1	N/A	-	548.14 ms	1
Concat50k	10.58 ms	1	✗	10.55 ms	1	✗	11.55 ms	1	N/A	-	17.18 s	1
FillSep5k	13.51 ms	668	✓	13.71 ms	668	✓	12.34 ms	668	3.32 s	668	⏱	-
FillSep50k	264.34 ms	6834	✓	262.99 ms	6834	✓	73.16 ms	6834	⏱	-	⏱	-
Flatten8k	42.10 ms	7986	✓	47.15 ms	7986	✓	3.33 s	7986	N/A	-	N/A	-
Flatten16k	94.86 ms	15986	✓	92.62 ms	15986	✓	22.14 s	15986	N/A	-	N/A	-
SExpFull15	3.28 s	4107	✓	8.55 s	4107	✓	60.44 ms	4107	678.92 ms	4107	929.26 ms	4107
SExpFull16	5.51 s	8246	✓	21.79 s	8246	✓	98.79 ms	8246	1.35 s	8246	1.83 s	8246
RandFit1k	130.97 ms	**629**	✓	243.59 ms	**629**	✓	15.16 ms	943	51.68 ms	**629**	80.17 ms	**629**
RandFit10k	1.13 s	**7861**	✓	4.66 s	**7861**	✓	42.65	10459	582.83 ms	**7861**	902.99 ms	**7861**
RandOver1k	88.06 ms	**1531**	✗	958.34 ms	**1531**	✓	7.27 ms	1635	N/A	-	73.95 ms	1105
RandOver10k	486.81 ms	**15027**	✗	13.98 s	**15027**	✓	135.75 ms	16015	N/A	-	1.18 s	7953
JSON1k	2.13 ms	564	✓	2.25 ms	564	✓	1.87 ms	564	N/A	-	5.02 ms	564
JSON10k	27.30 ms	5712	✓	25.86 ms	5712	✓	23.06 ms	5712	N/A	-	106.73 ms	5712
JSONW	2.15 ms	**721**	✗	2.20 ms	**721**	✓	8.52 ms	**721**	N/A	-	5.44 ms	709

Table 4.3: The code formatter benchmarks. The table is in the same format as the SNOWWHITE column in Table 4.2.

Benchmark	$W_{\mathcal{F}} = 100$			$W_{\mathcal{F}} = 1000$		
`class-internal`	651 ms	5750	✗	665 ms	**5749**	✓
`xform`	796 ms	5154	✗	819 ms	5154	✓

Benchmark	$W_{\mathcal{F}} = 100$			$W_{\mathcal{F}} = 1000$		
`list`	98 ms	993	✓	90 ms	993	✓
`hash`	8 ms	83	✓	9 ms	83	✓

The benchmarks (Table 4.2) are mostly taken from Bernardy [8], and we add a few more to test basic constructs. While Leijen's printer is expressive enough to handle all benchmarks (due to the inclusion of `align` to support aligned concatenation in addition to constructs from the traditional PPL), Bernardy's printers are not applicable to benchmarks that require constructs from the traditional PPL. Furthermore, Bernardy's naïve printer is not applicable to benchmarks that require extra features like unavoidable overflow handling.

In more detail, the benchmarks test the following kinds of documents:

Concat benchmarks test a long chain of concatenations, which are identified by Peyton-Jones [72] as a source of quadratic time complexity in Hughes' printer.

FillSep benchmarks test the `fillSep` construct (also known as `fill`), which fills lines with words as long as they fit.

Flatten benchmarks test repeated flattening, as shown in Figure 4.4.

SExpFull benchmarks are the last two data points from the "full tree" benchmark in Bernardy [8]'s paper. They create complete binary trees and print them as S-expressions.

RandFit benchmarks [8] are similar to SExpFull, but use random Dyck paths to generate random trees and filter only those that fit within the page width limit.

RandOver benchmarks are like RandFit with the opposite filtering.

JSON benchmarks are also from Bernardy [8]'s paper. They format large JSON files.

JSONW benchmark is the same as JSON1k but with a page width limit of 50 instead of 80, and we further adjust SnowWhite's default $W_{\mathcal{F}}$ from 100 to 60 to test the tainting system.

4.7.2 Racket code formatter

We evaluate the effectiveness of a Racket code formatter that uses the Racket SnowWhite as its foundation. Racket [36] is a programmable programming language. Its main syntax is S-expression, but this can be customized via its `#lang` protocol to read an arbitrary syntax. Even in the S-expression syntax, users can define custom forms via the macro system. Our long-term plan for the code formatter is to make it extensible to support any syntax and custom forms. SnowWhite is thus a natural choice as a foundational printer, due to its expressiveness.

The code formatter currently supports only S-expression formatting. However, the task is already challenging. While the S-expression syntax may look simple and uniform, Racket users employ a variety of styles for different forms to make them look distinctive in order to improve readability. Each function application, for example, has three possible styles (while most languages have two function application styles). The search space of the code formatter is thus quite large.

The benchmarks (Table 4.3) consist of files of different sizes from the Racket language codebase[8]. `class-internal` and `xform` are the two largest files. We use the code formatter to format these files with the page width limit of 80. We run the code formatter twice, once with $W_{\mathcal{F}} = 100$ and once with $W_{\mathcal{F}} = 1000$.

