Precise Reasoning with Structured Time, Structured Heaps, and Collective Operations

GRÉGORY ESSERTEL, GUANNAN WEI, and TIARK ROMPF, Purdue University, USA

Despite decades of progress, static analysis tools still have great difficulty dealing with programs that combine arithmetic, loops, dynamic memory allocation, and linked data structures. In this paper we draw attention to two fundamental reasons for this difficulty: First, typical underlying program abstractions are low-level and inherently *scalar*, characterizing compound entities like data structures or results computed through iteration only indirectly. Second, to ensure termination, analyses typically project away the dimension of time, and merge information per program point, which incurs a loss in precision.

As a remedy, we propose to make collective operations first-class in program analysis—inspired by Σ-notation in mathematics, and also by the success of high-level intermediate languages based on `map`/`reduce` operations in program generators and aggressive optimizing compilers for domain-specific languages (DSLs). We further propose a novel structured heap abstraction that preserves a symbolic dimension of time, reflecting the program's loop structure and thus unambiguously correlating multiple temporal points in the dynamic execution with a single point in the program text.

This paper presents a formal model, based on a high-level intermediate analysis language, a practical realization in a prototype tool that analyzes C code, and an experimental evaluation that demonstrates competitive results on a series of benchmarks. Remarkably, our implementation achieves these results in a fully semantics-preserving strongest-postcondition model, which is a worst-case for analysis/verification. The underlying ideas, however, are not tied to this model and would equally apply in other settings, e.g., demand-driven invariant inference in a weakest-precondition model. Given its semantics-preserving nature, our implementation is not limited to analysis for verification, but can also check program equivalence, and translate legacy C code to high-performance DSLs.

CCS Concepts: • **Software and its engineering → Automated static analysis**; *Software verification*; *General programming languages*; *Semantics*.

Additional Key Words and Phrases: static analysis, verification, program transformation, semantics

ACM Reference Format:
Grégory Essertel, Guannan Wei, and Tiark Rompf. 2019. Precise Reasoning with Structured Time, Structured Heaps, and Collective Operations. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 157 (October 2019), 30 pages. https://doi.org/10.1145/3360583

1 INTRODUCTION

Programs like the one in Figure 1 are a real challenge for analysis and verification tools. The language features used include arithmetic, dynamic memory allocations, linked heap structures, and loops. Analysis tools need to reason about all these features with high precision. If precision is lost at any point, it may be impossible to obtain any useful result. In practice, many state-of-the-art tools are unable to verify this program, including tools that score highly on software verification competitions such as CPAchecker [Beyer and Keremoglu 2011] and SeaHorn [Gurfinkel et al. 2015],

Authors' address: Grégory Essertel; Guannan Wei; Tiark Rompf, Purdue University, USA, gesserte@purdue.edu, guannanwei@purdue.edu, tiark@purdue.edu.

2475-1421/2019/10-ART157
https://doi.org/10.1145/3360583

```
// build list of numbers < n
x := null; l := 0
while l < n do {
  y := new;
  y.head := l;
  y.tail := x;
  x := y;
  l := l + 1
}
```

```
// traverse list, compute sum
z: = x; s: = 0
while z != null do {
  s := s + z.head;
  z := z.tail
}

// check result (closed form)
assert(s == n*(n-1)/2)
```

Fig. 1. Challenge program: in constrast to state-of-the-art tools like CPAchecker, SeaHorn, or Infer, our approach is able to verify the final assertion, as well as the absence of memory errors such as dereferences of `null` or missing fields. Dynamic allocations and linked data structures pose particular difficulties w.r.t. disambiguation of memory references, manifest in potential aliasing and the problem of "strong" updates. Our structured heap model represents allocations inside a loop using a collective form for sequence construction $y = \langle . \rangle(i < n).\ [\texttt{head} \mapsto i, \texttt{tail} \mapsto \ldots]$, based on which the second loop maps to a collective sum $s = \Sigma(i < n).\ y[i].\texttt{head} = \Sigma(i < n).\ i$ over this sequence. Knowledge about closed forms for certain sums validates the final assert (details, including the definition of $\texttt{tail} \mapsto \ldots$, are shown in Figure 2 and Section 2.2).

as well as Facebook's Infer tool [Calcagno et al. 2015]. While it is easy to come up with a long list of individual reasons that make this kind of analysis hard, we make two overarching observations:

(1) Program abstractions used in common analysis methods are typically *scalar*, i.e., they represent individual program variables, relations between individual variables as in the case of relational abstract domains, array updates at individual positions, and so on. But program abstractions do not typically represent *collective* entities such as "an array that contains the natural numbers from 1 to n" or "the sum of all elements in an array." Instead, such information must be encoded extensionally using quantified and often recursive formulae.
(2) Program abstractions typically project away the dimension of *time*. Most analyses gather and collapse information into a single abstract value per *program point* (possibly with some context-sensitivity). This means that in the presence of loops, values computed in different loop iterations are not distinguished. Hence, program abstractions do not typically represent *space-time* information such as "field `tail` of the object allocated here in a given loop iteration points to the object allocated at the same position in the preceding loop iteration."

In this paper, we address both points through first-class collective operations and a structured heap representation, coupled with a structured notion of time.

First-Class Collective Operations. For the first point, we propose to model *collective operations* such as sums or array formation as first-class entities, without quantifiers or recursive definitions. This idea is inspired by Σ-notation in mathematics, and by recent advances in highly optimizing compilers where high-level IRs based on `map`, `reduce`, and similar collective abstractions have had significant success.

In mathematics, collective forms such as $\langle a_i \rangle_i$ for sequences and $\Sigma_i a_i$ for series are not just compact syntactic sugar, but they give rise to intuitive algebraic laws. Thus, collective forms enable reasoning about sequences and series on a higher level than directly about the underlying recurrences. Introduced by Fourier [1820], big-Σ and related operators have rapidly become an integral part of modern mathematics, and they have found their way into programming languages in the form of comprehensions via SETL [Schwartz 1970], via Dijkstra's Eindhoven Quantifier Notation [Dijkstra 1976], and of course as functional operators via APL [Iverson 1980].

If these abstractions help manual reasoning, then it seems only logical that they should also help automated reasoning—so it is surprising that program analysis tools do not in general afford

collective forms first-class status and do not try to reverse-engineer low-level code into such higher-level representations. Instead, automated tools usually reason at the level of scalar recurrences, which poses all kinds of challenges.

Compilers: From Optimizing for Performance to Simplifying for Clarity. In this aspect, the field of optimizing compilers is ahead of general program analysis. Compiler writers have long recognized that aggressive transformations such as automatic parallelization are very hard to perform on low-level, imperative program representations. Hence, there has been ample work on trying to extract structure from low-level code. For example, the Chains of Recurrences (CoR) model [Bachmann et al. 1994; Engelen et al. 2004] is a collective and closed-form representation for classes of functions including affine, multivariate polynomial, and geometric functions, which is widely used in optimizing compilers for generating efficient code to compute a given function on an interval of indexes, e.g., in a loop.

Over the last decade, a thriving line of research has demonstrated that even more aggressive transformations such as automatic parallelization are imminently practical for domain-specific languages (DSLs) that restrict mutability and make collective operations such as `map`, `reduce`, `filter`, `groupBy`, etc., first-class, so that the DSL compiler can reason about them algebraically when making optimization decisions, potentially coupled with auto-tuning and/or search for the best implementation based on cost models and dynamic programming. These systems outperform comparable code written in general-purpose languages by orders of magnitude and achieve asymptotically better parallel scaling [Brown et al. 2016, 2011; Ragan-Kelley et al. 2013; Rompf et al. 2013, 2011; Steuwer et al. 2015, 2017; Sujeeth et al. 2014, 2011]. And while the orignal goal was for programmers to write DSL code directly, recent research has also shown that it is often practical to "decompile" low-level legacy code into high-level DSLs, whose role shifts to that of an intermediate representation [Ahmad and Cheung 2016; Kamil et al. 2016; Mendis et al. 2015; Radoi et al. 2014; Raychev et al. 2015; Rompf and Brown 2017; Rompf et al. 2014].

The key thrust of this paper is to take this approach further and apply it to more general program analysis settings, including for the purpose of automatic verification. Thus we shift the goal from optimizing programs for performance to simplifying programs for clarity, by extracting high-level collective operators from low-level code. It is not intuitively clear that this reverse-engineering task is easier than the desired analysis itself, but we will show how several techniques come together to make this approach practical, in particular by fusing several simplification and analysis tasks into a single iterative fixed-point computation (Section 4).

Structured Time and Structured Heaps. A major challenge remains: dynamic memory allocation, coupled with unbounded iteration constructs, may lead to an unbounded number of runtime objects, which need to be mapped to a finite static representation. The crux of this challenge is to find a static representation that still enables effective disambiguation of memory references in order to minimize potential aliasing of pointers, which, among other detrimental effects, stands in the way of *strong updates*: recognizing when the previous value of a variable or memory location is definitely overwritten. More generally, strong updates are one example of a situation where an analysis needs to reason about the dimension of *time*, i.e., relating values at different points during the dynamic execution of the program. To address this challenge, a key ingredient of our approach is to identify useful collective-form representations for arbitrary-size dynamic heap structures such as arrays and linked lists.

