Less is more: refinement proofs for probabilistic proofs

Kunming Jiang,* Devora Chait-Roth, Zachary DeStefano, Michael Walfish, and Thomas Wies

NYU Department of Computer Science, Courant Institute  *Now at Carnegie Mellon

Abstract

There has been intense interest over the last decade in implementations of probabilistic proofs (IPs, SNARKs, PCPs, and so on): protocols in which an untrusted party proves to a verifier that a given computation was executed properly, possibly in zero knowledge. Nevertheless, implementations still do not scale beyond small computations. A central source of overhead is the front-end: translating from the abstract computation to a set of equivalent arithmetic constraints. This paper introduces a general-purpose framework, called Distiller, in which a user translates to constraints not the original computation but an abstracted specification of it. Distiller is the first in this area to perform such transformations in a way that is provably safe. Furthermore, by taking the idea of “encode a check in the constraints” to its literal logical extreme, Distiller exposes many new opportunities for constraint reduction, resulting in cost reductions for benchmark computations of 1.3–50×, and in some cases, better asymptotics.

1 Introduction

Probabilistic proofs [5–7, 38–40]—PCPs, IPs, NIZKs, SNARKs, SNARGs, and so on—are fundamental in complexity theory and cryptography. They enable an untrusted prover to convince a verifier of some statement (for example, that a given computation Ψ, on specific input x, produces an alleged output y). In these protocols, the verifier does not inspect a classical witness to the truth of the statement (or re-execute Ψ) but instead checks an encoded proof probabilistically. Zero-knowledge variants allow the prover to keep some of the input to the computation—and the proof itself—hidden from the verifier. Astonishingly, the verifier’s checks are (in some protocols) constant-time, regardless of the size of the computation [5, 6, 37]. The appeal of these properties in emerging application areas (most notably, outsourced computation, blockchains, and their intersection) has fueled intense interest in implementations over the last 13 years. The results have included 20 orders of magnitude reduction in costs, deployment of SNARKs in cryptocurrencies [31, 52, 68, 84], and an explosion of frameworks [75, 78, 81].

Yet, probabilistic proofs are heavily limited in scalability, making them impractical for general-purpose use (the hype notwithstanding). One source of costs is the back-end, which is the complexity-theoretic and cryptographic proving machinery. The other source of costs is the front-end, which translates high-level computations into the format that the back-end works over. In most probabilistic proof implementations, that format is some variant of arithmetic constraints: equations over a finite field.

Unfortunately, not only must the prover perform cryptographic operations proportional to the number of constraints (often with memory requirements that scale similarly), but also constraints are a verbose way to represent computations (§2). For example, every iteration of a loop requires separate constraints—likewise with all branches of conditional statements. Inequality tests, when translated into constraints, are expensive. So is RAM.

The question that we ask and answer in this paper is: if back-end costs are here to stay and we are stuck translating computations to constraints, what can we do to mitigate costs? Any such technique should achieve:

• Conciseness. Compared to a naive translation of a computation Ψ, we want to produce a smaller set of constraints.
• Coupling. There should be a way for the prover to actually satisfy the alternate constraints, which is non-trivial, since they may not correspond to the individual program steps that the prover takes to execute Ψ.

These two requirements have been addressed, at least partially. The authors of almost all front-ends observe that translation from a high-level computation Ψ need not result in constraints that simulate execution [23, 24, 27, 28, 44, 53, 63, 65, 66, 70, 72, 76, 79, 92] (§2). Rather, if the constraints are satisfiable if the execution is valid. For example, consider a computation that invokes a quicksort subroutine. The naive approach is to compile quicksort into constraints. As an alternative [44, Appx. C], the prover can sort “outside the constraints”, with the constraints enforcing that (a) the output is a permutation of the input [18, 80], and (b) this permutation is sorted. The naive approach requires \( O(n \log n) \) inequality tests while the alternative requires only \( O(n) \) inequality tests (for adjacent elements in part (b)). As inequality tests dominate this computation, the improvement is substantial.

We call such a checker of required properties a widget. This generalizes “gadget” [53], which refers to constraints that have been written by hand; widgets can additionally be encoded in a higher-level language and then compiled to constraints. Widgets have been proposed for arithmetic and bitwise operations [70], multiprecision operations [44], storage [24, 62], concurrent access to state [72], cryptographic operations [13, 22, 24, 28, 52, 62, 68], recursive composition [22, 28, 46], and optimization problems [4].

Yet, to the extent that these works make arguments about the correctness of substituting a computation with a widget, none of them provides formal justification: it is entirely possible that there are wrong widgets out there! Note that any such bug destroys the soundness of the end-to-end application. Thus, we add a third requirement:

• Correctness. This is not about correctness of the translation to constraints, which is crucial and complementary, and has
been studied [33]. Our focus on correctness in this paper is on substitutions (of computations by a widget) that happen “upstream” of compilation-to-constraints.

There is work addressing this third requirement [76] but at the expense of the first two (§7).

This paper’s contribution is a framework, Distiller, that addresses all three of these desiderata (§4). Distiller takes the goals in reverse order: it starts with Correctness. For each class of computations, the user writes down a specification of the computation, and proves a formal relationship between the implementation and the specification. This justifies compiling the specification (rather than the implementation) to constraints. This relationship is ensured by representing both the implementation and the specification as transition systems and adapting ideas from the theory of refinement [29, 47, 48, 54, 85]. A refinement relates the externally observable behaviors of two transition systems, formalizing the notion of correct substitution. A proof of refinement then yields a blueprint for the prover to satisfy the abstract constraints (Coupling). For Conciseness, a consequence of Distiller and its generality is to expose new opportunities for constraint reduction: Distiller lets us take the idea of widgets to its literal logical extreme.

We apply Distiller to a series of examples, including binary search, convex hull, maximal strongly connected components, and minimum spanning tree (§5). We also implement and evaluate Distiller (§6). To demonstrate the importance of Correctness, we encode the refinement proofs for our examples and verify them using the program verifier Viper [60]. The verification unveiled bugs that would have compromised Correctness in initial versions of two widgets. For Coupling, we build on the Pequin toolchain [66] to simulate an end-to-end system implementing Distiller where the prover executes the implementation whose result is then checked against the widget. Finally, we also evaluate Conciseness. Distiller achieves reductions in constraint size ranging from small constant factors to asymptotic improvements for some problems, which for small problem instances already result in double-digit factors. Qualitatively, the more complex a computation, the more improvement Distiller generally yields. Computations with many memory accesses or searches of memory see particular benefit under Distiller.

To be clear, replacing an implementation by its specification does not guarantee more concise constraints. However, as we explain (§4–§5), we can often use Distiller to find, and establish the correctness of, an intermediate point between specification and implementation that does yield a substantial improvement.

Distiller is not perfect (§8). As a built system, its trusted computing base includes the encoding of computations in Viper as well as the Pequin toolchain; also, it stops short of fully automating Coupling. However, none of these restrictions are fundamental.

The bottom line is that Distiller has taken a crucial step in improving front-ends: it has exhibited the logically most general way to exploit non-determinism in arithmetic constraints, while doing so soundly, with performance improvements that range from good constants to orders of magnitude.

2 Background: applied probabilistic proofs

This section is intended to give just enough context for the rest of the paper. For a full, rigorous treatment of probabilistic proof implementations, see Thaler [75].

Back-end. In these setups, a back-end is a cryptographic or complexity-theoretic protocol between an untrusted prover \( P \) and a verifier \( V \) in which \( P \) convinces \( V \) that a given set of equations \( C \) has a solution.

In more detail, \( V \) and \( P \) (which are possibly probabilistic) agree on \( C \), as defined by a protocol, or defined by a user who invokes \( V \) and \( P \). The variables in \( C \) are elements in a finite field, typically \( \mathbb{F}_p \) (the integers mod \( p \)), where \( p \) is a large prime (128 bits or more). For many back-ends, \( C \) is required to be in RIACS format [15, 16, 37, 65, 71]. RIACS generalizes arithmetic circuits, which generalize Boolean circuits. We refer to such a set of equations as constraints.

\( V \) does not trust anything \( P \) says; \( P \) can follow an arbitrarily malicious strategy (though some protocols presume a computational bound on \( P \) and cryptographic hardness assumptions of one kind or another).

\( P \) wants to prove to \( V \) that \( P \) holds a solution, or satisfying assignment, \( z \) to \( C \)—but \( V \) does not want to receive \( z \), and \( P \) may wish to keep \( z \) hidden. Instead, \( P \) gives \( V \) a certificate, possibly revealed interactively, which \( V \) checks. The guarantees are:

- **Completeness**: If \( C \) is satisfiable, then a correct \( P \) makes \( V \) accept, always (regardless of random choices made by \( P \), \( V \), or by the user in an offline phase).

- **Soundness**: If \( C \) is not satisfiable, the probability that \( V \)’s checks pass is negligible (the probability is over random choices made by the verifier or by the user in an offline phase). Some applications require a more general property, Proof of Knowledge (PoK): if \( P \) does not have access to a satisfying \( z \) (even if \( C \) is satisfiable), then \( V \) accepts with negligible probability. Note that these properties hold regardless of \( P \)’s strategy.

- **Zero knowledge**: \( V \) gets no information about \( z \) other than what can be deduced from the fact that \( C \) can be satisfied.

Examples of recent back-ends are [20, 25, 26, 35, 36, 45, 46, 50, 55, 69, 82, 83, 88, 89]. These trade off different properties, including the nature of the cryptographic assumptions, noninteractivity, whether there is an offline phase, whether that phase has to be repeated each time the structure of \( C \) changes, and so on. However, in all of these works, the costs have a major dependence on the number of constraints, \(|C|\), and thus all of these works will benefit from improvements to front-ends.

Pipeline. Posit a user who cares about verifying the execution of some high-level computation \( \Psi \), on some input \( x \). \( P \) supplies \( y \) that is purportedly \( \Psi(x) \), and wants to convince some \( V \), which is trusted by the user, that \( y=\Psi(x) \). As a generalization, \( \Psi \) can be a relation, so the goal is to prove that \( y \in \Psi(x) \). Existing implementations have the following pipeline:

**Offline (one-time for \( \Psi \))**:

0. The user writes down the computation \( \Psi \).
1. The user compiles $\Psi$ to constraints, $C$, over variables $X, Y, Z$, where $X$ and $Y$ are vectors of variables that represent the inputs and outputs.
2. The user runs any setup procedure required by the back-end.