4.7.3 Results

Performance The benchmarking results in Table 4.2 and Table 4.3 show that overall, SNOWWHITE is sufficiently fast in practice. While not the fastest, it can process large, practical workloads `class-internal` and `xform` under a second. Furthermore, it provides a performance guarantee even on tricky inputs. The same is not true for other printers. The Flatten benchmarks work very poorly for Wadler's printer, and the FillSep benchmarks work very poorly for Bernardy's printer. Interestingly, Bernardy's naïve printer is faster than its practical variant, even though the latter is more optimized; this is due to the extra features that the practical printer needs to support. SNOWWHITE, by contrast, is set to support these features from the start.

We note two interesting observations on SNOWWHITE. First, it performs poorly on SExpFull relative to other printers. This is due to the memory pressure from memoization. Better engineering effort may be able to alleviate this issue. Second, although the time complexity of Π_e is $O(nW_{\mathcal{F}}^4)$, this worst case behavior happens only if Pareto frontiers are always full. In practice, this is not the

[8] `https://github.com/racket/racket/tree/master/racket/collects` at commit 4f1a2bd4

case[9], as evidenced by the fact that increasing $W_{\mathcal{F}}$ tenfold does not multiply the running time by 10^4. On the contrary, increasing $W_{\mathcal{F}}$ does not affect the running time at all on most benchmarks.

Optimality We find that SnowWhite is the prettiest compared to others, offering high quality output when we use the cost factory described in Section 4.6. Table 4.2 shows (via line count) that the output layouts in many benchmarks agree in all printers. The exceptions are RandFit, RandOver, and JSONW benchmarks. Upon manual inspection, we find that the layouts produced by SnowWhite are better. JSONW and RandOver are cases where there is an unavoidable overflow, causing Bernardy's printer to overflow more than necessary. Figure 4.6 shows a concrete example of this problem. RandFit and RandOver are cases where the greedy minimization and the `align` construct in Leijen's printer interact poorly, as discussed and illustrated in Bernardy [8].

It should also be noted that neither Leijen's nor Bernardy's printers support custom optimality objectives, as their optimality objectives are integral to their algorithms. SnowWhite, in contrast, allows users to customize optimality objective via the cost factory.

Lastly, we evaluate the effectiveness of the tainting system. For almost every benchmark that gets a tainted layout (✗) with the default $W_{\mathcal{F}}$, we find that using $W_{\mathcal{F}} = 1000$ in an attempt to avoid taintedness[10] yields the same result, confirming the optimality of the output layout. The only exception is the `class-internal` benchmark in Table 4.3, for which the output layouts are different in one line and otherwise identical, because the greedy heuristic in the taint operation prunes the optimal choice away. This demonstrates that despite being tainted, and thus no longer guaranteed to be optimal, the output layout is still reasonable (at least with respect to the cost factory that we employ and the heuristic to avoid bias described in Section 4.8).

[9]This observation also applies to Bernardy's printers, which are also based on Pareto frontiers.

[10]Therefore, the Concat benchmarks do not count, since they are still tainted afterwards. The benchmarks are not interesting anyway, since there is no choice in the documents, so the output layouts are always optimal.

4.8 Discussion

In this section, we broadly discuss the design of our work.

4.8.1 Additional constructs

SNOWWHITE supports additional constructs `fail` and `flat`. The Racket SNOWWHITE further supports additional constructs `full` and `cost`. These constructs are out of scope for the work, and we leave their formalization as future work.

Failure The `fail` construct widens to the empty set, thus introducing the possibility that a printing could fail. Furthermore, it is the identity for the operation `<|>`. `fail` makes Σ_e more expressive because it is impossible to make Σ_e evaluate to the empty set. In this sense, it could be said that Σ_e is not truly "functionally complete," but Σ_e with `fail` is. Supporting `fail` can be done via rewriting rules: every document with `fail` can be normalize to a semantically-equivalent document without `fail`, or to a single `fail`. Hence, there is no need to modify the core printer to add the construct.

Flatness `fail` paves way to support another construct `flat` d, which is also implemented in Bernardy [7]'s practical printer as *single line*[11]. The construct adds a constraint to d to eliminate layouts with multiple lines. This is especially very useful as an addition to the arbitrary-choice PPL, since it can eliminate the "ladder" layouts (which are undesirable) that result from multiple aligned concatenations. Π_e can support `flat` in the same way it supports `flatten` (Section 4.5.8), although we would replace `nl` with `fail` instead of a single space. Note that `flat` interacts with `flatten` in the intended way. For example, `flat (flatten (text "\n"))` evaluates to {`" "`}, but `flatten (flat (text "\n"))` evaluates to the empty set. The construct presents a challenge for

[11]This is one of the extra features that is supported in Bernardy's practical printer but not by the naïve printer.

us to state the correctness of the printer, because a widened choiceless document may now fail to render.

Fullness `full` d marks d as *full*, which means there must be no more text afterward in the same line. The construct is especially useful for code formatters that need to support line comments, as it is illegal to collapse a piece of code after a comment. A simpler variant of `full` is also implemented in Yelland's printer for the R code formatter [103]. Unlike `fail` and `flat`, which can be supported via rewriting rules to avoid the core printer modification, `full` requires involved changes to the core printer.

- The measure set definition is now required to recognize the empty set (where we prefer a tainted measure set over an empty set).
- The resolver would consume additional two boolean arguments, which indicate the fullness status before and after the document.
- Merging two tainted measure sets must now keep both tainted measure sets, and we may need to try both if the first one resolves to the empty set.