We propose a novel structured heap model that incorporates the dimension of time in the structure of addresses and the allocation policy (see Figure 2). This concrete heap model leads directly to an abstract heap model that permits concise and expressive symbolic representation. Instead of abstracting dynamic allocations uniformly per program point, and having the abstract

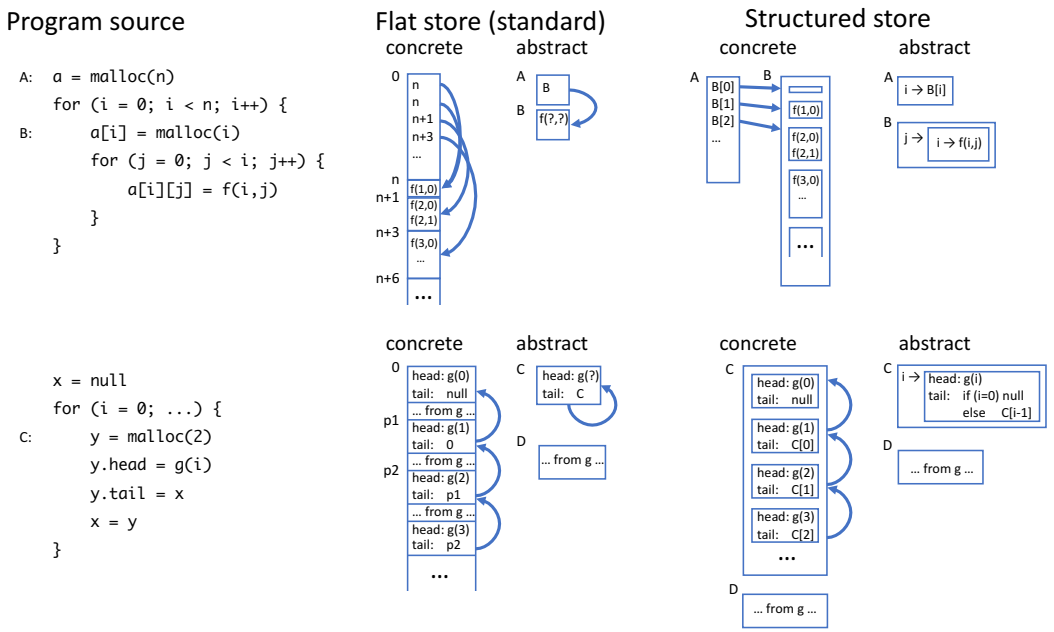

Fig. 2. Flat vs. structured stores for two example programs. The standard flat store model assigns consecutive numeric addresses for each allocation. By contrast, addresses in our structured store model consist of a program point and the surrounding loop variables at the time of allocation. `A, B[1], C[42]` are all valid concrete addresses. For program points inside loops (B and C), all objects allocated there are are represented as sequences ($B = \langle \ldots \rangle_i$ and $C = \langle \ldots \rangle_i$) in the store. Going from concrete to abstract stores, these sequences can be abstracted as collective forms, with greatly improved precision over an abstract store that only distinguishes allocations by program point. In particular, it is straightforward to capture the property that elements in a linked list point to the element allocated in the previous loop iteration and the last element points to null.

heap map abstract locations to sets of abstract values, we represent objects allocated at a program point inside a loop as a potentially unbounded sequence, indexed by the loop variable. This enables us to reason about all objects allocated at a given program point in a collective way, and assign a concise abstract summary based on a *symbolic* loop index. This in turn gives us fine-grained abilities to reason about objects allocated in *different* loop iterations and about their interactions, including strong updates deep within data structures (see Figure 2).

Framework Instantiation. We instantiate our framework in a way that is similar to deductive verification with forward reasoning, i.e., a strongest postcondition model. We translate imperative source programs into a functional representation and simplify, preserving correctness and error behavior modulo termination. The translation makes all error conditions explicit, so verification amounts to checking that the `valid` tag that signals the occurrence of errors in the final program state has been simplified to the constant `true`.

In the simplest case, simplification can happen entirely after translation, based on various rewriting strategies that apply simplification rules one by one. The strategies can be deterministic, e.g., apply simplification rules bottom-up, or non-deterministically search for the most profitable simplification based on various heuristics (Section 3.3).

From Pessimistic to Optimistic. However, even search-based post-hoc simplification strategies are fundamentally limited in that each individual rewrite has to be equality-preserving. The result is essentially a phase ordering problem (typical in compilers). The solution is to *interleave* translation

and simplification [Lerner et al. 2002], which enables a form of *speculative* rewriting where chains of rewrites can be tried and either committed, if they are found to preserve equality, or rolled back, if not (Section 4). The post-hoc approach is also characterized as *pessimistic*, and the interleaved approach as *optimistic*. To illustrate the difference, the optimistic approach can simplify a loop based on the assumption that a variable remains constant throughout the loop, if it verifies the assumption afterwards. By contrast, the pessimistic approach would need prove that the variable is loop-invariant first, which may not be possible without the simplifications currently on hold.

In both pessimistic and optimistic approaches, the end result is a precise symbolic representation of the program state post execution. Although inefficient, this representation could be used to compute the concrete program output for any given input. This means that our approach leads to more precise information than strictly necessary for verification, and essentially solves a harder problem than demand-driven verification approaches. The fact that we are able to gain this much precision in a strongest-postcondition setting that is typically considered an unworkable verification approach makes us confident that the core of our method will apply just as well in weakest precondition scenarios or analyses based on abstract interpretation that approximate deliberately, and lead to increased precision there as well.

We implement our approach in a prototype system called `SIGMA`, which analyzes C code. Since our approach produces precise symbolic program representations, we can use `SIGMA` not only for verification, but also for checking program equivalence, and for translating legacy code to high-performance DSLs, as we demonstrate in our evaluation.

Contributions. To the best of our knowledge, no previous work models first-class collective operations in a general-purpose program analysis setting. There are specialized uses in aggressive optimizing compilers that aim to retarget legacy code to high-performance DSLs [Ahmad and Cheung 2016; Kamil et al. 2016; Mendis et al. 2015; Radoi et al. 2014; Raychev et al. 2015; Rompf and Brown 2017; Rompf et al. 2014], but these systems (a) are specific to a given target DSL, and (b) only deal with flat arrays, not linked lists or dynamic memory allocation. Likewise, simple classes of structured heap models have been used in previous work [Dillig et al. 2011b; Tan et al. 2017], but these are restricted to separating container instances. The idea of indexing program values by loop iterations has also been proposed in the context of dynamic program analysis [Xin et al. 2008] and in polyhedral compilation [Benabderrahmane et al. 2010]. Our key novel insight, not found in any previous work, is to push execution indexing all the way into the heap and allocation model. We give the heap a structure that uniformly reflects the program's loop structure, with objects allocated at a program point inside a loop represented as a sequence indexed by the loop variable, and use this concrete heap model as a basis for symbolic analysis. We make the following specific contributions:

- We describe the basics and intuition behind our approach of deriving collective forms through a series of examples with increasing complexity (Section 2).
- We present a detailed formal semantics of our source and target languages, including the structured heap model. We prove correctness of the translation and target-level simplification rules. The simplification rules we present give rise to a large space of rewriting opportunities that can be realized either deterministically bottom-up, or nondeterministically through search. Each rule is guaranteed to be equality-preserving, which leads to a simple but overall *pessimistic* approach if rules are applied one-by-one after the translation step (Section 3).
- We extend the pessimistic, equality-preserving, simplification model to an *optimistic* approach that interleaves translation and simplification, based on Kleene-iteration. This model further increases precision by enabling a form of speculation, e.g., assuming that parts of a data structure

remain constant throughout a loop, that an arithmetic recurrence has a closed form, or that a write to a data structure is the initialization of a dense array (Section 4).
- We present SIGMA, which scales up the ideas to analyze C programs, and discuss its implementation (Section 5).
- We evaluate SIGMA on benchmarks for verification, program equivalence checking, and translation of legacy code to high-performance DSLs (Section 6).

Section 7 discusses related work and Section 8 concludes. Our mechanized Coq model and prototype tool SIGMA are available online at: https://github.com/tiarkrompf/sigma

2 COLLECTIVE & CLOSED FORMS, STEP-BY-STEP

We take a program in the imperative source language IMP as input (Figure 3), and translate it into an equivalent functional program in our intermediate analysis language FUN (Figure 6), on which we perform symbolic simplification to expose the properties of interest (either after the translation or interleaved with it). For soundness, we require simplification to preserve all potential error conditions, but we do not, for example, need to preserve cases of divergence. FUN is a good intermediate representation for several reasons. First, it eschews side effects and makes all data dependencies explicit, which enables simplification through structural rewriting and enables us to represent IMP-level error conditions explicitly in the language. Second, it provides both recursive functions and collective forms, which enables us to gradually move from one to the other within the same language. While it can be helpful to think of the translation to FUN as a form of abstract interpretation of IMP programs, especially w.r.t. the Kleene iteration in Section 4, it is important to stress that the basic translation is exact (i.e., fully semantics preserving), without any approximation (details in Section 3).

As a running example, let us consider a simple while loop, which sums the integers from 0 to k−1 in variable s:

```
j := 0;
s := 0;
while j < k do {
  s := s + j;
  j := j + 1
}
```

Our goal is to characterize the program state after the loop, i.e., to obtain a mapping of the form [j ↦ ?, s ↦ ?], from program variables to abstract values (in our case, symbolic expressions).