**Online (for each $x, y$):**

3. Given a specific input $x$, $\mathcal{P}$ identifies a satisfying assignment $z$ to $C_{x,y}$, perhaps by executing $\Psi$. Here, $C_{x,y}$ means $C$ with $X$ bound to $x$ ($\mathcal{P}$’s requested input) and $Y$ bound to $y$ (the purported output).
4. $\mathcal{P}$ convinces $\mathcal{V}$ that it has, or knows, a satisfying assignment to $C_{x,y}$.

One property that we need from a pipeline is *End-to-end Soundness*: if $y \neq \Psi(x)$, then $\mathcal{V}$ rejects with overwhelming probability. This property relies on the back-end’s properties and the correctness of the compilation from $\Psi$ (see below). Another essential property is *End-to-end Completeness*: if $y = \Psi(x)$, then a correct $\mathcal{P}$ makes $\mathcal{V}$ accept with probability 1. This property relies on the back-end’s properties and the mechanics of Step 3.

**Front-end.** The front-end is Steps 1 and 3. We detail these steps below, incurring some textual debts to Buffet [79]. We focus on a compilation approach that we call the “ASIC approach”. The front-end is Steps 1 and 3. We detail these steps

---

**Motivating example: merging sorted lists**

Consider a computation, merge, that takes as input multiple sorted lists with unique elements (unique across all lists) and outputs a sorted union of the elements. An example implementation of merge, which we denote $T_{\text{merge}}$, is in Figure 1. When translated, merge comprises a number of constraints proportional to $L \cdot (\sum A_k \cdot \text{len})$, because of the nested loops on lines 9 and 11.

Observe that merge is computing its result. But in the setup of probabilistic proofs, the goal is to provide a proof about some alleged, exogenously-computed output. Thus, the set of constraints could instead check that a specification is met. We are interested in how to perform such a substitution systematically, meaning that the requirements in Section 1 are met.

A natural starting point is to translate the weakest logical specification (WLS) of merge that still expresses functional correctness: intuitively, one expects that logically weaker specifications “enforce less” and thus should yield smaller constraints when translated. Informally, the WLS is: “merge($L, A_0, \ldots, A_{k-1}, B$) terminates and, upon termination, $B$ is monotonically increasing and holds just each element from $\{A_k\}$ exactly once.” Pseudocode to check this specification, which we denote $T_{\text{merge}}$, is depicted in Figure 2. Its complexity is $2 \cdot (\sum A_k \cdot \text{len})$, which is an asymptotic
would result from simply executing the original computation.

A refinement proof coupling technique is to capture this relationship formally in terms of the actual code executed by the prover (\(T_I\)). We must prove a relationship between the generated transitions and their specifications. From a semantic perspective, a transition system \(T\) defines a language \(L(T)\), which contains for each execution trace \(o\) of \(T\), a sequence of observations \(o(\sigma)\) made about how \(T\) interacts with its environment during the execution.

We relate transition systems in terms of their languages. This allows us to formally capture when the execution of one transition system behaves like the execution of another, from the perspective of an external observer.

### 4 Framework

We formalize our framework in terms of transition systems, which provide a uniform formalism for representing both implementations and their specifications. From a semantic perspective, a transition system \(T\) defines a language \(L(T)\), which contains for each execution trace \(o\) of \(T\), a sequence of observations \(o(\sigma)\) made about how \(T\) interacts with its environment during the execution. These observations may for instance encompass I/O, network traffic, etc.

Figure 2: Pseudocode for the weakest logical specification (\(T_S\)) of the merge computation. The precondition only requires that \(B\) has enough physical space for the elements of all \(A_k\).

```
void merge(L, A_0, ..., A_{L-1}, B) {
    \ell_0: int[L] curr = (0);
    int len, running_min, kstar; bool found;
    len = 0;
    \ell_1: for (int k = 0; k < L; k++) {
        len += A_k[len];
        \}
    \ell_2: for (int i = 0; i < len; i++) {
        found = false
        \ell_3: for (int k = 0; k < L; k++) {
            if (curr[k] < A_k.len && (found ||
                A_k[curr[k]] < running_min)) {
                running_min = A_k[curr[k]];
                // running_min is the current min element
                kstar = k;
                // kstar indexes the list that contains
                // running_min
                found = true;
                // indicates that branch has been taken
            }
        }
        B[i] = running_min;
        curr[kstar]++;
        \}
    \ell_4: return;
}
```

```
void merge_spec_naive(L, A_0, ..., A_{L-1}, B) {
    int k_i, j_i, i_kj;
    havoc B.len;
    for (int i = 0; i < B.len; i++) {
        havoc B[i];
        assume i == 0 || B[i-1] < B[i];
        havoc k_i;
        assume 0 <= k_i && k_i < L;
        havoc j_i;
        assume 0 <= j_i && j_i < A_k_i.len;
        assume B[i] == A_k_i[j_i];
    }
    for (int k = 0; k < L; k++) {
        for (int j = 0; j < A_k.len; j++) {
            havoc i_kj;
            // each element in some A_k is in B
            assume 0 <= i_kj && i_kj < B.len;
            assume A_k[j] == B[i_kj];
        }
    }
    return;
}
```

Figure 1: Pseudocode for the computation \(merge(L, A_0, ..., A_{L-1}, B)\) (\(T_I\)).

The precondition of merge requires that the \(A_k\) are strictly sorted and their elements pairwise distinct. Also, there must be enough physical space in \(B\) to store the elements of all \(A_k\).

Improvement over \(L \cdot (\sum_k A_k.len)\) from earlier.

To read the pseudocode, note that the keyword \texttt{havoc} denotes a nondeterministic choice, while \texttt{assume} constrains choices.

Concretely, when this pseudocode is compiled to \(L(T_I)\) (§2), \texttt{havoc} statements become free variables that the prover supplies while \texttt{assume} statements become constraints that enforce the given statement. The specification uses \texttt{for}, which (logically) means bounded universal quantification, and (mechanically) unrolls and repeats the enclosed requirements.

In Figure 2, lines 4–12 constrain \(B\) to be sorted (in increasing order), and enforce that \(B \subseteq \bigcup_k A_k\). In particular, for each position \(i\) in \(B\), the prover nondeterministically supplies which list \((k_i)\) contributes to the \(i\)th position, and which index in that list \((j_i)\) holds the contributed element. For the other direction, lines 13–20 specify that \(\bigcup_k A_k \subseteq B\).

But how does the prover supply these values? Ideally they would result from simply executing the original computation.

This brings us to the Correctness and Coupling requirements (§1). We must prove a relationship between \(T_S\) and the actual code executed by the prover (\(T_I\)). The basic technique is to capture this relationship formally in terms of refinement [47, 48, 54]. A refinement proof coupling \(T_I\) and \(T_S\) not only establishes the correctness of the substitution, it also tells us how to augment \(T_I\). The prover then executes the augmented implementation, which yields the values for the non-deterministically assigned variables in the specification.

A further improvement is possible. Notice that the implementation \(T_I\) (Fig. 1) uses the facts that the input lists are unique and sorted, whereas \(T_S\) (Fig. 2) uses neither fact. In the framework that we lay out in the sections ahead, we will have the freedom to choose a specification that refines the WLS yet still abstracts the computation. For example, by taking advantage of the uniqueness of the input lists, we obtain a less general but more concise specification than \(T_S\). Specifically, we discard the lines in Figure 2 (13–20) that enforce \(\bigcup_k A_k \subseteq B\), resulting in Figure 3, which we call \(T_E\). When translated, \(T_E\) now yields a number of constraints proportional to \(\sum_k A_k.len\), which saves a factor of two compared to \(T_S\).
4.1 Transition systems and refinement

In our formalization, we adapt the classical setup of Abadi and Lamport [1]. A transition system \( T = (\Sigma, \Theta, \Delta, O, \alpha) \) consists of a set of states \( \Sigma \), a nonempty set of initial states \( \Theta \subseteq \Sigma \), a set of transitions \( \Delta \subseteq \Sigma \times \Sigma \), a set of observations \( O \), and an observation function \( \alpha : \Sigma \rightarrow O \). Intuitively, the function \( \alpha \) formalizes which aspects of a given state are observable. When \( T \) is known, we denote a transition \( (s, s') \in \Delta \) by \( s \rightarrow s' \) and say \( s \) steps to \( s' \). We also call \( s' \) a successor of \( s \).

Example 4.1. We illustrate with our motivating example (§3). We can regard \( T_1 \) (Figure 1) as defining a transition system \((\Sigma, \Theta, \Delta, O, \alpha)\), as follows. The states \( \Sigma \) of \( T_1 \) are mappings from program variables to values. For \( s \in \Sigma \), we denote by \( s.x \) the value of program variable \( x \) in \( s \). We sometimes write \( x \) for a value of the program variable \( x \) when the state \( s \) is unspecified. We write \( s[x \mapsto v] \) to denote the new state obtained from \( s \) by updating the value of \( x \) to \( v \) and keeping the values of all other program variables unchanged. The program variables include a dedicated variable \( pc \) storing the value of the program counter, which ranges over the control locations \( \ell_0, \ldots, \ell_4 \). (For simplicity of exposition, we are treating the execution of a basic block, such as one iteration of a non-looped loop, as a single transition.)