To keep the time complexity of the printer $O(nW^4_{\mathcal{F}})$, we rely on the fact that "emptiness" in resolving (that is, resolving to the empty set) is independent from column positions and indentation levels. Thus, even though we now need to try many tainted measure sets, a document can be tried at most four times (one for each before and after fullness statuses), which bounds the time complexity.

Cost `cost` C d adds a cost C to measures due to d. This construct is not expressive in the traditional sense, as it does not affect layout results. However, it allows us to make *weighted*

choices, so that we can prefer one style over another when all else is equal. Due to the flexibility of the cost factory, it is even possible to make multidimensional weights.

4.8.2 Safety

As shown in the proof of Lemma 4.2, the traditional PPL is not functionally complete because all layouts must have the same content, modulo whitespaces. While this property is restrictive for many tasks as elaborated in this chapter, it does provide a sort of safety guarantee that the layouts will not be wildly different. `<|>`, however, allows us to violate this property. In fact, some arbitrary-choice printers (e.g. a prototype of Bernardy's printer [5]) *intend* that `<|>` should be restricted to maintain the property. Similarly, the inclusion of `fail`, `flat`, or `full` makes it possible to evaluate to an empty set, but the PPLs without the essence of `fail` provide a safety guarantee that an evaluation will never result in an empty set. Generally, the more expressive a language is, the more properties it will break, and the more burden will be put on the users to carefully use the constructs.

We argue that the spirit of these safety properties can still be accomplished in PPLs with a functionally complete core. One possible approach is similar to Wadler's treatment of `<|>` and `group`: define high-level, "safe" constructs with just enough expressiveness to solve a domain-specific task, based on the core, "unsafe" constructs, and then hide these "unsafe" constructs away from the external interface. For example, one may hide `<|>`, and instead provide groupParen(d) = `(text "(" <>` d `<> text ")") <|> flatten` d, which evaluates to either d with parentheses wrapped around, or the flattened d. The language as defined by the external interface is no longer functionally complete, but enjoys the property that all layouts are the same modulo whitespaces and parentheses. Another possible approach is to export the core, "unsafe" constructs, but perform a static analysis to make sure that the document satisfies the intended safety property.

In any case, the expressive core constructs are what enable the advanced features that may be required by tasks. Thus, our view is that an expressive printer is the key. We should start with an expressive albeit unsafe printer, rather than a safe but non-expressive one.

4.8.3 Memoization

While memoization is important to guarantee that Π_e will not take exponential time, it is also the performance bottleneck when the input document is large, due to too much memory allocation. In SnowWhite, we employ a heuristic to reduce memory allocation by adding a metadata *memoization weight* to each document node, which counts how long memoization has not been performed on descendant nodes. When the weight reaches a limit (set to 6 in our implementation), we perform memoization on the node, and reset the weight to 0. This can significantly speeds up the performance of SnowWhite in some large documents.

4.8.4 Fusing resolving and rendering

One optimization in SnowWhite is to fuse resolving a document to a measure set and rendering of a choiceless document to a layout together. This is done by replacing the doc component in a measure with a *token function*, which consumes a list of rendered tokens *after* the document is placed, and returns a new list of rendered tokens. A similar technique was employed by Podkopaev and Boulytchev [75]. For example, the token function on printing a `nl` at the indentation level i would be toks $\mapsto$ `["\n", " "` $\times i$`]@ toks`. As another example, the token function on printing a `<>` would be toks $\mapsto f_1(f_2(\text{toks}))$, where f_1 and f_2 are the token functions from the left and right measures to be concatenated together. The printer would then pick a measure with the lowest cost, and invoke the corresponding token function with an empty list to obtain the full list of rendered tokens, which could then be displayed to users.

4.8.5 Handling bias in presence of taintedness

In Section 4.5, we see that the merge operation and thus the `<|>` operator is left-biased in presence of taintedness. When exceeding $W_{\mathcal{F}}$ is unavoidable, all text could be put in one line in the worst case if all left sub-documents use the "horizontal styling"! The proper solution is to increase $W_{\mathcal{F}}$. However, SNOWWHITE also implements a heuristic to infer a sub-document with the "vertical styling." The heuristic adds a metadata *estimated number of lines* (estl) to each document node. For example, we would have that $\mathrm{estl}(\texttt{nl}) = 1$, $\mathrm{estl}(\texttt{text}\ s) = 0$, $\mathrm{estl}(d_a \texttt{<>} d_b) = \mathrm{estl}(d_a) + \mathrm{estl}(d_b)$, and $\mathrm{estl}(d_a \texttt{<|>} d_b) = \max(\mathrm{estl}(d_a), \mathrm{estl}(d_b))$. SNOWWHITE then uses a document with a larger estl as the left sub-document in choice documents.

4.8.6 Rewriting rules and cost factory

Our cost factory is minimally specified and only concerns with the correctness of the core printer. Therefore, if one wishes to employ rewriting rules to transform a document to another semantically-equivalent document, one must make sure that the specified cost factory admits the rewriting rules. For example, if we want to rewrite a document D `<> text ""` to simply D for any D, it would require, at the very least, that the $+_{\mathcal{F}}$ operation has an identity element, and that the cost of `text ""` is the identity element. Associativity and commutativity of $+_{\mathcal{F}}$ are another examples of properties that ordinary cost factories should satisfy.