The first step is to transform this IMP program into an equivalent FUN program. We make loop indices explicit and represent the values of j and s *after* a certain loop iteration i by a set of recursive functions, derived from the program text. For example, after the first iteration ($i = 0$), j and s are equal to 1 and 0 respectively, and are increased by 1 and $j(i-1)$ respectively for each iteration:

```
let j = λ(i). if i ≥ 0 then j(i − 1) + 1 else 0
let s = λ(i). if i ≥ 0 then s(i − 1) + j(i − 1) else 0
```

Now we can meaningfully talk about values at iterations i and $i-1$, and about their relationship: we reason in both space and time. We describe the trip count of the loop declaratively, as the first n for which the condition is false, using the built-in # functional—an example of a collective form:

```
let n = #(i). ¬(j(i) < k)
```

The operational interpretation of $\#(i).\ f(i)$ is to find the smallest $i \geq 0$ for which $f(i)$ evaluates to true or to diverge if no such i exists. Variable i is bound within the term following the dot, i.e, $f(i)$. We are now ready to describe the program state after the loop as a FUN term by mapping each

variable to a precise symbolic description of how it is computed using the previous definitions: $[\ \mathtt{j} \mapsto j(n-1)\ ,\ \mathtt{s} \mapsto s(n-1)\ ]$

We go on by identifying patterns in the recursive definitions. The following chain of rewrites transforms j and s to collective forms: an explicit sum construct, comparable to the mathematical Σ notation. For uniformity with other collective forms, we use the syntax $\Sigma(i < n).\ f(i)$ to denote the sum of all $f(i)$ for all $0 \le i < n$. Again, variable i is bound in the body of the term. As part of the simplification, β-reduction is performed for non-recursive functions:

$$
\begin{aligned}
&\textbf{let } j = \lambda(i).\ \textbf{if } i \ge 0 \textbf{ then } j(i-1)+1 \textbf{ else } 0\\
&\quad = \lambda(i).\ \Sigma(i_2 < i+1).\ 1\\
&\textbf{let } s = \lambda(i).\ \textbf{if } i \ge 0 \textbf{ then } s(i-1)+j(i-1) \textbf{ else } 0\\
&\quad = \lambda(i).\ \Sigma(i_3 < i+1).\ j(i_3-1)
\end{aligned}
$$

The collective sums for j, s are readily transformed to closed forms, which also provides a closed form loop count n:

$$
\begin{aligned}
&\textbf{let } j = \lambda(i).\ \Sigma(i_2 < i+1).\ 1\\
&\quad = \lambda(i).\ \textbf{if } i \ge 0 \textbf{ then } i+1 \textbf{ else } 0\\
&\textbf{let } s = \lambda(i).\ \Sigma(i_3 < i+1).\ j(i_3-1)\\
&\quad = \lambda(i).\ \Sigma(i_3 < i+1).\ i_3\\
&\quad = \lambda(i).\ \textbf{if } i \ge 0 \textbf{ then } (i+1) * i/2 \textbf{ else } 0\\
&\textbf{let } n = \#(i).\ \neg(j(i) < \mathtt{k})\\
&\quad = \#(i).\ \neg(i+1 < \mathtt{k})\\
&\quad = \textbf{if } \mathtt{k} \ge 0 \textbf{ then } \mathtt{k} \textbf{ else } 0
\end{aligned}
$$

With that, we obtain the desired closed form representation for the final program state based on $j(n-1)$ and $s(n-1)$:

$$[\ \mathtt{j} \mapsto \textbf{if } \mathtt{k} \ge 0 \textbf{ then } \mathtt{k} \textbf{ else } 0,\ \mathtt{s} \mapsto \textbf{if } \mathtt{k} \ge 0 \textbf{ then } \mathtt{k}*(\mathtt{k}-1)/2 \textbf{ else } 0\ ]$$

This symbolic representation can be used for multiple purposes at this point, either to verify programmer-specified assertions, as in Figure 1, to test equivalence of the source program with another one, or to generate optimized code (Section 6).

To keep the presentation high-level, we have deliberately omitted some details above, including exactly *how* recursive relations are converted into collective and closed forms. A simple approach can be realized based on pattern-based rewriting (Section 3.3), while the more sophisticated iterative approach based on speculative rewriting is discussed in Section 4, using the same running example.

2.1 Collective Forms for Arrays

We now turn our attention to dynamic memory operations. The simplest case is arrays. We modify our example program to first store the numbers in an array a, and then compute the sum by traversing the array a:

```
// build array of numbers < n
a := new;
l := 0;
while l < n do {
  a[l] := l;
  l := l + 1
}
```

```
// traverse array, compute Σ
j := 0;
s := 0;
while j < l do {
  s := s + a[j];
  j := j + 1
}
```

The additional challenge now is that our analysis needs to reason about array construction, as well as array traversal. In particular, we need to ensure that all array accesses are safe, and in addition, we need to precisely identify the values each array slot contains after the first loop, without any ambiguity. We solve this challenge by representing the array a using a closed form for sequence construction after the first loop, and recognizing that the second loop sums the elements of that sequence in s, which enables us again to use a collective sum expression.

The FUN representation is as follows. For simplicity, we show `l` and `j` already rewritten to closed forms, and the number of iterations already resolved to $n \geq 0$. The syntax `seq[i ↦ x]` denotes a copy of sequence `seq`, with the element at position `i` updated to `x`. We extract the recursive dependencies for the first loop:

let l = $\lambda(i)$. **if** $i \geq 0$ **then** $i + 1$ **else** 0
let a = $\lambda(i)$. **if** $i \geq 0$ **then** $a(i-1)[i \mapsto i]$ **else** []

We can now describe the state after the first loop (a(n−1) refers to the definition above):

`[ l ↦ n, a ↦` a`(n−1) ]`

Alas, this will not allow us to relate the array accesses of both loops. Can we do better? In addition to sums, products, and boolean connectives, our language FUN also contains collective form constructors for sequences. The notation $\langle . \rangle(\, i < n)\,.\ f(i)$ initializes a sequence or array with index range $i = 0, \ldots, n-1$, where each i is mapped to $f(i)$. Simplification proceeds as follows:

let a = $\lambda(i)$. **if** $i \geq 0$ **then** $a(i-1)[i \mapsto i]$ **else** []
= $\lambda(i)$. $\langle . \rangle(i_2 < i + 1)\,.\ i_2$

And we obtain a much more useful description of the state after the first loop:

`[ l ↦ n, a ↦` $\langle . \rangle(i <$ `n)`$.\ i$ `]`

We can then proceed for the second loop:

let j = $\lambda(i)$. **if** $i \geq 0$ **then** $i + 1$ **else** 0
let s = $\lambda(i)$. **if** $i \geq 0$ **then** $s(i-1)$ + a$[j(i-1)]$ **else** 0
= $\lambda(i)$. **if** $i \geq 0$ **then** $s(i-1)$ + $(\langle . \rangle(i_2 <$ n+1)$.\ i_2)[i]$ **else** 0

Simplifying the array access $(\langle . \rangle(i_2 < n + 1)\,.\ i_2)[i]$ to i depends on the presence of the enclosing loop precondition $i < n$, i.e., rewriting may be context- and flow-sensitive. Afterwards, simplification of `s` proceeds as before.

let s = $\lambda(i)$. **if** $i \geq 0$ **then** $s(i-1) + i$ **else** 0
= $\lambda(i)$. **if** $i \geq 0$ **then** $(i+1) * i/2$ **else** 0

Finally the state at the end of the program is:

`[ l ↦ n, j ↦ n, a ↦` $\langle . \rangle(i <$ `n)`$.\ i$`, s ↦ n*(n-1)/2 ]`

2.2 Collective Forms for Linked Structures

To complicate matters further, we might store the numbers in a linked list instead of an array, and build the sum by traversing the list, leading to the code from Figure 1:

```
// build list of numbers < n
x := null;
l := 0;
while l < n do {
  y := new;
  y.head := l; y.tail := x;
  x := y;
  l := l + 1
}
```

```
// traverse list, compute sum
z: = x;
s: = 0;
while z != null do {
  s := s + z.head;
  z := z.tail
}
```

Now we need to reason about individual heap cells, allocated in different iterations of the first loop, as well as strong updates to the `head` and `tail` fields in these dynamically allocated objects. At this point, our structured heap model introduced in Figure 2 plays a crucial role.

At runtime, there will be one object created for `y` per loop iteration `i`. While variable `y` holds the *address* of an object, we identify the actual object by its location `p` in the program text, indexed by the loop variables of its enclosing loops, i.e., `p[i]`, and refer to the freshly allocated (but deterministically chosen) address as `&new:p[i]`.

Here, p is an abbreviation for the precise path in the program tree, i.e., `root.snd.snd.while.fst`, and `p[i]` is an abbreviation for `root.snd.snd.while[i].fst`.