The observations \( O \) of \( T_1 \) are the values of the input arrays and output array at the program start and return. Intuitively, these are the values that an external user can observe from the program. All intermediate program states of the computations are unobservable, which we denote by the special observation \( \tau \). Formally, we define \( O \) using the following grammar:

\[
O := \text{in}(L, A_0, \ldots, A_{L-1}, B) \mid \text{out}(L, A_0, \ldots, A_{L-1}, B) \mid \tau .
\]

The observation function \( \alpha : \Sigma \rightarrow O \) is then defined as follows:

\[
\alpha(s) = \begin{cases} 
\text{in}(s.L, s.A_0, \ldots, s.A_{(s\cdot \text{len})-1}, s.B) & \text{if } s.pc = \ell_0 \\
\text{out}(s.L, s.A_0, \ldots, s.A_{(s\cdot \text{len})-1}, s.B) & \text{if } s.pc = \ell_4 \\
\tau & \text{otherwise} \end{cases}.
\]

The transitions \( \Delta \) of \( T_1 \) are obtained from the program description in the expected way. For instance, the body of the for loop at control location \( \ell_1 \) yields all transitions \( s \rightarrow s' \) such that \( s.pc = \ell_1, s.k < s.L \), and

\[
s' = s[\text{len} \mapsto s.\text{len} + s.A_{(s.k)}\cdot \text{len}][k \mapsto s.k + 1] .
\]

The set of initial states \( \Theta \) consists of all states \( s \) that satisfy the precondition of \( T_1 \) (Figure 1). We assume that this precondition is specified by a formula \( \varphi_{\text{pre}} \). That is, \( \varphi_{\text{pre}} \) states that \( pc = \ell_0 \), and that the arrays \( A_k \) are sorted in strictly increasing order and its elements pairwise distinct. We write \( s \models \varphi_{\text{pre}} \) to indicate that \( s \) satisfies \( \varphi_{\text{pre}} \).

An infinite sequence of states \( \sigma \) is called an (execution) trace of \( T \) if it starts in an initial state and respects \( T \)'s transition relation: formally, \( \sigma_0 \in \Theta \) and for all \( i \geq 0 \), either \( \sigma_i \) steps to \( \sigma_{i+1} \) or \( \sigma_i = \sigma_{i+1} \) and \( \sigma_i \) has no successors in \( \Delta \). If \( \sigma_i = \sigma_{i+1} \), we say that \( \sigma \) stutters in step \( i \). A terminating execution of \( T \) corresponds to a trace that stutters forever in its final state. By abuse of notation, we write \( \alpha(\sigma) \) to denote the sequence of observations obtained by applying \( \alpha \) pointwise to the states in \( \sigma \). We denote the set of all traces of \( T \) by \( \text{traces}(T) \).

Let \( \beta \) be the function that maps a sequence \( \sigma \) to the sequence obtained from \( \sigma \) by replacing all repeated consecutive copies of elements by a single copy, for example, \( \beta((0, 0, 1, 1, 1, 2, 3, 3, 3, 3, 3)) = (0, 1, 2, 3) \).

The language of \( T \), denoted \( L(T) \), is defined by applying \( \alpha \) pointwise to each trace in \( \text{traces}(T) \) and then removing stutters. The intuition for removing stuttering is that we want to capture only the observable behavior: stuttering steps correspond to unobservable internal computation steps. Formally, we define the sequence of observations \( o(\sigma) \) made from a trace \( \sigma \) as \( o(\sigma) \overset{\text{def}}{=} \beta(\alpha(\sigma)) \) and then let

\[
L(T) \overset{\text{def}}{=} \{ o(\sigma) \mid \sigma \in \text{traces}(T) \} .
\]

Example 4.2. In the motivating example (§3), the language of the transition system \( T_1 \) is simply

\[
L(T_1) = \{ (\alpha(s), \tau, \alpha(s')) \mid s \models \varphi_{\text{pre}} \land s' \models \varphi_{\text{post}} \} .
\]

Here, the preconclusion \( \varphi_{\text{pre}} \) is as defined above. The postcondition \( \varphi_{\text{post}} \) states that \( pc = \ell_4 \), \( B \) is sorted in strictly increasing order, and the set of elements of \( B \) is equal to the union of the set of elements of the arrays \( A_k \). The single \( \tau \) in each observation sequence in \( L(T_1) \) summarizes all intermediate states of the computation.

A transition system \( T_r \) refines another transition system \( T_s \) iff \( L(T_r) \subseteq L(T_s) \). This definition captures the idea that from the perspective of an external observer, every execution of \( T_r \) behaves like some execution of \( T_s \). Typically, we think of \( T_s \) as the specification and \( T_r \) as the implementation. We denote a refinement relationship by \( T_r \preceq T_s \).

A classical approach to proving refinement relationships is to construct a refinement mapping. Formally, a refinement mapping between \( T_r \) and \( T_s \) is a function \( r : \Sigma_r \rightarrow \Sigma_s \) such that

1. \( r(\Theta_r) \subseteq \Theta_s \),
2. \( \forall s \in \Sigma_r \), \( \alpha_r(s) = \alpha_s(r(s)) \), and
3. \( \forall s, s' \in \Sigma_f, \text{if } s \rightarrow_f s', \text{then } r(s) \rightarrow_S r(s') \text{ or } r(s) = r(s'). \)

The first property states that \( r \) maps the initial states of \( T_I \) to those of \( T_S \). The second property states that the observations computed from states are preserved by \( r \). The third property states that every transition in \( \Delta_f \) is matched by a corresponding transition in \( \Delta_S \) under \( r \) or by a stuttering step. Together, these properties capture the intuition that the relationship between a refinement and its specification is that the specification abstracts steps that are “internal” to the implementation.

Once it has been established that \( r: \Sigma_f \rightarrow \Sigma_S \) is a refinement mapping, \( T_I \leq T_S \) follows: given a trace \( \sigma_I \) of \( T_I \), the sequence \( r(\sigma_I) \) is a trace of \( T_S \) (modulo stuttering). Moreover, \( r(\sigma_I) \) makes the same observations as \( \sigma_I \), i.e., \( \exists(\alpha_i(\sigma_I)) = \exists(\alpha_3(r(\sigma_I))). \)
Hence, the existence of \( r \) establishes that \( T_I \) refines \( T_S \).

We write \( T_I \leq S \) to indicate that \( r \) is a refinement mapping between \( T_I \) and \( T_S \). An important property that we will use freely later is that refinement mappings compose: \( T_1 \leq S, T_2 \leq S, T_3 \leq S \) implies \( T_1 \leq S, T_3 \).

### 4.2 Refinement-based widgets

We can now use the language of transition systems to recast weakest specification; the transition systems\( T \) implies a means to an end to solve the problem. L\( \) for the prover\( C \) is no systematic approach to compute a satisfying assignment for\( C \). The problem is for the prover\( P \) to convince the verifier\( V \) that \( C \) is satisfiable iff \( \sigma \) does not execute\( W \). An important property that we will use freely later is that refinement mappings compose: \( T_1 \leq S, T_2 \leq S, T_3 \leq S \) implies \( T_1 \leq S, T_3 \).

Constructing refinement mappings. It remains to show how to construct refinement mappings. We demonstrate this with the merge computation as a guiding example, using general principles inspired by refinement calculi such as [56, 58, 59]. These principles apply broadly (Section 5 contains further examples).

Our first step is to construct a refinement mapping \( r \) between the transition system \( T_I \) of the merge computation (Fig. 1) and its intermediate specification \( T_E \) (Fig. 3). As we will explain below, \( r \) can then be used to obtain a satisfying assignment for the constraints \( C(T_E) \) from a given execution of \( T_I \), enabling more efficient verification of that execution. In a second step, we then show that the intermediate specification \( T_E \) refines the naive specification \( T_S \).

To prove \( T_I \leq T_E \), we divide the construction of the refinement mapping into three steps by deriving two auxiliary transition systems \( T_{IE} \) and \( T_{IE} \) that yield a refinement sequence \( T_I \leq T_{IE} \leq T_{IE} \leq T_{IE} \leq T_{IE} \). Intuitively, the auxiliary transition systems couple \( T_I \) and \( T_E \) so that they are executed together.

A transition system \( T_I \) refined by an implementation \( T_I \) will typically involve nondeterministic (havoc, see §3) assignments to program variables that do not appear in the implementation. In our example of the merge computation, these are the assignments to \( k_i \) and \( j_i \) in Fig. 3 (lines 8 and 10). The execution of \( T_E \) can proceed only if the value chosen by each nondeterministic assignment satisfies the constraints imposed by the subsequent assume statements. A key step in the refinement proof is therefore to show that such values can be obtained from the trace \( \sigma_I \). We make this step explicit in the construction of the intermediate transition system \( T_{IE} \). This transition system augments \( T_I \) with those variables unique to \( T_E \) as well as assignments to these variables that determine the desired values to be chosen for the nondeterministic assignments in \( T_E \). In our example, this augmentation can be seen in lines 3, 21, and 23 shown in blue in Fig. 4.

Observe that the assignments to \( k_i \) and \( j_i \) in \( T_{IE} \) of our example depend only on the original program variables of \( T_I \). Moreover, the variables do not interfere with the other parts of the transition system in any way. Such auxiliary variables that
void merge (L,A₀,...,Aₙ₋₁,B) {
  ℓ₀: int[L] curr = {0};
  int len, running_min, kstar, k_i, j_i;
  bool found;
  len = 0;
  ℓ₁: for (int k = 0; k < L; k++) { len += Aₖ.len; }
  B.len = len;
  ℓ₂: for (int i = 0; i < B.len; i++) {
    B[i] = running_min;
    assume i == 0 || B[i-1] < B[i];
    k_i = kstar;
    assume 0 <= k_i && k_i < L;
    j_i = curr[kstar];
    assume 0 <= j_i && j_i < Aₖ.len &&
    B[i] == Aₖ[j_i];
    curr[kstar]++;
  }
  ℓ₄: return;
}

Figure 4: Pseudocode for the transition system \( \hat{T}_{IE} \). The prover will execute the black and blue code \( T_{IE} \) instead of \( T_I \). The values in blue are used to create an assignment to the nondeterministic variables occurring in the constraints obtained from \( T_E \) (the red assume statements).

are used only for the purpose of proving a refinement relation are sometimes referred to as ghost variables. A transition system \( T \) refines any transition system \( T' \) that is obtained from \( T \) by adding ghost variables [56]. What is more, the refinement mapping between \( T \) and \( T' \) can be constructed in a generic fashion. Thus, we obtain \( T_I \leq T_{IE} \).

In the context of program translation for probabilistic proofs, augmenting an implementation with ghost variables is not only useful for proving the refinement between \( T_I \) and \( T_{IE} \). The system \( T_{IE} \) also instructs the prover how to obtain the satisfying assignment for the constraints \( C(T_E) \). That is, the prover will actually execute \( T_{IE} \) instead of \( T_I \).