4.8.7 Rewriting rules and partial evaluation

Similar to how we can perform partial evaluation in programming languages, we can also perform partial evaluation in PPL using rewriting rules. For example, a concatenation of two `text` can be partially evaluated to a single `text` right away. However, this partial evaluation must be done with care to still preserve the sharing structure, since unconstrained rewriting may unfold the DAG

```
(define (bvmul2_?? x)
  (bvshl x (?? (bitvector 8))))

(define sol
  (synthesize
    #:forall (list x)
    #:guarantee
    (assert
     (equal? (bvmul2_?? x)
             (bvmul x (bv 2 8))))))

(print-forms sol)
```

```
(define (bvmul2_?? x) (bvshl x (bv #x01 8)))
```

```
(define (bvmul2_?? x)
  (bvshl x (bv #x01 8)))
```

Figure 4.19: An example solver-aided program in Rosette, taken from the documentation of Rosette's solver-aided synthesis library [89], along with the outputs before and after our change.

structure into a tree. It is also worth noting that the partial evaluation may not necessarily preserve the semantics in presence of taintedness. For example, one may want to reduce a **nest** n (**text** s) to **text** s for any n and s, but when $n > W_{\mathcal{F}}$, the document will definitely resolves to a tainted measure set, while the partially evaluated one does not necessarily[12].

4.9 Applications of Π_e in solver-aided programming

We have shown the utilities of Π_e as a standalone pretty printer. In this section, we demonstrate how Π_e can improve productivity in interfacing with end users in solver-aided programming via two case studies. The first case study is on the default Rosette's synthesis output, which was previously produced by an ad-hoc pretty printer. The second case study is on MemSynth's output, which was previously produced without any pretty printing.

4.9.1 Rosette

Rosette, as an implementation of $\mathcal{S}_c$ that extends Racket (Section 2.6.2), allows programmers to formulate synthesis queries with its language constructs. Figure 4.19 illustrates a solver-aided program that synthesizes a function that multiplies its input by 2 with the requirement that it must use the left shift operation. The program first defines a function `bvmul2_??` with a syntactic hole `??`. Then, it asserts that `bvmul2_??` is the double function on all inputs. This generates queries to be submitted to the solver in order to find a constant to fill in the syntactic hole. Once a solution is found, programmers would need to display the synthesized code so that end users (who could be the programmers themselves) can understand the solution. Because the syntactic hole is from the Rosette solver-aided synthesis library [89], Rosette knows how to emit the synthesized code as a Racket code when programmers call the function `print-forms`.

The synthesized outputs by Rosette, however, generally do not follow the style that Racket users tend to write. When the above program is run, it outputs the code shown in the top right of Figure 4.19, which does not have a newline after the function header—a common Racket coding convention. As a result, the code could be difficult to read to end users. The underlying issue is that `print-forms` employs Racket's ad-hoc pretty printing library, which has limited capability and customization. From Rosette and programmers' perspective, there is not much adjustment that could be done to the Racket library to improve the output. The issue shows that it can be challenging to provide high-quality output with an ad-hoc pretty printer.

To address this issue, we replaced the ad-hoc pretty printer with our Racket code formatter, which employs Π_e under the hood. The code formatter transforms a Racket syntax tree to a document in Σ_e, with extensibility points so that programmers can customize the document construction to control the styling. This high flexibility can only be effective when the underlying

[12] One may argue, however, that this semantic change is acceptable, because the change is for the better.

```
(((((rf . ((po :> Syncs) . po)) + (((po :> Syncs) . po) . rf)) + ((((po :> Syncs) .
   ↪  po) :> Writes) + (((po :> Syncs) . po) :> Reads))) :> (((Writes + Reads) . (
   ↪ rf + ((po :> Syncs) . po))) + ((((po :> Syncs) . po) . Reads) :> Reads))) +
   ↪ ((((Reads <: (((Writes -> Writes) + (Reads -> MemoryEvent)) & ((po :> Lwsyncs
   ↪ ) . po))) + ((((Writes -> Writes) + (Reads -> MemoryEvent)) & ((po :> Lwsyncs
   ↪ ) . po)) :> Reads)) :> ((Writes + Reads) <: Reads)) + (((((Writes -> Writes)
   ↪ + (Reads -> MemoryEvent)) & ((po :> Lwsyncs) . po)) + (rf . (((Writes ->
   ↪ Writes) + (Reads -> MemoryEvent)) & ((po :> Lwsyncs) . po)))) :> ((Writes +
   ↪ Reads) . ((((Writes -> Writes) + (Reads -> MemoryEvent)) & ((po :> Lwsyncs) .
   ↪  po)) :> Writes)))))
```

(a) Before our change

```
((rf.((po :> Syncs).po) + ((po :> Syncs).po).rf) +
 ((po :> Syncs).po :> Writes + (po :> Syncs).po :> Reads)) :>
((Writes + Reads).(rf + (po :> Syncs).po) + ((po :> Syncs).po).Reads :> Reads) +
((Reads <: ((Writes->Writes + Reads->MemoryEvent) & (po :> Lwsyncs).po) +
  ((Writes->Writes + Reads->MemoryEvent) & (po :> Lwsyncs).po) :> Reads) :>
 ((Writes + Reads) <: Reads) +
 ((Writes->Writes + Reads->MemoryEvent) & (po :> Lwsyncs).po +
  rf.((Writes->Writes + Reads->MemoryEvent) & (po :> Lwsyncs).po)) :>
 (Writes + Reads).(
   ((Writes->Writes + Reads->MemoryEvent) & (po :> Lwsyncs).po) :> Writes
 ))
```