The notation `p[i]` already suggests that we can treat p just like an array of objects. After loop iteration i, x and y contain the address `&new:p[i]` of the latest allocated object:

```
let x = λ(i). if i ≥ 0 then &new:p[i] else null
let y = λ(i). if i ≥ 0 then &new:p[i] else ⊥, // ⊥ = uninitialized
```

The collection (array!) of objects p is defined and gets rewritten as follows:

```
let p = λ(i). if i ≥ 0 then p(i − 1)[i ↦ [tail ↦ x(i − 1), head ↦ i] ] else []
      = λ(i). ⟨.⟩(i₂ < i). [tail ↦ x(i₂-1), head ↦ i₂]
      = λ(i). ⟨.⟩(i₂ < i). [tail ↦ if i₂ > 0 then &new:p[i₂-1] else null, head ↦ i₂]
```

We can see how the recursive dependency between runtime objects is captured precisely. Note however that after simplification, p is no longer a recursive function: the address `&new:p[`$i_2 - 1$`]` is a purely syntactic term, which can be used to look up an object later by *dereferencing* the address.

After the first loop, the program state represents a proper store with static as well as dynamically allocated objects:

```
[ l ↦ n,
  x ↦ if n > 0 then &new:p[n-1] else null,
  y ↦ if n > 0 then &new:p[n-1] else ⊥
  p ↦ ⟨.⟩(i < n). [tail ↦ if (i > 0) then &new:p[i-1] else null, head ↦ i]
```

As indicated before, the structure of the store is hierarchical and mirrors the program structure. Dereferencing an address entails accessing the store. We will use the notation σ`[addr]`, but need to keep in mind that for a composite address like `&new:p[0]`, two steps of lookup are necessary: first by p, and then by 0.

The second loop leads to the following definitions of z, s:

```
let z = λ(i). if i ≥ 0 then σ[z(i − 1)][tail] else x
let s = λ(i). if i ≥ 0 then s(i − 1) + σ[z(i − 1)][head] else 0
```

Simplification by rewriting yields the desired closed forms (remember $i < n$ as we are within the loop):

```
let z = λ(i). if n > 0 then { if i ≥ 0 then σ[z(i − 1)][tail] else &new:p[n−1] } else null
      = λ(i). if i ≥ 0 then &new:p[n−1 − i] else null
let s = λ(i). if i ≥ 0 then (i + 1) ∗ i/2 else 0
```

Thus, we obtain the desired analysis result.

Throughout this section, we have glossed over some details. For example, we did not include explicit error checks in our translation, but checks for, e.g., validity of field accesses, need to be accounted for, and the details are described in Section 3.2. We also did not bother with variables that were obviously loop invariant. In reality, it is part of our analysis' job to determine which variables are the loop-invariant ones. We will return to this question in Section 4.

3 FORMAL MODEL

Since our approach hinges on translating imperative to functional code and applying transformation and simplification rules, it is imperative to formally establish the correctness of all these components to ensure the overall soundness of the approach. In addition, our structured store allocation model incurs some subtleties that warrant a formal description that explains the connection with standard store semantics.

In this section, we formalize the source and target languages and prove correctness of the translation, and the rewrite and simplification rules. This establishes the key result that an analysis engine that applies an arbitrary combination of equality-preserving simplifications will produce a

Expressions $\boxed{e \in \text{Exp}}$

$$n \in \text{Nat}, b \in \text{Bool}, x \in \text{Name}$$

$$
\begin{array}{lll}
e ::= & & \\
 & n \mid b \mid \&x & \text{Constant (nat, bool, addr)} \\
 & e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 & \text{Arithmetic} \\
 & e_1 < e_2 \mid e_1 = e_2 \mid e_1 \wedge e_2 \mid \neg e & \text{Boolean} \\
 & e_1[e_2] & \text{Field read}
\end{array}
$$

Statements $\boxed{s \in \text{Stm}}$

$$
\begin{array}{lll}
s ::= & & \\
 & x := \text{new} & \text{Allocation} \\
 & e_1[e_2] := e_3 & \text{Assignment} \\
 & \text{if } e \text{ then } s_1 \text{ else } s_2 & \text{Conditional} \\
 & \text{while } e \text{ do } s & \text{Loop} \\
 & s_1;\ s_2 & \text{Sequence} \\
 & \text{skip} & \text{No-op} \\
 & \text{abort} & \text{Error}
\end{array}
$$

Syntactic Sugar

$$
\begin{array}{lll}
x & \equiv \&x[0] \\
x := e & \equiv x[0] := e \\
e.x & \equiv e[\text{fieldId}(x)] \\
\text{assert } e & \equiv \text{if } e \text{ then skip else abort}
\end{array}
$$

Fig. 3. IMP: Surface language syntax.

sound result with respect to the semantics of the source language IMP. We have implemented our formal model in Coq and mechanized the results in this Section.

3.1 Source Language IMP

The syntax of our imperative model language IMP is defined in Figure 3. Our version of IMP is similar to imperative model languages found in a variety of textbooks, but is extended with dynamic memory operations and includes the possibility of certain runtime errors which reflect verification scenarios of practical interest.

IMP's syntax is split between expressions e and statements s. The language supports allocation statements $x := \text{new}$, as well as array or field references $e_1[e_2]$ (note that both array indices and fields can be computed dynamically) and corresponding assignments. Addresses of local variables $\&x$ are available as constant expressions, and variable references x are treated as syntactic sugar for dereferencing the corresponding address $\&x[0]$. Named field references $e.x$ are desugared into a numeric index assuming an injective global mapping `fieldId`. The `null` value is not part of the language, but can be understood as a dedicated address that is not otherwise used. It is important to note that there is no syntactic distinction between arithmetic and boolean expressions. Hence, evaluation may fail at runtime due to type errors or undefined fields. The syntax also includes an explicit `abort` statement for user-defined errors with `assert` as syntactic sugar.

Relational Semantics. The semantics of IMP is defined in big-step style, shown in Figure 4. Many of the evaluation rules are standard: expressions evaluate to values, and statements update the store. A store σ is a partial function from locations l to heap objects o, which are partial functions from numeric field indexes to values. We use square brackets to denote store or object lookup, i.e., $\sigma[l] = o$ and $o[n] = v$, as well as update, i.e., $\sigma[l \mapsto o]$ and $o[n \mapsto v]$. However, two aspects of the semantics deserve further attention.

Runtime Structures

$v \in \text{Val}$	$::= n \mid b \mid l$		Value (nat, bool, ptr)
$l \in \text{Loc}$	$::= \&x \mid \&\text{new}{:}c$		Store location (static, dynamic)
$o \in \text{Obj}$	$: \text{Nat} \rightharpoonup \text{Val}$		Object
$\sigma \in \text{Sto}$	$: \text{Loc} \rightharpoonup \text{Obj}$		Store

$c \in \text{Ctx} ::=$		Context path
	root	At top level
	$c.\text{then} \mid c.\text{else}$	In conditional
	$c.\text{fst} \mid c.\text{snd}$	In sequence
	$c.\text{while}[n]$	In loop (iteration n)

Expression Evaluation $\boxed{\sigma \vdash e \Downarrow v}$

$$\frac{}{\sigma \vdash n \Downarrow n}\ (\text{ENum}) \qquad \frac{\sigma \vdash e_1 \Downarrow n_1 \qquad \sigma \vdash e_2 \Downarrow n_2}{\sigma \vdash e_1 + e_2 \Downarrow n_1 + n_2}\ (\text{EPlus})$$

$$\frac{\sigma \vdash e_1 \Downarrow l_1 \qquad \sigma \vdash e_2 \Downarrow n_2 \qquad \sigma[l_1] = o \qquad o[n_2] = v_3}{\sigma \vdash e_1[e_2] \Downarrow v_3}\ (\text{EField})$$

Loop Evaluation $\boxed{\sigma, c \vdash (e\ s)^n \Downarrow \sigma'}$

$$\frac{}{\sigma, c \vdash (e\ s)^0 \Downarrow \sigma}\ (\text{EWhileZero})$$

$$\frac{\sigma, c \vdash (e\ s)^n \Downarrow \sigma' \qquad \sigma' \vdash e \Downarrow \text{true} \qquad \sigma', c.\text{while}[n] \vdash s \Downarrow \sigma''}{\sigma, c \vdash (e\ s)^{n+1} \Downarrow \sigma''}\ (\text{EWhileMore})$$

Statement Evaluation $\boxed{\sigma, c \vdash s \Downarrow \sigma'}$

$$\frac{}{\sigma, c \vdash x := \text{new} \Downarrow \sigma[\&\text{new}{:}c \mapsto [],\ \&x \mapsto [0 \mapsto \&\text{new}{:}c]]}\ (\text{ENew})$$

$$\frac{\sigma \vdash e_1 \Downarrow l_1 \qquad \sigma \vdash e_2 \Downarrow n_2 \qquad \sigma \vdash e_3 \Downarrow v_3 \qquad \sigma[l_1] = o}{\sigma, c \vdash e_1[e_2] := e_3 \Downarrow \sigma[l_1 \mapsto o[n_2 \mapsto v_3]]}\ (\text{EAssign})$$

$$\frac{\sigma \vdash e \Downarrow \text{true} \qquad \sigma, c.\text{then} \vdash s_1 \Downarrow \sigma'}{\sigma, c \vdash \text{if } e \text{ then } s_1 \text{ else } s_2 \Downarrow \sigma'}\ (\text{EIfTrue})$$

$$\frac{\sigma \vdash e \Downarrow \text{false} \qquad \sigma, c.\text{else} \vdash s_2 \Downarrow \sigma'}{\sigma, c \vdash \text{if } e \text{ then } s_1 \text{ else } s_2 \Downarrow \sigma'}\ (\text{EIfFalse})$$

$$\frac{\sigma, c \vdash (e\ s)^n \Downarrow \sigma' \qquad \sigma' \vdash e \Downarrow \text{false}}{\sigma, c \vdash \text{while } e \text{ do } s \Downarrow \sigma'}\ (\text{EWhile})$$

$$\frac{\sigma, c.\text{fst} \vdash s_1 \Downarrow \sigma' \qquad \sigma', c.\text{snd} \vdash s_2 \Downarrow \sigma''}{\sigma, c \vdash s_1;\ s_2 \Downarrow \sigma''}\ (\text{ESeq})$$

$$\frac{}{\sigma, c \vdash \text{skip} \Downarrow \sigma}\ (\text{ESkip})$$

Fig. 4. IMP: Relational big-step semantics (primitive constants and operators other than n and + elided). Note the evaluation rules for `while` loops and the role of program context c for address allocation in rule (ENew).