The next step in our construction is to augment \( T_{IE} \) with the assume statements in \( T_E \) that constrain the values chosen for the nondeterministic assignments. We call the resulting transition system \( \hat{T}_{IE} \). In our merge example, \( \hat{T}_{IE} \) is shown in Fig. 4 with the added assume statements highlighted in red (lines 8, 20, 22, and 24).

Establishing the refinement \( T_{IE} \leq \hat{T}_{IE} \) follows another generic construction. We first show that the added assume statements express invariants of \( T_{IE} \). That is, the assumed expressions must always evaluate to true in \( T_{IE} \), at the appropriate program points. In Appendix A, we discuss this step of the proof in more detail.
Given a non-empty array $A$ of length $n$, the problem is to find its smallest element, $\min$, and mark all occurrences of the minimum using another array $B$. More precisely, there must exist an index $p$ such that the following conditions hold:

1. $0 \leq p < n$ and $\min = A[p]$, 
2. for each $i \in [0, n]$, $\min \leq A[i]$, 
3. for each $i \in [0, n]$, $B[i] = (A[i] = \min ? 1 : 0)$. 

$T_S$ encodes this specification by nondeterministically choosing $\min$, each $B[i]$, and $p$. It uses two loops that iterate over $A$ to enforce conditions 2 and 3.

$T_I$ is shown in Fig. 6 (without the code in red, which we will discuss later). It also requires two loops: $\ell_0$ to compute $\min$, and $\ell_1$ to compute the $B[i]$. Comparing $T_I$ and $T_S$, we note that the two loops in $T_I$ and $T_S$ have exactly the same costs. However, $T_S$ performs an additional dynamic LOAD, namely $A[p]$, to enforce Condition 1. Hence, $C(T_S)$ incurs the extra cost of RAM initialization, which performs $n$ STOREs to write $A$ into the memory, and is therefore larger than $C(T_I)$.

However, we can do better than either $T_S$ or $T_I$. First, observe that unlike in $T_I$, we can merge the two loops in $T_S$ for conditions 2 and 3 into a single loop because $\min$ can be chosen nondeterministically upfront. Compared to $T_I$, this saves one of the two inequality tests $i < n$ that $C(T_I)$ would otherwise include for each iteration of the two loops. Furthermore, we can eliminate the LOAD $A[p]$ in Condition 1 of $T_S$ by introducing an auxiliary variable $\hat{p}$ that indicates whether $\min$ has been encountered at least once in the loop that checks conditions 2 and 3. The pseudocode of the resulting $T_E$ is shown in Fig. 7 (excluding the blue code, which we will use later to establish that $T_E$ refines $T_S$).

Thus, $C(T_E)$ needs only $2 \cdot n$ inequality tests, saving $1/3$ over $C(T_I)$. Since the encoding of inequality tests dominates the size of the generated constraints, we observe a similar constant factor improvement in the overall constraint size.

Turning to the refinement proofs, if we add the red code in Fig. 6 to $T_I$, we obtain the augmented transition system $\hat{T}_{IE}$ for showing $T_I \leq T_E$ (see §4.2). Recall that the main part of the refinement proof is to show that the added assume statements in $\hat{T}_{IE}$ coming from $T_E$ always succeed. We focus on the assume on Line 18, which is the most interesting one. Observe that the loop at $\ell_0$ ensures $0 \leq p < n \land A[p] = \min$ after the loop has terminated. Using this fact, we can then establish the loop invariant $i < p \lor \hat{p}$ found for the second loop at $\ell_1$. This then allows us to prove that the assume statement on Line 18 is safe.

Next consider the refinement $T_E \leq T_S$. Adding the blue code in Fig. 7 to $T_E$ yields an augmented transition system $\hat{T}_{ES}$ for the refinement proof $T_E \leq T_S$. We focus on showing that $T_E$ ensures Condition 1. (The other two conditions follow immediately from the loop in $T_E$.) To this end, we can establish the loop invariant found $= 0 \lor (0 \leq p < n \land A[p] = \min)$ for the loop at $\ell_1$. Together
1. If Line 14, this implies that adding the assume on Line 15 is safe. This line then establishes Condition 1.

We note that we would not be able to improve over \( T_I \) if the array \( A \) was guaranteed to have a single minimum, or if we were satisfied with finding any of the minimums in \( A \). The loops at \( \ell_1 \) and \( \ell'_1 \) would be unnecessary.

5.2 Binary Search

Given a sorted array \( A \), the bounds \( l, r \) of a possibly empty segment in \( A \), and a value \( x \), the problem is to compute \( i \) such that \( l \leq i \leq r \) and \( A[i] = x \). If no such \( i \) exists, return \( i = -1 \).

\( T_S \) for this problem checks \( i \) according to the specification above. That is, if \( i \neq -1 \), \( T_S \) checks that \( l \leq i < r \) and \( x = A[i] \); otherwise it iterates over \( A[l ... r] \) and checks that the segment does not contain \( x \). \( T_I \) is based on standard binary search.

For our cost analysis we focus on the number of \texttt{LOAD} operations, which is the largest contributor to the size of the generated constraints. In the worst case, \( T_I \) performs \( \log(n) \) \texttt{LOAD} operations to search through the segment \( A[l ... r] \) where \( n = r - l \). In contrast, \( T_S \) performs \( n + 1 \) \texttt{LOAD} operations in the worst case. That is, \( T_S \) is asymptotically worse than \( T_I \).

We can do better by exploiting that \( A \) is sorted. Introducing an auxiliary value \( s \), we divide the specification for the case when \( x \) is not present (\( i = -1 \)) into four subcases:

1. If \( i = -1 \), then \( l = r \) or \( x < A[l] \) or \( x > A[r - 1] \) or \( l < s < r - 1 \land A[s] \leq x < A[s + 1] \).
2. else \( l \leq i < r \land A[l] = x \).

\( T_E \) is the direct encoding of this case analysis. It performs a constant number of \texttt{LOAD} operations, achieving an asymptotic improvement over \( T_I \). We note that if the search is viewed as a standalone program, then this improvement is overshadowed by the cost of storing the array segment into RAM, which is linear in \( n \). However, if the search is executed many times or viewed as a subroutine, then the RAM initialization can be amortized.

For proving \( T_E \leq T_S \), observe that each of the subcases of Condition 1 implies that \( x \) cannot be present anywhere in the segment.

For the last three cases, the proof relies on the precondition that \( A \) is sorted in strictly increasing order. For proving \( T_I \leq T_E \), recall that binary search iteratively shrinks a subsegment \( A[l \ldots r] \) of \( A[l \ldots r'] \) that may still contain \( x \). This process continues until the subsegment converges to a single point \( l' = r' \), which is the index of the least element larger than \( x \). In the nontrivial case where \( l \neq r \) and \( x \) is not present in the segment but within the range of values defined by \( A[l] \) and \( A[r - 1] \), we define \( s = l' - 1 \) for the final point \( l' = r' \). Then \( s \) is the index that satisfies the last disjoint in Condition 1.

5.3 2D Convex Hull

Given a set of points \( P = \{ p_0, \ldots, p_{n-1} \} \subseteq \mathbb{Z}^2 \) with \( n > 1 \), assume no three points are on the same line, the problem is to find all points in \( P \) that lie on the convex hull of the set.

We additionally require \( P \) to satisfy the precondition of Graham Search [41], a popular algorithm that solves the 2D Convex Hull problem. Specifically, \( p_0 \) has the smallest \( y \) coordinate among all points in \( P \), and the greatest \( x \) coordinate among all points in \( p \) with the same \( y \) coordinate as \( p_0 \). The remaining points are sorted in counterclockwise order when using \( p_0 \) as a reference point. In other words, for each \( i \in (0, n) \), let \( L_i \) be the line passing through \( p_0 \) and \( p_i \). Then intersect \( L_i \) with a horizontal line at \( p_0 \) and define \( \angle_i \) to be the top-right angle of the intersection. \( P \) is ordered so that \( \angle_i < \angle_{i+1} \) for all \( i \).

With these assumptions, \( C \) defines the convex hull of \( P \) iff the following conditions hold:

1. \( C \subseteq P \).
2. \( p_0 \in C \) and for all \( i \in (0, n) \), \( p_i \in C \) or the angle defined by the points \( \langle \text{prv}_i, p_i, \text{nxt}_i \rangle \) bends inwards, where \( \text{prv}_i \) and \( \text{nxt}_i \) are the first points before and after \( p_i \) in \( P \) that are also in \( C \). If no such \( \text{nxt}_i \) exists, then \( \text{nxt}_i = p_0 \).

Condition 1 ensures that \( C \) contains no points outside \( P \). Since \( P \) is sorted, Condition 2 guarantees that \( C \) contains all the points of \( P \) that lie on the convex hull of \( P \).

\( T_S \) nondeterministically chooses \( C \) and then checks the above conditions. The size of \( C(T_S) \) is \( O(n^2) \). (In particular, for each \( p_i \), \( T_S \) needs to iterate through \( P \) again to find \( \text{prv}_i \) and \( \text{nxt}_i \).)

We use Graham Search as the \( T_I \) for this problem. For each of the \( n \) points, \( T_I \) needs two \texttt{STORE} operations and two dynamic \texttt{LOAD} operations.

\( T_E \) is shown in Fig. 8. It nondeterministically chooses \( k, C \), and the points \( \text{nxt}_i \), then it iterates over the \( p_i \) and checks all relevant conditions in constant time for each \( i \). The refinement proof showing \( T_I \leq T_E \) uses the fact that \( T_I \) computes the points in \( C \) in the order in which they appear in \( P \). Moreover, the \( \text{nxt}_i \) can be computed by \( T_{IE} \) using a simple linear scan of the final \( C \).