(b) After our change

Figure 4.20: Synthesized outputs from MemSynth in the Alloy* syntax before and after our change. The red arrow indicates line wrapping.

pretty printer is expressive and optimal, and Π_e satisfies the requirement. After our change, the above Rosette program produces the output shown in the bottom right of Figure 4.19, which follows the Racket coding convention. Thus, the case study shows that Π_e is a suitable tool for improving the productivity in solver-aided programming.

4.9.2 MemSynth

MemSynth [13] is a solver-aided program for synthesizing memory models from framework sketches and litmus tests, where a memory model is represented with a set of axioms in relational logic. Because many existing memory model tools employ Alloy* specification [58] as a programming language to write the formula in relational logic, MemSynth follows the approach

for interoperability.

An example synthesized output is shown in Figure 4.20a. Unlike Rosette's default synthesis output, which could have newlines to avoid exceeding the page width limit, the output of MemSynth does not concern with the page width limit at all, putting all text in one line. Producing an output in one line is a common approach in many solver-aided programs due to how easy it is to implement, and how it could be substantially more difficult to produce a human-readable output. However, it evidently results in an output code that is difficult to read for end users.

By employing Π_e, we can add newlines and indentation to the synthesized output, with only a few change to MemSynth's printing facility. As a result, the output after our change is more readable, as shown in Figure 4.20b. This demonstrates how Π_e facilitates the development of an effective solver-aided program to interface with end users.

4.10 Conclusion

We have described Π_e, an expressive printer that supports a variety of optimality objectives and is practically efficient. We developed a framework for reasoning about the expressiveness of PPLs, and we used this framework to guide the design of the PPL that Π_e targets. By surveying existing pretty printers, we have shown that Π_e is well-placed in the design space of printers. Π_e is proven correct in the Lean theorem prover and implemented as a practical printer SNOWWHITE, which powers a real-world code formatter for the Racket programming language. Our results show that SNOWWHITE (and Π_e) is both pretty and fast.

In the context of solver-aided programming, we showed how human-readable output is desirable, but often difficult to achieve due to the difficulty to develop an effective pretty printer. Our case studies demonstrated how Π_e can be employed to address the issue, improving the productivity in interfacing with end users.

Chapter 5

Conclusion

This dissertation has demonstrated that it is possible to develop effective productivity tools for solver-aided programming by making them recontextualizing and extensible. In Chapter 2, we presented a formal foundation for symbolic evaluation with merging, whose derived reusable symbolic evaluator enhances productivity in programming to interact with solvers. In Chapter 3, we presented a profile-guided symbolic optimizer, which improves productivity in scaling up solver-aided programs. Lastly, in Chapter 4, we presented an expressive pretty printer, enhancing productivity in interfacing with end users. Along with existing productivity tools, our three new tools support the thesis that recontextualization and extensibility are keys to developing productivity tools for solver-aided programming.

There are still many potential for productivity improvement in the space of solver-aided programming. One notable stage of solver-aided programming that is not covered in this dissertation is debugging. Bugs in solver-aided programming often manifest as runtime exceptions. Unlike traditional programming, the runtime exceptions are treated as assertion failures in the solver-aided host language, and the all-path evaluation would prune paths that lead to the assertion failures away. As a result, traditional debugging tools like error tracer [38] are unviable for debugging.

In our preliminary work, we developed a *symbolic error tracer* [91], which is a recontextualization of the traditional error tracer in traditional programming. While the tool is currently a prototype, the early feedback shows that programmers who use the tool already find it useful for debugging their solver-aided programs, although the tool is not yet scalable to work on large solver-aided programs, limiting its usefulness. Future work could pursue how to make the tool more scalable. In any case, we hope that the development of future tools will benefit from the guiding principles at the core of this dissertation.

Bibliography

[1] Amazon Web Services. 2018. Quivela. https://github.com/awslabs/quivela

[2] Pablo R Azero Alcocer and S Doaitse Swierstra. 1998. Optimal pretty-printing combinators. https://web.archive.org/web/20040911044443/http://www.cs.uu.nl/groups/ST/Software/PP/pabloicfp.ps.

[3] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV) (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 171–177. https://doi.org/10.1007/978-3-642-22110-1_14

[4] Nils Becker, Peter Müller, and Alexander J. Summers. 2019. The Axiom Profiler: Understanding and Debugging SMT Quantifier Instantiations. In *Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Prague, Czech Republic, 99–116.

[5] Jean-Philippe Bernardy. 2015. Towards The Prettiest Printer. https://jyp.github.io/posts/towards-the-prettiest-printer.html.

[6] Jean-Philippe Bernardy. 2017. Disjunctionless. https://github.com/jyp/prettiest/pull/10.

[7] Jean-Philippe Bernardy. 2017. prettiest. https://github.com/jyp/prettiest/blob/5e7a12cf37bb01467485bbe1e9d8f272fa4f8cd5/Text/PrettyPrint/Compact/Core.hs.