First, store addresses are not flat but have structure. For dynamic allocations, rule (ENew) deterministically assigns a fresh store address &new:c where c is the current program context c. This context is maintained throughout all statement rules and uniquely determines the *spatio-temporal* point of execution in the program. It combines static information, i.e., the location in the program text, with dynamic information, i.e., progress of execution, represented by the current iteration vector of all enclosing loops. This will later enable us to talk about abstract locations from within the same loop, but at different iterations.

Second, `while` loops are executed with the help of an auxiliary judgement $\sigma, c \vdash (e\ s)^n \Downarrow \sigma'$, which characterizes the result of executing a loop body n times. This already hints at what we want to achieve later: replace iteration by a collective form for a given n. Declaratively, the number of iterations a loop will be executed is the particular n after which the condition becomes false. Operationally, rule (EWhile) has to "guess" the correct n. While not necessarily the best fit for deriving an *implementation*, this formulation of while loops renders the semantics compositional [Siek 2016, 2017], a good basis for deriving a semantics-preserving translation.

Given these differences, we first establish equivalence of the given semantics with a more standard formulation that assigns store locations nondeterministically, and defines while loops without explicit reference to iteration numbers.

Runtime Structures

$o \in \text{Obj} : \text{Nat} \to \text{Option Val}$ Object

$\sigma \in \text{Sto} : \text{Loc} \to \text{Option Obj}$ Store

Monad operations:

$m \in \text{Option } T ::= \text{None} \mid \text{Some } \tau \text{ where } \tau \in T$

$x \leftarrow m;\ f(x) = m \gg\!= f$

$\gg\!= : \text{Option } T \to (T \to \text{Option } U) \to \text{Option } U$

$\text{getOrElse} : \text{Option } T \to T \to T$

$\text{toNat} : \text{Val} \to \text{Option Nat}$

$\text{toBool} : \text{Val} \to \text{Option Bool}$

$\text{toLoc} : \text{Val} \to \text{Option Loc}$

Iteration primitive:

$\# : (\text{Nat} \rightharpoonup \text{Bool}) \rightharpoonup \text{Nat}$

$\#f = g(0)$ where
$g(i) = \text{if } f(i) \text{ then } i \text{ else } g(i+1)$

Expression Evaluation $\boxed{[\![\ e\ ]\!](\sigma) = v}$

$$
\begin{aligned}
[\![\ .\ ]\!] &: \text{Exp} \to \text{Sto} \to \text{Option Val} \\
[\![\ n\ ]\!](\sigma) &= \text{Some } n \\
[\![\ e_1 + e_2\ ]\!](\sigma) &= n1 \leftarrow [\![\ e_1\ ]\!](\sigma) \gg\!= \text{toNat}; \\
&\quad\ n2 \leftarrow [\![\ e_2\ ]\!](\sigma) \gg\!= \text{toNat} \\
&\quad\ \text{Some } (n1 + n2) \\
\ldots &= \ldots \\
[\![\ x\ ]\!](\sigma) &= o \leftarrow \sigma[\&x];\ o[0] \\
[\![\ e_1[e_2]\ ]\!](\sigma) &= l \leftarrow [\![\ e_1\ ]\!](\sigma) \gg\!= \text{toLoc}; \\
&\quad\ n \leftarrow [\![\ e_2\ ]\!](\sigma) \gg\!= \text{toNat}; \\
&\quad\ o \leftarrow \sigma[l]; \\
&\quad\ o[n]
\end{aligned}
$$

Loop Evaluation $\boxed{[\![\ e\ s\ ]\!](\sigma, c)(n) = \sigma'}$

$$
\begin{aligned}
[\![\ .\ ]\!] &: \text{Exp} \times \text{Stm} \to \text{Sto} \times \text{Ctx} \to \text{Nat} \\
&\quad \to \text{Option Sto} \\
[\![\ e\ s\ ]\!](\sigma, c)(n) &= f(n) \text{ where} \\
&\quad f(0) = \text{Some } \sigma \\
&\quad f(n+1) = \sigma' \leftarrow f(n) \\
&\qquad\ \text{true} \leftarrow [\![\ e\ ]\!](\sigma') \gg\!= \text{toBool} \\
&\qquad\ [\![\ s\ ]\!](\sigma', c.\text{while}[n])
\end{aligned}
$$

Statement Evaluation $\boxed{[\![\ s\ ]\!](\sigma, c) = \sigma'}$

$$
\begin{aligned}
[\![\ .\ ]\!] &: \text{Stm} \to \text{Sto} \times \text{Ctx} \\
&\quad \to \text{Option Sto} \\
[\![\ x := \text{new}\ ]\!](\sigma, c) &= \sigma[\&\text{new}{:}c \mapsto [], \\
&\qquad \&x \mapsto [0 \mapsto \&\text{new}{:}c]] \\
[\![\ e_1[e_2] := e_3\ ]\!](\sigma, c) &= l \leftarrow [\![\ e_1\ ]\!](\sigma) \gg\!= \text{toLoc} \\
&\quad\ n \leftarrow [\![\ e_2\ ]\!](\sigma) \gg\!= \text{toNat} \\
&\quad\ v \leftarrow [\![\ e_3\ ]\!](\sigma); \\
&\quad\ o \leftarrow \sigma[l]; \\
&\quad\ \sigma[l \mapsto o[n \mapsto v]] \\
[\![\ \text{if } (e)\ s_1 \text{ else } s_2\ ]\!](\sigma, c) &= b \leftarrow [\![\ e\ ]\!](\sigma) \gg\!= \text{toBool} \\
&\quad\ \text{if } b \text{ then } [\![\ s_1\ ]\!](\sigma, c.\text{then}) \\
&\quad\ \text{else } [\![\ s_2\ ]\!](\sigma, c.\text{else}) \\
[\![\ \text{while } e \text{ do } s\ ]\!](\sigma, c) &= [\![\ e\ s\ ]\!](\sigma, c)(n) \text{ where} \\
&\quad\ n = \#(\lambda i.(\\
&\qquad\ \sigma' \leftarrow [\![\ e\ s\ ]\!](\sigma, c)(i) \\
&\qquad\ b \leftarrow [\![\ e\ ]\!](\sigma') \gg\!= \text{toBool} \\
&\qquad\ \text{Some } \neg b) \text{ getOrElse true}) \\
[\![\ s_1;\ s_2\ ]\!](\sigma, c) &= \sigma' \leftarrow [\![\ s_1\ ]\!](\sigma, c.\text{fst}) \\
&\quad\ [\![\ s_2\ ]\!](\sigma', c.\text{snd}) \\
[\![\ \text{skip}\ ]\!](\sigma, c) &= \text{Some } \sigma \\
[\![\ \text{abort}\ ]\!](\sigma, c) &= \text{None}
\end{aligned}
$$

Fig. 5. `IMP` Functional semantics with explicit errors, partiality reserved for divergence. Note the use of monad operations throughout and the use of an iteration primitive in the evaluation of `while` loops.

Definition 3.1 (Standard Semantics). Let $\Downarrow^0$ be the relation derived from $\Downarrow$ by dropping contexts c and replacing rules (ENew) and (EWhile) with the following rules:

$$\frac{\&\text{new}{:}n \notin \sigma}{\sigma \vdash x := \text{new} \Downarrow \sigma[\&\text{new}{:}n \mapsto [], \&x \mapsto [0 \mapsto \&\text{new}{:}n]]} \quad \text{(ENewN)}$$

$$\frac{\sigma \vdash e \Downarrow^0 \text{true} \qquad \sigma \vdash s \Downarrow \sigma' \qquad \sigma' \vdash \text{while } (e)\ s \Downarrow^0 \sigma''}{\sigma \vdash \text{while } (e)\ s \Downarrow^0 \sigma''} \quad \text{(EWhileTrue)}$$

$$\frac{\sigma \vdash e \Downarrow^0 \text{false}}{\sigma \vdash \text{while } (e)\ s \Downarrow^0 \sigma} \quad \text{(EWhileFalse)}$$

Proposition 3.2 (Adequacy of $\Downarrow$). *$\Downarrow^0$ and $\Downarrow$ are equivalent, up to a bijection between store addrs &new:n and &new:c.*

We now study key properties of our semantics. First, we show that $\Downarrow$ is deterministic, and hence we can understand it as a partial function $\text{eval}_{\Downarrow}$.