To see that \( T_E \) yields smaller constraints than \( T_I \), observe that only the array access of \( C[\text{count}] \) on Line 15 is dynamic and incurs the cost of two \texttt{LOAD}s (one for each coordinate of the point). Also, there are no \texttt{STORE} operations. (Recall that a \texttt{havoc} command stands for augmented code in \( T_{IE} \).) Hence, it does not contribute to \( C(T_E) \). So \( T_E \) only performs two dynamic \texttt{LOAD} operations per iteration. The cost of a \texttt{STORE} depends on how deeply it is nested in conditionals whereas the cost of a \texttt{LOAD} does
int X_PROD(p, q, r) {
    return (q.x-p.x) * (r.y-p.y) - (q.y-p.y) * (r.x-p.x)
}  

void 2d_convex_hull_efficient(n, P, C) {
    int k;
    havoc k;
    point nxt, prv;
    havoc nxt; // nxt0
    prv = P[0]; // prv1
    havoc C[0]; assume C[0] == prv;
    int count = 1;
    for (int i = 1; i < n; i++) {
        point cur = P[i];
        if (nxt == cur) { // P[i] in C
            havoc C[count]; assume C[count] == cur;
            // get nxt, because nxt_{i-1} ≠ nxt,
            havoc nxt; // nxt_{i-1}
            // angle (prv, cur, nxt) must bend inwards
            assume X_PROD(prv, cur, nxt) > 0;
            prv = cur; // prv_{i+1}
            count++;
        } else { // P[i] !in C
            // nxt_{i-1} = nxt_{i-1}; prv_{i+1} = prv_{i}
            assume X_PROD(prv, cur, nxt) < 0;
            }
        }
    }
    assume nxt == P[0];
    assume k == count;
}  

Figure 8: Pseudocode of \( T_E \) for 2D Convex Hull.

not \cite[§3.1]{79}. Specifically, each STORE in \( T_I \) is four times more expensive than a LOAD in \( T_E \). We therefore expect that the size of \( C(T_E) \) is about five times smaller than that of \( C(T_I) \).

If we consider the variant where the problem is not to enumerate \( C \) but to compute its characteristic function on the indices of \( P \) (that is, mark the points in \( P \) that belong to \( C \)), then we can eliminate all dynamic LOAD operations from \( T_E \) and achieve an asymptotic \( \log(n) \) factor improvement over \( T_I \).

5.4 Maximal Strong Components

Given a directed graph \( G = (V, E) \) with nodes \( V \) and edges \( E \subseteq V \times V \), the problem is to partition \( V \) into the maximal strongly connected components \( C_0, \ldots, C_{k-1} \) of \( G \). We represent the \( C_i \) implicitly using an array \( \text{rank} \) that maps every node \( v \in V \) to the index of its maximal strongly connected component. That is, we define for all \( i \in [0, k), C_i = \{ v \in V \mid \text{rank}[v] = i \} \). Given this, the formal problem statement is to find \( k \) and \( \text{rank} \) such that the following three conditions hold:

1. For all \( v \in V \), \( 0 \leq \text{rank}[v] < k \).
2. For all \( i \in [0, k) \), there exists a cycle \( c_i \) in \( G \) that visits exactly the nodes in \( C_i \).
3. For all \( i \in [0, k) \) and all cycles \( c \) in \( G \), if \( c \) visits some node in \( C_i \) then \( c \) visits only nodes in \( C_i \).

\( T_S \) encodes the above specification by nondeterministically choosing \( k, \text{rank}[v] \) for each node \( v \in V \), and the cycles \( c_i \) for each component \( i \in [0, k) \). Condition 3 quantifies over the set of all cycles in \( G \), which is in general an infinite set. However, it can be shown that restricting the quantification to simple cycles in \( G \) yields an equivalent condition. A simple cycle is a path where only the first and last node are equal and all other nodes are distinct. The condition that quantifies over simple cycles can be encoded using a nested loop that iterates over all partial permutations \( p \) of nodes in \( G \) and then checks that if \( p \) forms a simple cycle in \( G \) and intersects with a \( C_i \), then it is fully contained in \( C_i \). As the number of partial permutations grows exponentially with \( |V| \), so does \( |C(T_S)| \).

We use Dijkstra’s MSC algorithm \cite[Chapter 25]{30} as our \( T_I \). The algorithm is similar to Tarjan’s algorithm \cite{74} but lends itself more directly to an efficient translation into constraints. The algorithm iterates over \( E \) and \( V \). In each iteration, it performs up to 13 LOAD and 8 STORE operations. These operations dominate the size of the generated constraints.

However, we can again construct a \( T_E \) that improves over both \( T_I \) and \( T_S \). The key idea for \( T_E \) comes from Dijkstra’s correctness argument for his algorithm. Dijkstra observed that a set of connected components \( C_0, \ldots, C_{k-1} \) is maximal iff it can be ordered so that all edges leaving a \( C_i \) target only nodes in components preceding \( C_i \). Given Dijkstra’s observation, we can replace Condition 3 in \( T_S \) with the following condition in \( T_E \):

3*. For all \((v, w) \in E \), \( \text{rank}[w] \leq \text{rank}[v] \).

Replacing Condition 3 by 3* yields a refinement of \( T_S \).

Additionally, Condition 2 can be reformulated as a Condition 2* that no longer relies on the construction of explicit cycles \( c_i \) connecting the nodes in each component. We observe that the nodes in each \( C_i \) can be arranged in a tree that implicitly witnesses the existence of an appropriate \( c_i \) (which we use in the \( T_E \leq T_S \) proof). The tree reflects the way \( T_I \) traverses the nodes in \( V \) and collapses candidate components whenever a node is revisited. These trees can be obtained from \( T_I \) using an augmentation that does not increase \( T_I \)’s asymptotic complexity. We use this augmentation to establish Condition 2* when proving that \( T_I \) refines \( T_E \).

We defer the details of how Condition 2* is expressed to Appendix B. What is important is that the combined size of these trees is linear in \( |V| \) and so is checking their correctness. As a result, for dense graphs \((|E| \approx |V|^2)\), the cost to enforce Condition 2* is insignificant. A detailed cost analysis yields an expected reduction in total constraint size for dense graphs by a factor of 17.5 for sufficiently large \( |E| \). For shallow graphs \((|E| \approx |V|)\), we still obtain a reduction by a factor of two. The principal savings come from the fact that conditions 3* and 2* can be checked by \( T_E \) (in the sense of validated inside an \assumecomma) with many fewer LOAD and STORE operations versus \( T_I \).

5.5 Minimum Spanning Tree

Given a connected graph \( G = (V, E) \) with \( |V| \) vertices and \( |E| \) undirected edges with unique positive weights, its minimum spanning tree (MST) is the minimum weight subgraph, \( M \), of \( G \) which contains all of the vertices in \( G \). The MST problem is to find \( M \).
given \(G\). By definition, because our graph is connected, \(M\) is a tree with \(|V| - 1\) edges. There are several equivalent alternative definitions of an MST. The one that we are concerned with is that for every cycle in \(G\), the heaviest edge in that cycle is not in \(M\). \(T_3\) encodes this specification by nondeterministically choosing \(|V| - 1\) edges to be \(M\) and \(|E| - |V| + 1\) cycles attesting to the non-membership of every edge in \(E\) not in \(M\). As each cycle contains at most \(|V|\) edges, the size of the constraints generated from \(T_3\) grows \(O(|V| \cdot |E| \cdot \log(|V| \cdot |E|))\).

Our \(T_I\) is Kruskal’s amortized MST algorithm, a procedure that sorts the edges by weight and then greedily builds \(M\) one edge at a time, only adding an edge to \(M\) when the edge does not form a cycle. The key to Kruskal’s efficiency is the use of a disjoint set data structure \(S\) to efficiently detect cycles. In \(S\), vertices are arranged in a forest of inverted trees with individual sets identified by the root vertices of these trees. \(S\) supports 3 operations: INSERT, FIND-SET, and UNION. INSERT adds an element to \(S\) in its own set in \(O(1)\) memory operations. FIND-SET traverses up the tree to the root to get the unique set identifier for a queried element in amortized \(O(\alpha(|V|))\) and worst case \(O(\log(|V|))\) memory operations (where \(\alpha\) is the Inverse Ackermann function). UNION takes two sets and combines them into one in \(O(1)\) memory operations.

When deciding whether or not to add an edge \(e\) to \(M\), the algorithm calls FIND-SET at the endpoints of \(e\) to efficiently test if that edge would form a cycle. If the edge does not form a cycle, it is added to \(M\), and \(S\) is updated by calling UNION on the sets for its respective endpoints. The amortization in FIND-SET comes from the careful implementation of FIND-SET with path compression (updating all pointers along the path to the root to be direct to the root) and UNION by rank (to minimize the height of the combined trees). When implementing FIND-SET we can either use the amortized version which results in \(O(|E| \cdot \alpha(|V|) \log(|E| \cdot \alpha(|V|)))\) constraints with a large overhead or the worst-case version which results in \(O(|E| \cdot \log |V|) \cdot \log(|E| \cdot \log |V|))\) constraints but a much smaller constant. However, by performing a refinement on the underlying memory model, we can get the near-optimal asymptotic behavior of the amortized version with the small constant of the worst-case version, outperforming both \(T_I\) implementations for all input sizes.

The amortized \(T_I\) needs to use multiple nested loops and conditionals to maintain asymptotic complexity. The large quantity of branches created by these loops and conditionals generate a massive constant overhead. We can avoid this overhead in our nested find set operations by replacing RAM with a new DS-SET memory construct. We combine the LOADs and STOREs into a single UPDATE operation, and we define a FIND-SET operation that completely eliminates the nested loops. In our model, there is a fixed buffer of nondeterministically supplied UPDATE operations to handle the amortized complexity of FIND-SET as every FIND-SET call is a series of UPDATE operations which result in a fixed-point (the root).

As a further reduction, we observe that the main loop, which iterates through the sorted edges, has a conditional that is true exactly \(|V| - 1\) times; however, when we unroll the loop to constraints we incur its cost for every iteration. Since there is a direct correspondence between the current edge weight, the current iteration of the while loop, and the program counter, we can replace time order in our memory transcript with edge weight. This eliminates the need to sort the edges and allows us to reorder iterations freely. We then split the main loop into two smaller loops without branching. The first loop asserts that the condition is true and executes the contents of the conditional block unconditionally; the second simply asserts that the condition is false. Combined with other small improvements, these RAM refinements completely eliminate all branching and state machine overhead (4 while loops and a nested if statement) and results in a set of constraints that is several orders of magnitude smaller for small inputs and converges to a 52.2 \times constant improvement for large inputs.

6 Experimental evaluation

This section answers the following questions:

(1) Does Distiller increase confidence in the correctness of widgets?