[8] Jean-Philippe Bernardy. 2017. A Pretty but Not Greedy Printer (Functional Pearl). *Proc. ACM Program. Lang.* 1, ICFP, Article 6 (Aug. 2017), 21 pages. https://doi.org/10.1145/3110250

[9] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking Without BDDs. In *Proceedings of the 5th International Conference on Tools*

and Algorithms for the Construction and Analysis of Systems (TACAS). Amsterdam, The Netherlands, 193–207.

[10] Jasmin Christian Blanchette, Mathias Fleury, and Christoph Weidenbach. 2017. A Verified SAT Solver Framework with Learn, Forget, Restart, and Incrementality. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*. Melbourne, Australia, 4786–4790. https://doi.org/10.24963/ijcai.2017/667

[11] Rastislav Bodik, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. 2010. Programming with Angelic Nondeterminism. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Madrid, Spain, 339–352. https://doi.org/10.1145/1706299.1706339

[12] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. 2016. Specifying and checking file system crash-consistency models. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Atlanta, GA, USA, 83–98.

[13] James Bornholt and Emina Torlak. 2017. Synthesizing Memory Models from Framework Sketches and Litmus Tests. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Barcelona, Spain, 467–481.

[14] James Bornholt and Emina Torlak. 2018. Finding Code That Explodes Under Symbolic Evaluation. *Proc. ACM Program. Lang.* OOPSLA, Article 149 (Oct. 2018), 149:1–149:26 pages.

[15] Alan Borning. 2016. Wallingford: Toward a Constraint Reactive Programming Language. In *Proceedings of the Constrained and Reactive Objects Workshop (CROW)*. Málaga, Spain.

[16] Stefan Bucur, Johannes Kinder, and George Candea. 2014. Prototyping symbolic execution engines for interpreted languages. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Salt Lake City, UT, USA, 239–254.

[17] Eric Butler, Emina Torlak, and Zoran Popović. 2017. Synthesizing Interpretable Strategies for Solving Puzzle Games. In *Proceedings of the 12th International Conference on the Foundations of Digital Games (FDG)*. Hyannis, MA, USA.

[18] Cristian Cadar. 2015. Targeted program transformations for symbolic execution. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Bergamo, Italy, 906–909.

[19] Kartik Chandra and Rastislav Bodik. 2018. Bonsai: Synthesis-Based Reasoning for Type Systems. *Proc. ACM Program. Lang.* 2, POPL (Jan. 2018), 62:1–62:34.

[20] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 265–278.

[21] Olaf Chitil. 2005. Pretty Printing with Lazy Dequeues. *ACM Trans. Program. Lang. Syst.* 27, 1 (jan 2005), 163–184. https://doi.org/10.1145/1053468.1053473

[22] Adam Chlipala. 2010. A Verified Compiler for an Impure Functional Language. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Madrid, Spain, 93–106. https://doi.org/10.1145/1706299.1706312

[23] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL. In *Proceedings of the 8th Biennial Conference on Innovative Data Systems (CIDR)*. Chaminade, CA, USA.

[24] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Barcelona, Spain, 168–176.

[25] Edmund Clarke, Daniel Kroening, and Karen Yorav. 2003. *Behavioral Consistency of C and Verilog Programs*. Technical Report CMU-CS-03-126. Carnegie Mellon University. https://doi.org/10.1145/775832.775928

[26] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. 1999. *Model Checking*. MIT Press, Cambridge, MA, USA.

[27] Lori A. Clarke. 1976. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering* 2, 3 (1976), 215–222.

[28] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Los Angeles, CA, 238–252. https://doi.org/10.1145/512950.512973

[29] Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *Proceedings of the 6th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. San Antonio, TX, 269–282. https://doi.org/10.1145/567752.567778

[30] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt, Matt Kaufmann, and Peter Schneider-Kamp. 2017. Efficient Certified RAT Verification. In *Proceedings of the 26th International Conference on Automated Deduction (CADE)*. Gothenburg, Sweden, 220–236.

[31] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Budapest, Hungary, 337–340.

[32] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover. In *Proceedings of the 25th International Conference on Automated Deduction (CADE)*. Berlin, Germany, 378–388.

[33] Nachum Dershowitz and Jean-Pierre Jouannaud. 1990. Rewrite Systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. 243–320.

[34] ESLint. 2014. Change no-comma-dangle to comma-dangle. `https://github.com/eslint/eslint/issues/1350`.

[35] Matthias Felleisen. 1991. On the expressive power of programming languages. *Science of Computer Programming* 17, 1 (1991), 35–75. `https://doi.org/10.1016/0167-6423(91)90036-W`

[36] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2018. A Programmable Programming Language. *Commun. ACM* 61, 3 (March 2018), 62–71. `https://doi.org/10.1145/3127323`

[37] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the 14th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Albuquerque, NM, 237–247. `https://doi.org/10.1145/173262.155113`

[38] Matthew Flatt. [n. d.]. Errortrace: Debugging and Profiling. http://docs.racket-lang.org/errortrace/.

[39] Matthew Flatt and PLT. 2010. *Reference: Racket*. Technical Report PLT-TR-2010-1. PLT Design Inc.

[40] Matt Fleming. 2017. A thorough introduction to eBPF. `https://lwn.net/Articles/740157/`.