Proposition 3.3 (Determinism). *The semantics is deterministic: if* $\sigma, c \vdash s \Downarrow \sigma'$ *and* $\sigma, c \vdash s \Downarrow \sigma''$ *then* $\sigma' = \sigma''$.

Definition 3.4 (Initial Store). Let $\sigma_\emptyset$ be the store with $\sigma_\emptyset[\&x] = []$ and $\sigma_\emptyset[\&\text{new}{:}c]$ undefined for all x and c.

Definition 3.5. Let $\text{eval}_\Downarrow(s) = \sigma$ iff $\sigma_\emptyset, \text{root} \vdash s \Downarrow \sigma$, and undefined otherwise.

Error Behavior. As presented, the semantics does not distinguish error cases from undefinedness due to divergence. If our goal is program verification, then we need to isolate the error cases precisely and introduce a distinction.

Proposition 3.6. *For all* s, $\text{eval}_\Downarrow(s)$ *is either: (1) a unique result* σ*, (2) undefined due to divergence (i.e., there exists a loop in the program for which the condition is true for all* n *in rule (EWhile)), or (3) undefined due to one of the following possible errors: type error (Nat, Bool, Loc), reference to nonexistent store location, reference to nonexistent object field, explicit* `abort`

Proof. We show that the property holds up to a given upper bound n on the number of iterations any loop can execute, and do induction over n. □

Functional Semantics. Based on these observations, we define a second semantics that makes all error conditions explicit by wrapping potentially failing computations in the `Option` monad, and which also replaces the nondeterminism in rule (EWhile) with an explicit and potentially diverging search for the correct number of iterations. With these modifications, the semantics can be expressed in a denotational style, directly as partial functions. We show the definition in Figure 5. Functions $[\![\ .\ ]\!]$ now take the role of the relation $\Downarrow$, and partial functions that could be undefined due to runtime errors are now replaced by total functions that return an `Option` T instance, i.e., either `None` to indicate an error or `Some` τ with a T-value τ to signal success. The evaluation of expressions becomes entirely total. The only remaining source of partiality is the potentially diverging computation of loop iterations $n = \#(\lambda i. \ldots)$.

Definition 3.7. Let $\text{eval}_{[\![\ ]\!]}(s) = [\![\ s\ ]\!](\sigma_\emptyset, \text{root})$.

Proposition 3.8. *For all* s, $\text{eval}_{[\![\ ]\!]}(s)$ *is either: (1) a unique result* `Some` σ*, (2) an explicit error* `None`*, or (3) undefined due to divergence.*

Proposition 3.9 (Adequacy). *For all* s, $\text{eval}_\Downarrow(s)$ *and* $\text{eval}_{[\![\ ]\!]}(s)$ *agree exactly on their value, error, and divergence behavior.*

Proof. By induction on an assumed upper bound on the number of iterations per loop. □

This functional semantics has the appealing property of denotational formulations that we can directly read it as a translation from `IMP` to mathematics. By mapping the mathematical notation into (a subset of) our target language `FUN`, we obtain a translator from `IMP` to `FUN`.

3.2 Target Language `FUN`

The syntax of `FUN` is defined in Figure 6. `FUN` is a functional language based on λ-calculus, with expressions g as the only syntactic category. The primitive data types are natural numbers, booleans, store addresses, and objects, i.e., records with numeric keys. In addition, `FUN` has a rich set of collective operators for sums, products, forall, and exists. While the presention here only covers those four monoids, the technique presented in this paper can be easily generalized to other monoids.

Figure 7 summarizes how the various entities in the definition of `IMP`'s functional semantics in Figure 5 map to `FUN` constructs. This enables us to directly read the given `IMP` semantics as translation rules.

Expressions $g \in \text{Exp}$

$g ::=$			
$n \mid b \mid l \mid []$	Const (nat, bool, loc, obj)		
x	Variable	$e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2$	Arithmetic
$e_1 \ni e_2$	Field exists?	$e_1 < e_2 \mid e_1 = e_2 \mid \neg e$	Logic
$e_1[e_2]$	Field read	if e then s_1 else s_2	Conditional
$e_1[e_2 \mapsto e_3]$	Field update	letrec $x_1 = g_1, \ \ldots$ in g_n	Recursive let
$g_1(g_2)$	Application	$\Sigma(x < g_1).\ g_2$	Sum
$\lambda(x).\ g$	Function	$\Pi(x < g_1).\ g_2$	Product
$\#(x).\ g$	First index	$\forall\ (x < g_1).\ g_2$	Conjunction
$\langle . \rangle (x < g_1).\ g_2$	Sequence	$\exists\ (x < g_1).\ g_2$	Disjunction
		$\ldots$	
$w ::=$	Value		
$n \mid b \mid l$	Constant	$[n_0 \mapsto w_0, \ \ldots]$	Object

Fig. 6. FUN: Target language syntax.

Translation

$$
\begin{aligned}
\text{None} &= [\text{valid} \mapsto \text{false}] \\
\text{Some } g &= [\text{valid} \mapsto \text{true}, \text{data} \mapsto g] \\
g \gg\!= f &= \text{if } g.\text{valid then } f(g.\text{data}) \text{ else None} \\
g_1 \text{ getOrElse } g_2 &= \text{if } g_1.\text{valid then } g_1.\text{data else } g_2 \\
\text{Val } n &= [\text{tpe} \mapsto \text{nat}, \text{val} \mapsto n] \\
\text{Val } b &= [\text{tpe} \mapsto \text{bool}, \text{val} \mapsto b] \\
\text{Val } l &= [\text{tpe} \mapsto \text{loc}, \text{val} \mapsto l] \\
\text{toNat } g &= \text{if } g.\text{tpe} = \mathit{nat} \text{ then Some } g.\text{val else None} \\
\text{toBool } g &= \text{if } g.\text{tpe} = \mathit{bool} \text{ then Some } g.\text{val else None} \\
\text{toLoc } g &= \text{if } g.\text{tpe} = \mathit{loc} \text{ then Some } g.\text{val else None} \\
o[n] &= \text{if } o \ni n \text{ then } [\text{valid} \mapsto \text{true}, \text{data} \mapsto o[n]] \text{else } [\text{valid} \mapsto \text{false}] \\
\sigma[l] &= \text{if } \sigma \ni l \text{ then } [\text{valid} \mapsto \text{true}, \text{data} \mapsto \sigma[l]] \text{else } [\text{valid} \mapsto \text{false}]
\end{aligned}
$$

Fig. 7. FUN: Mathematical notation as syntactic sugar, and value representation for the Option Monad. This enables reading Figure 5 as translation from IMP to FUN. Every monadic value has a valid flag that is set to false to indicate an error condition.

Proposition 3.10. *Functions* $[\![\ e\ ]\!](\sigma)$ *and* $[\![\ s\ ]\!](\sigma, c)$ *accomplish translation from IMP to FUN.*

The semantics of FUN follows the standard call-by-value (CBV) λ rules. The collective operators trivially map to recursive definitions. The only non-trivial addition is the mapping of store locations to numeric keys for record access.

Definition 3.11. Let $\rightarrow_v$ be the standard CBV λ reduction, extended to FUN with the following rules:

$$
\begin{aligned}
w[\&\text{new:fst}.p] &\rightarrow w[\text{fst}][\&\text{new}:p] \\
w[\&\text{new:snd}.p] &\rightarrow w[\text{snd}][\&\text{new}:p] \\
w[\&\text{new:while}[n].p] &\rightarrow w[\text{while}][n][\&\text{new}:p] \\
&\ldots
\end{aligned}
$$

Here we assume a standard mapping from names like 'fst', 'while', etc., to numbers, as before. We can see now that at the FUN level, the IMP store assumes a nested structure, mapping all values allocated in a loop into an array indexed by the loop variable. The update and field test rules are analogous to the lookup rules shown.

Structural Equivalences

$$
\begin{array}{rcl}
a[k \mapsto u][j] & \equiv & \text{if } j = k \text{ then } u \text{ else } a[j] \\
C[\text{if } c \text{ then } u \text{ else } v] & \equiv & \text{if } c \text{ then } C[u] \text{ else } C[v] \\
\text{letrec } f = (\lambda(i).[a \mapsto u, b \mapsto v]) \text{ in } e & \equiv & \text{letrec } f = (\lambda(i).[a \mapsto f_a(i), \mapsto f_b(i)]) \\
 & & \quad f_a = (\lambda(i).u), f_b = (\lambda(i).v) \text{ in } e \\
\text{if } 1 < e \text{ then } x \text{ else } y & \equiv & \text{if } 0 < e \text{ then } x \text{ else } y \\
 & & \quad \textit{if } x \equiv y \textit{ when } e = 0 \\
\text{if } a < e \text{ then if } b < e \text{ then } x' \text{ else } y' \text{ else } y & \equiv & \text{if } b < e \text{ then } x' \text{ else } y' \\
 & & \quad \textit{if } y \equiv y' \textit{ and } a \leq b \\
 & \ldots & \\
\end{array}
$$

Collective Forms

$$
\begin{array}{rcl}
\text{letrec } f = (\lambda(i).\ \text{if } 0 \leq i \text{ then} & & \\
\quad f(i-1)[i \mapsto g_i] \text{ else } []) \text{ in } f(a) & \equiv & \langle . \rangle (i < a + 1).\ g_i \\
\text{letrec } f = (\lambda(i).\ \text{if } 0 \leq i \text{ then} & & \\
\quad f(i-1) + g_i \text{ else } 0) \text{ in } f(a) & \equiv & \Sigma(i < a + 1).\ g_i \\
x + \Sigma(i < a).\ g_i & \equiv & \Sigma(i < a + 1).\ g_i \\
 & & \quad \textit{if } x \equiv g_i[i := a] \\
\Sigma(i < n).\ i & \equiv & \text{if } 0 < n \text{ then } n * (n-1)/2 \text{ else } 0 \\
\#(i).\ \neg(i < a) & \equiv & \text{if } 0 < a \text{ then } a \text{ else } 0 \\
\text{if } 0 < \#(i).\ g \text{ then } s1 \text{ else } s2 & \equiv & s1 \quad \textit{if } s1[(\#(i).\ g) := 0] \equiv s2 \\
 & \ldots & \\
\end{array}
$$

Fig. 8. Selected equivalences (non-exhaustive) that can be used as simplification rules, using a variety of deterministic or nondeterministic rewriting strategies ($x[y := z]$ means that y is substituted by z in x).