(1) How difficult is it to build an end-to-end system for probabilistic proof checking based on Distiller?

(1) Can we empirically achieve a constraint size reduction when using Distiller with existing front-ends?

Mechanizing refinement proofs. To answer Question ((1)) we use the deductive verification tool Viper [60] to mechanize the \(T_I \leq T_E\) and \(T_E \leq T_S\) proofs for 10 of our 11 benchmark problems (§5). For MST (§5.5), we perform the refinement proofs only on paper. In this case, \(T_S\) also encompasses the specification of a refined RAM, making the proof mechanization more elaborate.

For the proofs showing \(T_I \leq T_E\), we encode the transition system \(\tilde{T}_{IE}\) into Viper’s programming language, replacing all assume statements coming from \(T_E\) by assert statements. Then we annotate all loops with candidate invariants that are intended to imply the safety of the asserts (§4.2). The tool automatically checks that the annotated invariants are inductive and indeed imply the safety of the asserts. We proceed similarly for the proofs of \(T_E \leq T_S\).

Viper verifies all programs. However, for two of the benchmarks we discovered bugs in the initial version of \(T_E\). These bugs would have compromised End-to-End Soundness (§2). One bug was a missing array bounds check in \(T_E\) of the merge computation (§3). The other one was a subtle omission of a check in \(T_E\) for the Sum of Powers problem [30, Chapter 19]. We discovered these bugs when trying to annotate the respective \(T_E\) to prove \(T_E \leq T_S\).

End-to-end system. As outlined in §4.2, an end-to-end system would take as input \(\tilde{T}_{IE}\) and split it into \(T_E\) and \(T_{IE}\). Then it would compile \(T_E\) to constraints \(C(T_E)\), run \(T_{IE}\), and feed the values into \(C(T_E)\). We simulate these three steps (split, compile, and feed) using existing mechanisms of the Pequin toolchain [66]. Pequin’s front-end accepts programs written in a subset of C, extended with domain-specific constructs. In particular, Pequin allows the input C program to be annotated with assertions. Each assertion
translates to R1CS constraints checking that the assertion holds. We use this construct to compile the `assume` statements in $T_E$. Another construct is `exo_compute`, a hook allowing the prover to execute another program that produces values for arbitrary nondeterministic inputs to the generated constraints. This feature enables the prover to run $T_{IE}$ and supply the auxiliary inputs to $C(T_E)$ when solving the constraints. While the coupling of $T_E$ and $T_{IE}$ via `exo_compute` is done manually, the process is mechanical and only takes a few minutes per example, suggesting that it can be automated.

We perform the split, compile, and feed steps for a select subset of our benchmarks. To test End-to-end Completeness of the approach, we apply the Pequin toolchain to the encoded benchmarks and successfully run it on a range of inputs. We note that the overhead of executing $T_{IE}$ versus $T_I$ is negligible compared to the rest of the pipeline (recall that Step 3 contributes negligibly to costs in the first place; §2). To test End-to-end Soundness, we also run Pequin with buggy versions of the $T_{IE}$. In these cases, the back-end correctly rejects the computation.

**Constraint size reduction.** Recall that our primary performance metric is $|C|$ (§2). Our final experiment assesses Distiller against this metric. For all of our benchmarks, we express $T_I$, $T_E$, and $T_S$ as input programs for Pequin. Then, with the exception of MST, we run Pequin’s front-end on all three programs for a range of values for the loop unrolling bound that determines the maximal input size for each benchmark problem. As MST relies on a refined RAM construct that is not available in Pequin, we calculate the size of the constraints generated by all RAM operations by hand and run Pequin on the rest of the program. We then combine the result of the two parts to obtain the final constraint size. We enforce a timeout threshold of 240s per run, with the exception of MSC, where a 2000s threshold is chosen to enable computations on larger problem instance sizes that demonstrate the exponential behavior of $T_S$. For each successful run, we measure the size of the generated R1CS constraints and compute the relative sizes of $C(T_S)$ and $C(T_E)$ compared to $C(T_I)$.

Figure 9 shows the results obtained for those benchmarks where our theoretical analysis yields an improvement of $T_E$ over
We note that for the MSC problem (§5.4), the relative improvement between $T_E$ and $T_I$ increases with the problem instance size. The maximal MSC instance size for which the translation of $T_I$ does not time out is $n = 20, m = 400$. This is still too small to observe the $17.5 \times$ theoretical improvement that we predict for dense graphs. Conversely, for the MST problem $T_I$ has a large constant overhead that causes the improvement achieved with $T_E$ to be $3 \times$ larger on small instances than the predicted $25.5 \times$ improvement for large problem instances. Finally, for binary search (BinS) we observe that the cost of storing the input array to be

$$T_E - T_I = O \left( \frac{n^3}{2} \right)$$

agrees with the expected $T_E - T_I = O \left( \frac{n^3}{2} \right)$, which is linear in the size of the array, dwarfs the $\log(n)$ improvement achieved for a single invocation of the binary search (§5.2).

Figure 10 shows the results obtained for the three problems where our theoretical analysis predicts that $T_E$ performs asymptotically better than $T_I$ with increasing problem instance size. The experiment again confirms our predictions. In particular, for the merge problem discussed in §3, Fig. 10a shows that $|\mathcal{C}(T_E)|/|\mathcal{C}(T_I)|$ is approximately hyperbolic, which we expect because the predicted improvement for each point is $L \times$, where $L$ is the number of merged arrays. Also, if we discount the RAM initialization cost for binary search, then we see the expected $\log(n)$ factor improvement (Fig. 10b).

7 Other related work

Probabilistic proofs. Section 2 gave an overview of probabilistic proof implementations, covering back-ends and front-ends; see also [75, 78, 81]. Unlike Distiller, none of the front-ends achieves all three requirements stated in the Introduction; in fact, none creates a framework for proving the correctness of widgets.

Distiller combines formal methods and probabilistic proofs. Very few works live at this intersection. Some notable exceptions are as follows. CirC [63] is a toolkit for building compilers to a family of constraint formalisms, including R1CS and SMT instances. Its architecture takes advantage of the rich SMT toolbox, allowing users to build powerful analyses and optimizations. The two works are complementary: one could compile a Distiller-verified widget in CirC, to get further reductions.

The Orbis Specification Language (OSL) [76] has a similar ideology to ours: replace a computation with its formal specification, and compile the latter to constraints, in the hope of gaining more concise constraints. However, as our examples (§5) make clear, the cost of a naïve specification is often exorbitant. So one has to identify an “in-between” specification, and (a) relate it to the abstract specification, and (b) derive an implementation that knows how to satisfy the in-between specification or the original. Neither of these problems is addressed by OSL. The authors mention that they want to synthesize implementations from specifications. Though, for the rich specification language we consider (general transition systems), whether a specification even has an implementation is undecidable. Thus, to instantiate the ideology that OSL and we share, one needs human input (to write down $T_E$, and relate it to the specification and the implementation).

In an under-appreciated work, Fournet et al. [33] develop a compiler, based on CompCert [51], that formally connects the semantics of a higher-level program to the constraint formalism (specifically R1CS constraints). This work is complementary to Distiller—it provides the compiler correctness condition described in Section 2.

Leo [27] also has the goal of formally verified translations to constraints. Leo develops a compiler and uses the ACL2 [43] theorem prover to validate each stage of translation. However, this falls short of a verified compiler, as in Fournet et al. Moreover, the authors of Leo want to validate hand-crafted gadgets. It is not clear how to do this, since ACL2 cannot easily “reverse” R1CS instances to lift them to higher-level semantics. As a consequence, crucial pieces of verification are works in progress [27, §6.4]. By contrast, Distiller incorporates widgets soundly and completely, by treating them at the source code level and using refinement.

Refinement. The idea of developing a program from a specification in a step-wise refinement process goes back to early work by Dijkstra [29, 30] and Wirth [85]. The formal concept of refinement relations and mappings to relate the observable behaviors of transition systems was first explored in the 1980s [47, 48, 54]. It is a cornerstone of modern Formal Methods; applications include reasoning about concurrent and distributed systems, establishing program equivalence, and verifying security properties.

Abadi and Lamport [1] showed that refinement mappings yield a complete proof technique for establishing refinement. Though, in general, the technique requires the transition systems to be augmented with history variables (recording information about past states) and prophecy variables (predicting information about future states). Other related proof techniques for establishing refinement, for instance, based on (weak) simulation relations [57, 64, 77] are less suited for our purposes as they do not immediately provide a blueprint for computing satisfying assignments.

The notion of refinement considered in Distiller takes a monolithic view of transition systems, which makes it difficult to reason compositionally about subroutines. Contextual refinement [32] relates the observable behavior of subroutines subject to all possible client programs that may use them, thereby enabling compositional reasoning. For the relatively simple programming models supported by most existing probabilistic proof front-ends (no concurrency, object-oriented features, or higher-order functions), considering contextual refinement instead of global refinement does not add substantial complexity to the verification effort.

Distiller uses mechanized proofs (§6). Specifically, it uses a lightweight encoding of refinement proofs in the language of the deductive program verifier Viper [60]. The proofs are partially automated using SMT solvers. However, this is a choice. Nothing in our approach precludes or necessitates particular approaches to mechanization. In particular, there is a large body of work on refinement calculi that mechanize the correct step-wise refinement of programs and system models [2, 8, 56, 58, 59]. More recently, the many applications of proofs concerning products and couplings of two or more programs have fueled the development of relational program logics that provide frameworks for proof mechanization [9–11, 19, 34, 73, 87]. Several of these formal reasoning systems have been implemented in tools, including for instance TLA+ [49], Rodin [3], EasyCrypt [12], and ReLoC [34].


8 Discussion and conclusion

Distiller improves a key metric in implementations of probabilistic proofs, namely the number of arithmetic constraints required to encode the validity of the execution of a computation. The improvements typically range from small integer factors to orders of magnitude, depending on the computation.

Nevertheless, Distiller has limitations. First, we use Viper not on the actual code that the Pequin toolchain works over but rather on a representation of that code in Viper; this leaves a potential safety gap. Similarly, although we are encouraged by certified compiler work in this research area [33], we ourselves use Pequin, which is not verified; this is not fundamental but does mean that the built system has another potential gap. Speaking of our built system, it stops short of being automated end-to-end; coupling involves some manual code shuffling (§6). However, this step is mechanical, and we plan to automate it.