[41] José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. 2020. Gillian, Part I: A Multi-Language Platform for Symbolic Execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. London, United Kingdom, 927–942. `https://doi.org/10.1145/3385412.3386014`

[42] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009.

Trace-based Just-in-time Type Specialization for Dynamic Languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Dublin, Ireland, 465–478.

[43] Galois, Inc. 2018. Crucible. `https://github.com/GaloisInc/crucible`

[44] Andrew John Gill and Simon L. Peyton Jones. 1994. Cheap Deforestation in Practice: An Optimizer for Haskell. In *IFIP Congress*.

[45] Rajiv Gupta, Eduard Mehofer, and Youtao Zhang. 2002. Profile Guided Compiler Optimizations. In *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, Chapter 4.

[46] John Hughes. 1995. The design of a pretty-printing library. In *Advanced Functional Programming*, Johan Jeuring and Erik Meijer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 53–96.

[47] 2015. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation. Revision 53.

[48] Jacques-Henri Jourdan. 2016. *Verasco: a Formally Verified C Static Analyzer*. Theses. Universite Paris Diderot-Paris VII. `https://hal.archives-ouvertes.fr/tel-01327023`

[49] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. 2015. A Formally-Verified C Static Analyzer. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Mumbai, India, 247–259. `https://doi.org/10.1145/2676726.2676966`

[50] Milod Kazerounian, Niki Vazou, Austin Bourgerie, Jeffrey S. Foster, and Emina Torlak. 2018. Refinement Types for Ruby. In *Proceedings of the 19th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Los Angeles, CA, USA, 269–290.

[51] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394.

[52] Oleg Kiselyov, Simon Peyton-Jones, and Amr Sabry. 2012. Lazy v. Yield: Incremental, Linear Pretty-Printing. In *Programming Languages and Systems*, Ranjit Jhala and Atsushi Igarashi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 190–206.

[53] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient State Merging in Symbolic Execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Beijing, China, 89–98.

[54] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO)*. Palo Alto, California.

[55] Daan Leijen. 2000. wl-pprint: The Wadler/Leijen Pretty Printer. `https://hackage.haskell.org/package/wl-pprint`.

[56] Dorel Lucanu, Vlad Rusu, and Andrei Arusoaie. 2017. A generic framework for symbolic execution: A coinductive approach. *Journal of Symbolic Computation* 80 (May–June 2017), 125–163. `https://doi.org/10.1016/j.jsc.2016.07.012`

[57] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[58] Aleksandar Milicevic, Joseph P. Near, Eunsuk Kang, and Daniel Jackson. 2015. Alloy*: A General-Purpose Higher-Order Relational Constraint Solver. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 609–619. `https://doi.org/10.1109/ICSE.2015.77`

[59] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28*, André Platzer and Geoff Sutcliffe (Eds.). Springer International Publishing, Cham, 625–635.

[60] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. 225–242.

[61] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. 2020. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel. In *Proceedings of the 14th Symposium on Operating Systems Design and Implementation (OSDI)*. Banff, Alberta, Canada, 41–61.

[62] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*. 252–269.

[63] Max S. New, Burke Fetscher, Robert Bruce Findler, and Jay McCarthy. 2017. Fair enumeration combinators. *Journal of Functional Programming* 27 (2017), e19. `https://doi.org/10.1017/S0956796817000107`

[64] Julie L Newcomb and Rastislav Bodik. 2019. Using human-in-the-loop synthesis to author functional reactive programs. arXiv:1909.11206 [cs.PL]

[65] Phúc C. Nguyên, Sam Tobin-Hochstadt, and David Van Horn. 2017. Higher order symbolic execution for contract verification and refutation. *Journal of Functional Programming* 27 (2017), e3. https://doi.org/10.1017/S0956796816000216

[66] Aina Niemetz, Mathias Preiner, and Armin Biere. 2014. Boolector 2.0. *J. Satisf. Boolean Model. Comput.* 9, 1 (2014), 53–58. https://doi.org/10.3233/sat190101

[67] Dereck C. Oppen. 1980. Prettyprinting. *ACM Trans. Program. Lang. Syst.* 2, 4 (Oct. 1980), 465–483. https://doi.org/10.1145/357114.357115

[68] Susan Owicki and David Gries. 1976. Verifying Properties of Parallel Programs: An Axiomatic Approach. *Commun. ACM* 19, 5 (May 1976), 279–285. https://doi.org/10.1145/360051.360224

[69] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java HotSpot Server Compiler. In *Proceedings of the 2001 Symposium on Java Virtual Machine Research and Technology (JVM)*. Monterey, CA, USA.

[70] Stuart Pernsteiner, Calvin Loncaric, Emina Torlak, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Jonathan Jacky. 2016. Investigating Safety of a Radiotherapy Machine Using System Models with Pluggable Checkers. In *Proceedings of the 28th International Conference on Computer Aided Verification (CAV)*, Vol. 2. Toronto, ON, Canada, 23–41.

[71] Karl Pettis and Robert C. Hansen. 1990. Profile Guided Code Positioning. In *Proceedings of the 11th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. White Plains, NY, USA, 16–27.

[72] Simon Peyton-Jones. 1997. A pretty printer library in Haskell. https://web.archive.org/web/20080221052958/http://research.microsoft.com/Users/simonpj/downloads/pretty-printer/pretty.html. The identified mistakes are noted at https://github.com/haskell/pretty/blob/50b70d1be6e17a644dc3b5c80592cf7c5b339fd9/Text/PrettyPrint/HughesPJ.hs.