Proposition 3.12. *Evaluating a FUN expression via $\rightarrow_v^*$ can either: (1) terminate with a value, (2) terminate with a stuck term, or (3) diverge.*

We are now ready to express our main soundness result for the translation from IMP to FUN.

Definition 3.13. Let $v \cong w$ be the equivalence between IMP values v and FUN values w induced by the value representation in Figure 7. Let $w \cong \sigma$ be the relation extended to IMP stores in the nested representation defined above.

Theorem 3.14. *Translation from IMP to FUN is semantics preserving: For any IMP terms e or s translated to FUN via $[\![e]\!]$ or $[\![s]\!]$, FUN execution via $\rightarrow_v^*$ never gets stuck. Values and stores map to their equivalents $v \cong w$ and $\sigma \cong w$, errors map to clearly identified error values, and divergence to divergence.*

Proof. Again, by induction on an appropriate upper bound on the number of any loop iterations. □

3.3 Analysis and Verification via Simplification

Based on the sound translation from IMP to FUN, we now want to simplify FUN programs and extract higher-level information. In particular, we want to transform recursive definitions into collective operations. This approach to program analysis is similar in spirit to deductive verification: we translate the source language into an equivalent representation (or program logic) which can then be solved through a solver procedure (e.g., constraint simplification). In our case, the target language is the functional language FUN, and the solver performs simplification through equality-preserving rewriting of program terms (we discuss a strategy that interleaves translation and simplification in Section 4). For the concrete rewriting strategy there is considerable freedom, and we do not fix a particular strategy here.

Verification Based on Explicit Errors Values. The key property of the IMP to FUN translation was that any runtime error in the IMP program will be reflected as an observable error *value* in the target language, but not trigger erroneous behavior there (Theorem 3.14). Based on this property, verification just amounts to checking that the FUN program cannot produce a failure result. All that is required for this test is a syntactic check that the FUN program after simplification is equal to Some g—in other words, that the valid flag according to the value representation in Figure 7 statically simplifies to constant true. If the valid flag is any other symbolic expression, it means that verification did not succeed; i.e., that the program might exhibit erroneous behavior (such as an assertion failure).

Soundness of Simplification Rules. The property that any error in the IMP program will be reflected as an observable error value in FUN follows from the semantic preservation of the CBV FUN semantics. For purposes of verification, however, we may settle for a weaker correspondence, and pick a non-strict call-by-name semantics for FUN. This provides more flexibility for simplification, e.g., the ability to rewrite $0 * e \rightarrow 0$ even if evaluation of e may not terminate, but it also means that some diverging IMP programs may terminate in their FUN translation. In this case, verification may signal false positive errors. For example, for `while true do skip; assert false`, the analysis might miss that the assert is unreachable. Importantly, this result is still sound.

In the following, we therefore assume a non-strict call-by-name (CBN) or call-by-need semantics for FUN, and recall confluence of λ-calculus and that CBN terminates on more programs that CBV. We need a few other standard results:

Definition 3.15. Let $\rightarrow$ be the standard CBN λ reduction, extended to FUN as above. Let $\mathcal{E}[\![\ g\ ]\!]$ be the partial evaluation function induced by $\rightarrow^*$.

Definition 3.16 (Behavioral Equivalence). Let $g_1 \equiv g_2$ iff $\mathcal{E}[\![\ g_1\ ]\!] = \mathcal{E}[\![\ g_2\ ]\!]$.

Proposition 3.17 (Congruence). *For any context C, if $g_1 \equiv g_2$, then $C[g_1] \equiv C[g_2]$*

The congruence property enables us to prove the correctness of individual rewrite rules, and use them to soundly replace parts of an expression with behaviorally equivalent ones.

Simplification Rules in Practice. We show selected equivalences that give rise to useful rewrite rules for simplification in Figure 8. Besides standard arithmetic simplification, there are structural rules about objects and their fields. In particular, a key simplification is to split recursive functions into individual functions per object field. Since the IMP store is represented as a FUN object, this rule enables local reasoning about individual IMP variables, instead of only about the store as a whole. Combined with β-reduction for non-recursive functions, the original function may be replaced entirely by the component-wise ones. The same pattern also applies to the construction of sequences: instead of creating an array of objects, it is often better to create an object of arrays, one per field. If only parts of an object change, this enables a more detailed characterization of such changes. In addition, it is often helpful to distribute conditionals over other operations, e.g., to push conditionals into object fields. Another important set of rules is concerned with the actual extraction of collective forms for sums, sequences/arrays, etc. The $\#(i)$ rule is key for numeric loop bounds. It is also useful to add standard dead-code and common subexpression rules. The last rewriting rule related to $\#(i)$ in Figure 8 is quite interesting. After analyzing a loop, the resulting store will always be of the form: if the loop executed at least once ($0 < \#(i).g$) the new store is $s1$ otherwise it is $s2$. However, if the loop did not execute, $\#(i).g$ must be 0. Therefore if $s1$ is equivalent to $s2$ when the loop does not execute, then the condition can be removed, and the new store is $s1$.

We believe that it is a strong benefit of our approach that the set of simplification rules (Figure 8 and beyond) is not fixed and can be extended at any point. The only requirement for a rule is to individually preserve the CBN concrete semantics.

Rewriting Strategies. The set of equivalence rules available for simplification, including the rules in Figure 8 and beyond, gives rise to a whole space of rewriting opportunities. If each rule individually is proved to preserve semantics, an implementation is free to apply them in any order to reach a sufficiently simplified program. A simple and performance-oriented implementation can use a deterministic bottom-up strategy, but it would be entirely feasible to use auto-tuning, heuristic search, or strategies based on machine learning.

However, even search-based rewriting strategies are fundamentally limited by their pessimistic nature if they apply simplification rules one by one, due to the requirement that each individual rule preserves the program semantics, as opposed to a set of rules applied at once. Section 4 discusses an optimistic strategy based on Kleene iteration that removes this restriction and leads to more precise results in practice.

4 SPECULATIVE REWRITING & KLEENE ITERATION

The analysis and verification approach presented in Section 3.3 is based on applying equality-preserving simplification rules after a full `IMP` program is translated to `FUN`. While a useful starting point, this approach has clear limitations.

Fundamentally, equality-preserving simplification has to operate with *pessimistic* assumptions around loops and other recursive dependencies. We can only simplify a program if we are *sure* that each individual step will preserve the full extent of the program's semantics. In this section we will refine our approach towards *optimistic* simplification: this approach will simplify loop bodies no matter what, and *check* whether the simplification is indeed valid. If not, we try somewhat less optimistic assumptions, and repeat. This is inspired by Lerner et al.'s work on composing dataflow analyses and transformations in optimizing compilers [Lerner et al. 2002].

Errors and Loop-Invariant Fields. Concretely, equality-preserving rewriting works well as long as there are no mutually recursive dependencies, i.e., there is always one recursive function that can be rewritten first, leading to further rewriting opportunities in other functions. But this is not always the case. Consider the following program:

```
j := 0;
while j < n do {
  assert(j >= 0); j := j + 1
}
```

Recall that errors are represented as a `valid` flag in the record representing the overall program state (see Section 3). The `valid` flag is equivalent to a variable `v` initialized to `true` at the beginning of the program, and set to `false` if the `assert` fails. To illustrate, the program could be rewritten as follows:

```
j := 0;
v := true;
while j < n ∧ v do {
  if j >= 0 then j := j + 1 else v := false
}
```

To demonstrate the absence of errors, we need to demonstrate that the `valid` flag remains unchanged throughout the loop. However, this is difficult since the derived `FUN` representation contains mutual recursion between variables.

```
let j = λ(i). if i ≥ 0 then { if v(i − 1) ∧ j(i − 1) ≥ 0 then j(i − 1) + 1 else j(i − 1) } else 0
let v = λ(i). if i ≥ 0 then { if v(i − 1) ∧ j(i − 1) ≥ 0 then v(i − 1) else false } else true
let n = #(i). ¬(j(i) < n ∧ v(i))
```

In this situation, we cannot extract an individual recursive function for j (and much less a collective form) because the loop body may raise an error (set v to false), and we cannot eliminate v because we do not have enough knowledge about j. Thus the basic rewriting strategy from Section 2 cannot work.