Despite these limitations, Distiller has taken a crucial step forward. It introduces, for the first time, a framework for widgets that are correct by construction. This framework radically expands the space of potential widgets, thereby allowing probabilistic proofs to do much more, by paying much less.

Acknowledgments

We thank Sebastian Angel, Jacob Salzberg, Justin Thaler, and Riad Wahby for helpful conversations. This research was supported by DARPA under Agreement No. HR00112020022, NSF under grant CNS-1514422, and AFOSR under grant FA9550-18-1-0421. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the United States Government, DARPA, NSF, or AFOSR.

A Details of refinement mapping: merging

This section provides details on showing that the assume statements of $\hat{T}_{IE}$ (Figure 4) of the merge computation express invariants of $T_{IE}$. This guarantees that the assume statements will always evaluate to true in $T_I$ (Figure 1), and completes the refinement $T_{IE} \leq \hat{T}_{IE}$. See Section 4.2 for the full discussion.

We justify the addition of the assume statements by providing supporting invariants at relevant portions of $T_{IE}$. Figure 11 shows $\hat{T}_{IE}$ with the desired invariants explicitly added. There are four assume statements added to $T_{IE}$ that require justification, on Lines 33, 35, 37, and 38. We omit the proof of the assume statement on Line 11 as it is easy to show.

The first assume (Line 33) states that the most recently added element $B[i]$ is properly ordered with respect to its preceding element. To prove this, we establish the invariants on Lines 21 to 23 at the inner loop, and the invariant on Line 15 at the outer loop. These invariants imply that, at each iteration of the outer loop, $B[L]$ is set to a value greater than $B[i-1]$, provided $\text{found}$ is true after the inner loop terminates. To show the latter, first observe that $k = L$ holds after the inner loop terminates. Now, if

found were false at this point, then Line 20 would imply

$$\sum_{n=0}^{L-1} \text{curr}[n] \geq \sum_{n=0}^{L-1} A_n \cdot \text{len}. $$

However, the invariants on Line 14 together with the loop condition $i < B \cdot \text{len}$ of the outer loop and the equation on Line 11 imply

$$\sum_{n=0}^{L-1} \text{curr}[n] < B \cdot \text{len} = \sum_{n=0}^{L-1} A_n \cdot \text{len}. $$

This yields a contradiction. Hence, $\text{found}$ must be true after the inner loop terminates.

To prove the second assume (Line 35), we can again use the invariant at Line 21 and the fact that $\text{found}$ must hold after termination of the inner loop.

Similarly, the third assume (Line 37) follows from the invariants at Lines 21 and 22 at the inner loop, and 13 at the outer loop.

The final assume (Line 38) follows from the identity

$$\text{running} = A_{k\star}[\text{curr}[k\star]] $$

This is sufficient because $k\star$ is set to $k$, $i$ is set to $\text{curr}[k\star]$, and $B[i]$ is set to $\text{running} \cdot \text{min}$. The remaining invariants that are not used directly to prove the assume statements are needed to ensure that the invariants are inductive. In particular, Line 24 at the inner loop is needed to show that Line 15 is maintained by each iteration of the outer loop.

The invariants, thus, justify the addition of the assume statements in $\hat{T}_{IE}$, proving the refinement $T_{IE} \leq \hat{T}_{IE}$.

B Details of MSC Example

In this section, we expand on the discussion of the maximum strongly connected component problem in Section 5.4.

Given a directed graph $G = (V, E)$ with nodes $V$ and edges $E \subseteq V \times V$, the problem is to partition $V$ into the maximal strongly connected components $C_0, \ldots, C_{k-1}$ of $G$. We represent the $C_i$ implicitly using an array rank that maps every node $v \in V$ to the index of its maximal strongly connected component. That is, we define for all $i \in [0, k)$

$$ C_i = \{ v \in V \mid \text{rank}[v] = i \} . $$

Given this, the formal problem statement is to find $k$ and rank such that the following three conditions hold:

1. For all $v \in V$, $0 \leq \text{rank}[v] < k$.
2. For all $i \in [0, k)$, there exists a cycle $c_i$ in $G$ that visits exactly the nodes in $C_i$.
3. For all $i \in [0, k)$ and all cycles $c$ in $G$, if $c$ visits some node in $C_i$ then $c$ visits only nodes in $C_i$.

Weakest specification. $T_3$ encodes the above specification by nondeterministically choosing $k$, $\text{rank}[v]$ for each node $v \in V$, and the cycles $c_i$ for each component $i \in [0, k)$. Condition 3, which enforces that the components $C_i$ are maximal, is encoded by a
void merge (L,A[0],..,A[L-1],B) {
  ℓ₀: int[L] curr = (0);
  ℓ₁: int len, running_min, kstar, found, k_i, j_i;
  len = 0;
  ℓ₁: for (int k = 0; k < L; k++)
    { len += A[k].len; }
  B.len = len;
  ℓ₂: for (int i = 0; i < B.len; i++)
    { found = false;
      ℓ₃: for (int k = 0; k < L; k++)
        { found ⇒ ∀ j :: 0 ≤ j < k ⇒ curr[j] ≥ A[j].len
          found ⇒ ∑ₖ=0ⁿ⁻¹ curr[n] ≥ ∑ₖ=0ⁿ⁻¹ A[n].len
          found ⇒ 0 ≤ kstar < L
          found ⇒ curr[kstar] < A[kstar].len
          found ⇒ running_min == A[kstar][curr[kstar]]
          found ⇒ ∀ j :: 0 ≤ j < kstar ∧ curr[j] < A[j].len ⇒ running_min < A[j][curr[j]]
        }
      found = true;
    }
  B[i] = running_min;
  assume i == 0 || B[i-1] < B[i];
  ℓ₄: return;
}

Figure 11: Pseudocode for the transition system \( \hat{T}_{IE} \) with invariants.
nested loop, where the inner loop enumerates all cycles in \( G \) that visit exactly one node twice. As there are \( O(|V|^2) \) possible cycles that need to be considered, the size of the constraints generated from \( T_3 \) grows exponentially in \( |V| \).

**Implementation** We use Dijkstra’s MSC algorithm [30, Chapter 25] as our \( T_j \). The algorithm is similar to Tarjan’s algorithm [74] but lends itself more directly to an efficient translation into constraints.

Dijkstra observed that a set of strongly connected components \( C_0, \ldots, C_{k-1} \) is maximal if it can be ordered so that all edges leaving a \( C_i \) target only nodes in components preceding \( C_i \). That is, consider the graph \( G_C = (V_C, E_C) \) such that \( V_C = \{C_0, \ldots, C_{k-1}\} \) and \( (C_i, C_j) \in E_C \) iff \( i \neq j \) and \( (v, w) \in E \) for some \( v \in C_i \) and \( w \in C_j \). Then the above ordering of the \( C_i \) is a (reverse) topological sort of \( G_C \). The existence of this topological sort implies that \( G_C \) is acyclic and therefore the \( C_i \) cannot be joined into larger components. Given Dijkstra’s observation, one can replace Condition 3 in \( T_3 \) with the following:

\[
3^* \quad \text{For all } (v, w) \in E, \text{ rank}[w] \leq \text{rank}[v].
\]

The algorithm computes a rank that satisfies this revised condition. For pedagogy we discuss the abstract version of the algorithm shown in Figure 12, which assumes mathematical sets as an built-in type. Dijkstra’s concrete version of the algorithm implements the relevant set operations using auxiliary variables to achieve a running time that is linear in the size of \( G \).

The code highlighted in blue can be ignored for now. The algorithm iterates over \( V \) and \( E \) using two working sets \( VW \) and \( EW \) to keep track of the nodes and edges that still need to be processed. The stack \( cc \) is used to keep track of a chain of disjoint strongly connected components that are currently being traversed.

The algorithm maintains the invariant that each component is connected via an edge in \( G \) to its next component higher up on the stack. Moreover, the algorithm maintains that the set \( VC \) is the union of all the components in \( cc \). A node leaves \( VC \) (and, hence, \( cc \)) once its maximal strongly connected component has been identified and its rank has been assigned. The variable \( k \) keeps track of the number of maximal components that have been identified so far.

Whenever the stack \( cc \) becomes empty, the outermost loop nondeterministically chooses an unprocessed node \( v \) from \( VW \) and adds it as a new singleton component to \( cc \) (lines 14–17). While the stack is nonempty, the algorithm pops the topmost component \( C \) from \( cc \). Then it attempts to nondeterministically choose an unprocessed edge \((u, w)\) in \( EW \) such that its source node \( u \) is in \( C \) (lines 21–22). If no such edge exists, then \( C \) must be maximal and its nodes are removed from \( VC \) (lines 40–43). Otherwise, if the target node \( w \) of the chosen edge is in \( VC \setminus C \), then by the invariant of \( cc \) (that every component in \( VC \) is pointing to the next component higher up on the stack), there exists a cycle connecting all the components in \( cc \) starting with the component that contains \( w \), all the way to \( C \). The loop on lines 26–32 thus compacts these components to a new component \( C \). If on the other hand the target node \( w \) is an unprocessed node, then the current topmost component \( C \) is added back to the stack and then \( C \) is updated to the new topmost component consisting of \( w \) (lines 33–38). In all other cases, \( w \) is either already in \( C \) or is already in an identified

```plaintext
Figure 12: Pseudocode for implementation \( T_j \) of MSC. The code in blue is the augmentation needed for proving \( T_j \leq T_E \).
```

```
1 datatype comp =
2 Node(node)
3 | Rotate(node, comp, node)
4 | Concat(comp, comp)
5
6 int msc(V, E, rank) {
7 set<edge> EW = E;
8 set<node> VW = V;
9 set<node> VC = ∅
10 stack<set<node> * comp * node * node> cc = empty;
11 comp[] cycle;
12 int k = 0;
13 while (VW != ∅) {
14    int v = choose(VW);
15    VC = {v};
16    VW = VW \ {v};
17    push(cc, {v}, Node(v), v);
18    do {
19        set<node> C, comp c, node i, _ = pop(cc);
20        while ((\{(u, w) \in EW | u \in C \} != ∅)} {
21            edge (u,w) = choose((\{(u, w) \in EW | u \in C \});
22            EW = EW \ {\{(u,w)\};
23            if (w \in VC \ C) {
24                // compact the chain
25                c = Rotate(i, c, u);
26                do {
27                    set<node> C1, comp c1, node i1, node o1 =
28                        pop(cc);
29                    C = C \ C1;
30                    i = w \in C1 ? w : i1;
31                    c = Concat(Rotate(i, c1, o1), c);
32                    } while (w \in C)
33                    } else if (w \in VW) {
34                        VW = VW \ {w};
35                        VC = VC \ {w};
36                        push(cc, {\{(i,c1,u)\});
37                        C = {w}; c = Node(w); i = w;
38                    }
39                }
40            for (w \in C) rank[w] = k;
41            cycle[k] = c;
42            k = k + 1;
43            VC = VC \ C;
44        } while (VC != ∅);
45    }
46    assert isMSC(V, E, k, rank, cycle);
47    return k;
48 }
```


by compacting a cyclic chain in a new cycle each time the algorithm computes a new component, it follows that Condition 3 is assigned to its rank. This means that the component consists only of the node c, and all edges starting in c have rank larger than the rank of all previously assigned edges. Therefore, the augmentation additionally maintains for each component C, the target node i in C for the incoming edge from its predecessor in C, and the source node o in C for the outgoing edge to its successor in C (unless C is the topmost component, in which case o can be chosen arbitrarily).