[73] Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. 2019. Swizzle Inventor: Data Movement Synthesis for GPU Kernels. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 65–78.

[74] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 2016. Scaling Up Superoptimization. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Atlanta, GA, USA, 297–310.

[75] Anton Podkopaev and Dmitri Boulytchev. 2015. Polynomial-Time Optimal Pretty-Printing Combinators with Choice. In *Perspectives of System Informatics*, Andrei Voronkov and Irina Virbitskaite (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 257–265.

[76] Sorawee Porncharoenwase, Luke Nelson, Xi Wang, and Emina Torlak. 2022. A Formal Foundation for Symbolic Evaluation with Merging. *Proc. ACM Program. Lang.* 6, POPL, Article 47 (Jan. 2022), 28 pages. `https://doi.org/10.1145/3498709`

[77] Prettier. 2016. Technical Details. `https://prettier.io/docs/en/technical-details.html`.

[78] S2E 2019. S2E: Exponential Analysis Speedup with State Merging. `http://s2e.systems/docs/StateMerging.html`

[79] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Saint Petersburg, Russian Federation, 488–498.

[80] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: multi-path symbolic execution using value summaries. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Bergamo, Italy, 842–853.

[81] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2015. Conditionally Correct Superoptimization. In *Proceedings of the 30th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Pittsburgh, PA, USA, 147–162.

[82] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. 2018. Nickel: A Framework for Design and Verification of Information Flow Control Systems. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*. 287–305.

[83] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. San Jose, CA, USA, 404–415.

[84] Vincent St-Amour, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Optimization Coaching: Optimizers Learn to Communicate with Programmers. In *Proceedings of the 27th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Tuscon, AZ, USA, 163–178.

[85] S Doaitse Swierstra, Pablo R Azero Alcocer, and Joao Saraiva. 1999. Designing and Implementing Combinator Languages. In *Third Summer School on Advanced Functional Programming, volume 1608 of LNCS*. Springer-Verlag, 150–206.

[86] Sol Otis Swords. 2010. *A verified framework for symbolic execution in the ACL2 theorem prover*. Ph. D. Dissertation. The University of Texas at Austin.

[87] The Python Language Reference. 2010. Lexical analysis. `https://docs.python.org/2.7/reference/lexical_analysis.html`.

[88] Emina Torlak. 2018. Rosette. `http://github.com/emina/rosette`

[89] Emina Torlak. 2019. The Rosette Guide: Solver-Aided Libraries. `https://docs.racket-lang.org/rosette-guide/sec_rosette-libs.html`

[90] Emina Torlak. 2019. The Rosette Guide: Symbolic Reflection. `https://docs.racket-lang.org/rosette-guide/ch_symbolic-reflection.html`

[91] Emina Torlak. 2020. The Rosette Guide: Debugging. `https://docs.racket-lang.org/rosette-guide/ch_error-tracing.html`

[92] Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Proceedings of the 2013 ACM Symposium on New Ideas in Programming and Reflections on Software (Onward!)*. Indianapolis, IN, USA, 135–152.

[93] Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Edinburgh, United Kingdom, 530–541. `https://doi.org/10.1145/2666356.2594340`

[94] Richard Uhler and Nirav Dave. 2014. Smten with Satisfiability-Based Search. In *Proceedings of the 29th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Portland, OR, USA, 157–176.

[95] Philip Wadler. 1988. Deforestation: Transforming Programs to Eliminate Trees. In *Proceedings of the Second European Symposium on Programming*. 231–248.

[96] Philip Wadler. 2003. A prettier printer. *The Fun of Programming, Cornerstones of Computing* (2003), 223–243.

[97] Jonas Wagner, Volodymyr Kuznetsov, and George Candea. 2013. -Overify: Optimizing Programs for Fast Verification. In *Proceedings of the 14th Workshop on Hot Topics in Operating Systems (HotOS)*. Santa Ana Pueblo, NM, USA.

[98] Andrew Waterman and Krste Asanović (Eds.). 2019. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. RISC-V Foundation.

[99] Konstantin Weitz, Steven Lyubomirsky, Stefan Heule, Emina Torlak, Michael D. Ernst, and Zachary Tatlock. 2017. SpaceSearch: A Library for Building and Verifying Solver-Aided Tools. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Functional Programming (ICFP)*. Oxford, United Kingdom, Article 25, 28 pages. https://doi.org/10.1145/3110269

[100] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. 2016. Scalable Verification of Border Gateway Protocol Configurations with an SMT Solver. In *Proceedings of the 31st ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Amsterdam, The Netherlands, 765–780.

[101] Max Willsey, Luis Ceze, and Karin Strauss. 2018. Puddle: An Operating System for Reliable, High-Level Programming of Digital Microfluidic Devices. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Wild and Crazy Ideas Session*. Williamsburg, VA, USA.

[102] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT Solvers via Semantic Fusion. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. London, United Kingdom, 718–730. https://doi.org/10.1145/3385412.3385985

[103] Phillip Yelland. 2015. rfmt: A code formatter for R. https://github.com/google/rfmt.

[104] Phillip Yelland. 2016. A New Approach to Optimal Code Formatting. Technical note for open source project rfmt; https://github.com/google/rfmt.