Instead of representing each cycle c explicitly as a path in G, we represent it symbolically using an auxiliary tree-like datatype comp. To motivate this symbolic representation, consider the construction of an explicit cycle for a new compacted component obtained by the loop on lines 26–32. To this end, we need to construct a path c from w to u that visits all nodes in the new component. Since there exists an edge back from u to w, we obtain the desired cycle. The loop must therefore maintain the invariant that c is a path that visits all nodes in the components that have already been compacted into C. Moreover, this path must end in u and start in i. The latter is needed so that we can concatenate this path with another path that visits all nodes in the component C1 that precedes C in CC and that will be compacted next. The component C1 has an associated cycle c1 and is connected to its neighbors via i1 and o1. First, we rotate c1 to obtain a path that starts and ends in the node i1, thus visiting all nodes in C1 at least once. Next, we extend this path with the segment of c that goes from i1 to o1. Finally, we concatenate the resulting path with c using the edge (o1, i1) that must exist by the invariant of c. The final path goes from i1 to u and visits all nodes in C1 ∪ C (at which point i1 becomes the new i for C1 ∪ C), reestablishing the loop invariant.

The issue with this construction is that the size of the obtained cycle c can be exponential in the size of the compacted components. This blow-up would affect both the running time of the augmented Tij as well as the size of C(Tij). By representing the cycles using the type comp, we avoid this blow-up. Intuitively, a comp value c is a program that provides instructions for constructing a cycle that visits all the nodes appearing in c. The size of this program is linear in the number of visited nodes. Moreover, checking whether a given comp program c indeed constructs a cycle that contains all the nodes appearing in c can be done in linear time. This checker is our missing ingredient for Tij to guarantee Condition 2.

The comp programs have a tree-like structure where each vertex p is labeled with an instruction to construct a path from the paths computed by p’s children. There are three kinds of instructions, corresponding to the three different operations involved in the construction of cycles during compaction. The instruction Node(n) for a node n constructs a (trivially cyclic) path that consists only of the node n. The instruction Rotate(l, c, r) constructs a new path from l to r that visits all nodes in c, provided c constructs a cycle that contains l and r. Finally, Concat(c1, c2) concatenates the two paths constructed by c1 and c2, provided there exists an edge in G from the end point of c1 to the start point of c2.

The implementation of isMSC is shown in Figure 14. It uses the function isCycle to check for each i ∈ [0, k] whether cycle[i] constructs a cycle in G that visits all nodes appearing in cycle[i]. Moreover, it ensures that the rank of each of these nodes is indeed i. The auxiliary array seen is used to ensure that each node in V occurs in some cycle[i].
bool isMSC(V, E, k, rank, cycle) {
    bool b = true;
    bool[V] seen = {false};
    for (i ∈ [0,k))
        b = b & isCycle(V, E, rank, cycle[i], seen, i);
    for (v ∈ V) b = b & seen[v];
    for ((u,v) ∈ E) b = b & rank[v] <= rank[u];
    return b;
}

bool isCycle(V, E, i, rank, c, seen) {
    match c {
        case Node(n) =>
            seen[n] = true;
            return n ∈ V & rank[n] == i
        case Rotate(l, c1, r) =>
            return !seen[l] & !seen[r]
                & isCycle(V, E, i, rank, c1, seen)
                & seen[l] & seen[r]
        case Concat(c1, c2) =>
            return isPath(V, E, i, rank, c1, seen)
                & isPath(V, E, i, rank, c2, seen)
                & (right(c1),left(c2)) ∈ E
                & (right(c2),left(c1)) ∈ E
    }
}

bool isPath(V, E, i, rank, c, seen) {
    match c {
        case Node(n) =>
            seen[n] = true;
            return n ∈ V & rank[n] == i
        case Rotate(l, c1, r) =>
            return !seen[l] & !seen[r]
                & isCycle(V, E, i, rank, c1, seen)
                & seen[l] & seen[r]
        case Concat(c1, c2) =>
            return isPath(V, E, i, rank, c1, seen)
                & isPath(V, E, i, rank, c2, seen)
                & (right(c1),left(c2)) ∈ E
                & (right(c2),left(c1)) ∈ E
    }
}

node left(c) {
    match c {
        case Node(n) => return n
        case Rotate(l, _, _) => return l
        case Concat(c1, _) => return left(c1)
    }
}

node right(c) {
    match c {
        case Node(n) => return n
        case Rotate(_, l, r) => return r
        case Concat(_, c1) => return right(c1)
    }
}

We briefly discuss the $T_E \leq T_S$ refinement proof. The proof boils down to establishing conditions 1, 2, and 3 as an invariant after the assume at Line 7 of Figure 13. First, note that if $k$ is negative, then the loop on Line 5 of Figure 14 is skipped. Hence, the next loop on Line 7 will set $b$ to false because at least one node will not be seen, assuming $V$ is nonempty. Therefore, the checker implies Condition 1. Next, to prove that isCycle(V, E, rank, cycle[i], seen, i) implies Condition 2 for component $i$, we construct an interpreter for comp programs. Given a comp program c that satisfies isCycle, the interpreter executes c to construct the cyclic path in G encoded by c. Finally, the loop on Line 8 directly checks Condition 3*. As we have argued earlier, this implies Condition 3.

The combined size of all comp programs is linear in $|V|$ and so is checking their correctness (lines 5 and 6 in Figure 14). As a result, for dense graphs ($|E| \approx |V|^2$), the cost to enforce Condition 2 is insignificant. While checking Condition 3* is still linear in $|E|$, it involves only two LOADs per edge and no STORE. A detailed cost analysis yields an expected reduction in total constraint size for dense graphs by a factor of 17.5 for sufficiently large $|E|$. For shallow graphs ($|E| \approx |V|$), we still obtain a reduction by a factor of two.

C Details of MST Memory

Here we delve into the details of our technique in Section 5.5 as well as giving background on Pequin’s implementation of Buffet’s [79] RAM technique.

The complexity of our amortized $T_I$ is dominated by $261 \cdot |E| \cdot \alpha(|V|)$ disjoint set related RAM operations while our $T_E$ is dominated by $2 \cdot |E| \cdot \alpha(|V|)$ specialized disjoint set memory operations, each of which is 2.5× as expensive as a Buffet RAM operation.

In Buffet’s RAM model, operations are 4-tuples (time, op, val, addr), referring to the time, operation, value, and address respectively. The consistency of a transcript of RAM operations is checked by sorting these tuples by addr with ties decided by pc, and then checking the pairwise consistency of LOAD and STORE operations at the same address. Sorting tuples is done via a permutation network that contains $O(s \cdot n \log n)$ constraints in general where $n$ is the number of inputs and $s$ is the size of the tuple.

For our refined approach, we combine the LOADs and STOREs into a single UPDATE operation, and we define a FIND–SET operation that completely eliminates the nested while loops while maintaining amortized complexity. These operations are 5-tuples, (time, op, old, new, parent, node), and the transcript is divided into external and internal data-structure operations. The external operations are those called by the main algorithm and are generated when the program is unrolled during constraint compilation. To achieve amortized complexity with minimal overhead, we introduce a fixed size buffer of fully non-deterministic internal UPDATE operations which ensure the consistency of the external FIND–SET operations. Every FIND–SET call is certified by a series of UPDATE operations which result in a fixed-point where old parent, new parent, and node are the same (the root).

Figure 14: Pseudocode for checker used by $T_E$ of MSC.
The ordering of these operations on the initial transcript does not matter because the transcript is sorted before any consistency checks are performed. To check memory consistency, we follow the techniques for standard RAM, and we also run a second sort over the transcript by time to check the fixed-point property of \textsc{Find-Set}. Our 5-tuples are sorted first by time, with ties decided by placing \textsc{Find-Set} first and the requisite number of \textsc{Update} operations in their proper order. The overhead of doing a second reordering of the transcript (2 ×) combined with the overhead of handling 5-tuples instead of 4-tuples (1.25 ×) results in 2.5 × more expensive memory operations.

We make one additional major change to memory by replacing the time with weight since there is a direct correspondence between the two. Then, instead of sorting the edges by weight, we sort them into two groups, those in the MST and those not contained in the MST. This alternative ordering allows us to split the main loop into two loops, and we can remove the nested conditional from one of the loops. Instead of the expensive operations in the conditional block being unrolled into constraints |E| times, they only appear |V| − 1 times (from the unrolling of the first loop). Calling any memory operations out of time order in the circuit (in both Buffet’s RAM and our disjoint set memory model) does not matter because they are later reordered for consistency checks.

Replacing time with weight and reordering iterations allows us to remove the overhead of nested conditionals in loops, and introducing a buffer for internal operations allows for cheaper amortized memory operations. Together these improvements result in a substantial 52.2 × reduction in the cost of memory operations on large input sizes with additional smaller order improvements, particularly when |E| ≫ |V|.

References


19