We will explain the process in more detail based on a concrete example of scalar recurrences below, but it is important to note that the approach is more general and applies to all kinds of expressions and data types. To complete the verification of the program above, recall that error conditions are represented as a `valid` flag in the record representing the overall program state (see Section 3). To demonstrate the absence of errors, we need to demonstrate that the `valid` flag remains unchanged throughout a loop. Fortunately, identifying loop-invariant parts of data structures is straightforward with the speculative rewriting approach: we make initial optimistic assumptions that all variables and record fields (including the valid flag as special case), are loop-invariant, and roll back these assumptions only if writes to certain vars/fields are observed. With optimistic assumptions, there is no write to v in the loop, and the program verifies. The following table illustrates the simplification process, that terminates once a fixpoint has been reached.

	Before loop y_0	Before ith iter. $\hat{f}(i-1)$	After (expected) $\hat{f}(i)$	After (actual) $\Delta(\hat{f}(i-1))$
Step 1	j = 0	0	0	1
	v = true	true	true	true
Step 2	j = 0	i	i + 1	i + 1
	v = true	true	true	true

Scalar Recurrences. Consider the example from Section 2:

```
j := 0; s := 0;
while j < k do {
  s := s + j;
  j := j + 1
}
```

Our refined approach is as follows: let y_0 be the program state before the loop and let Δ be the transfer function of the loop, describing the effect of one loop iteration on the program state. We use $f(i)$ to denote the program state after iteration i, subject to $f(-1) = y_0$ and $f(i+1) = \Delta(f(i))$. The goal is now to approximate f iteratively by a series of *increasingly pessimistic* functions $\hat{f}_k$ until we reach f.

At the first step $\hat{f}_0$ we assume (maximum optimism) all variables to be loop invariant, i.e., that we can use the following per-variable functions, where n is the current symbolic value of k:
`let` $k = \lambda(i).$n, `let` $j = \lambda(i).0$, `let` $s = \lambda(i).0$

Then, we evaluate the assumed functions to compute the expected value before and after loop iteration i, i.e., $\hat{f}_0(i-1)$ and $\hat{f}_0(i)$. We compare the expected post-iteration value with the actual symbolic evaluation of the loop body, using the same expected initial values $\Delta(\hat{f}_0(i-1))$ (Figure 9, top). In general, if $\Delta(\hat{f}_k(i)) = \Delta(\hat{f}_k(i-1))$ for a symbolic representation of i, we know that $\hat{f}_k = f$. But in this case, we can see that our assumption about j was too optimistic. We need to try a non-loop-invariant transfer function — but which one?

For scalar values, one of our strategies is to focus on polynomials. The observed difference d_j between before and after the loop iteration can be seen as the discrete derivative of the transfer function we are approximating. In this case, d_j is a constant, i.e., a polynomial in i of degree 0. Thus we try the (uniquely defined) polynomial of degree 1 (a linear function) that matches the observed values for $i = 0$, $j = 1$ and has derivative $d_j = 1$. `let` $k = \lambda(i).$n, `let` $j = \lambda(i).i+1$, `let` $s = \lambda(i).0$

The computed expected and actual values are shown in Figure 9 (middle). Now the representation of j has been settled, but s is no longer correct. We follow the same strategy as before and generalize

Before loop y_0	Before ith iter. $\hat{f}(i-1)$	After (expected) $\hat{f}(i)$	After (actual) $\Delta(\hat{f}(i-1))$
k = n	n	n	n
j = 0	0	0	1
s = 0	0	0	0
k = n	n	n	n
j = 0	i	i + 1	i + 1
s = 0	0	0	i
k = n	n	n	n
j = 0	i	i + 1	i + 1
s = 0	((i-1)*i)/2	(i*(i+1))/2	(i*(i+1))/2

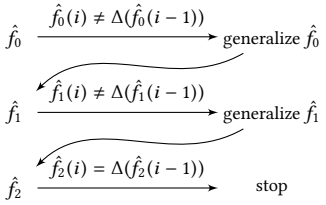

Fig. 9. Fixpoint iteration for running example, iterations 0 (top) to 2 (bottom), converging to a 2nd-degree polynomial for s. The generalization treats different data types differently: (1) try a higher degree of polynomial for numerics, (2) apply generalization to fields recursively for records, (3) extract the collective form for arrays if writing to the adjacent slot, or (4) create a recursive function for fallback.

the transfer function for s. The difference $d_s = i$ is a polynomial of degree 1, and discrete integration yields a quadratic function:

`let` $k = \lambda(i).$`n`, `let` $j = \lambda(i).i + 1$, `let` $s = \lambda(i).(i * (i + 1))/2$

Now we observe convergence, shown in Figure 9 (bottom). Therefore, our strategy succeeded and we simultaneously computed sound symbolic representations of k, j, and s. We can now compute the number of iterations executed: $\#(i).\ \neg(j(i) < \text{n}) =$ if 0 < n then n else 0. Thus, the last iteration executed was n−1 (or -1 if the loop was not executed at all), and the values of k, j, and s after the loop are therefore:

```
[ k ↦ if 0 < n then k(n−1) else n = n,
  j ↦ if 0 < n then j(n−1) else 0 = if (0 < n) then n else 0,
  s ↦ if 0 < n then s(n−1) else 0 = if (0 < n) then (n-1)*n/2 else 0 ]
```

In general, polynomials are just one option. Since not all functions can be described as polynomials, we cannot rely on convergence, i.e., we need to stop at a certain degree. In the event that the analysis did not converge, it needs to stop and produce a conservative solution. The fallback (always valid) is to create a recursive definition:

$$\hat{f}_\omega(i) = \text{if } (i \geq 0)\ \Delta(\hat{f}_\omega(i-1)) \text{ else } y_0$$

This solution is the last resort for our analysis. Therefore, we can view the function space as partially ordered, from optimistic to pessimistic: $\hat{f}_0 \sqsubset \hat{f}_1 \cdots \sqsubset \hat{f}_\omega$.

Here, $\hat{f}_0$ can be polynomials of degree 0, $\hat{f}_1$ polynomials of degree 1 etc., with the recursive form $\hat{f}_\omega$ at the top of the chain. While polynomials are useful, other chains of functions would be possible (e.g., Fourier series). The Kleene iteration is subject to the usual conditions, i.e., that sequences of functions $\hat{f}_k$ picked during iteration must be monotonic in k and without infinite chains.

Detecting Sequence Construction. Similar to the extraction of closed forms from scalar recurrences, we use speculative rewriting to extract collective forms for sequence construction. Consider the following program:

```
a := new;
j := 0;
while j < k do {
  a[j] := g(j);
  j := j + 1
}
```

Just like we speculate on a closed form for `j`, we recognize that the first loop iteration writes to index 0 in `a`, and we speculate that subsequent loop iterations will write to indexes 1, 2, etc. Hence, for the next Kleene iteration step we propose a collective form for `a`, and verify its validity in the next iteration. During this process, we notice that `j` is equal to the loop index, which means that `a` is being assigned at the loop index. Therefore we can assume that `a` is an array $\langle . \rangle (i_2 < i) .\ g(i_2)$ and continue the iteration process. As explained in Section 2, extracting collective forms for heap-allocated data structures is key for reasoning about programs like the one in Figure 1.

5 SCALING UP TO C

In the preceding sections, we have instantiated our approach for a representative model language `IMP`. To validate this model in practice, we have built a prototype tool called `SIGMA` that applies essentially the same approach to C code. Compared to the formal model, there are several challenges posed by a large and realistic language. Two important features that `IMP` does not include are functions, and intraprocedural control flow other than `if` and `while`. These include in particular `goto`, `break`, `continue`, `switch/case`, etc.

`SIGMA` uses the C parser from the Eclipse project to obtain an AST from C source. `SIGMA` then computes a control-flow graph for each function in the AST, and converts it back into a structured loop form using standard algorithms [Ramshaw 1988; Yakdan et al. 2015]. We chose this approach for its relative simplicity and consistency with the formal description. It would also be possible to adapt the fixpoint algorithm from Section 4 to work directly on control-flow graphs. As part of the iterative translation to a slightly extended `FUN` language, `SIGMA` resolves function calls and inlines the function body at the call site, which provides a level of context-sensitivity. A potentially more scalable and performant alternative would be to compute `FUN` summaries for each function separately, leading to a more modular analysis approach. `SIGMA` currently does not support recursive functions at the C source level, beyond inlining them up to a variable cutoff.

Our simplification approach is based (1) on normalizing rewrites using smart constructors, e.g. pushing a constant in a product to the right, and (2) on an explicit simplification procedure. The main ingredient here is a solver for linear inequalities over integers, which in our case consists of a custom implementation of the Omega test [Pugh 1991]. Other algorithms would also be feasible [Dillig et al. 2011a]. Instead of this integrated implementation one could also consider invoking an external SMT solver. However, care must be taken to faithfully encode `FUN` terms, since the current generation of SMT solvers cannot directly represent collective operators.

Another feature that is required for realistic analysis is dealing with nondeterministic input, often called `havoc` or `rand?`. `SIGMA` models this in a manner very similar to dynamic allocations: each call to `rand?` is parameterized with the program context `rand?(path)`, so that the results of different `rand?` calls can be uniquely identified even on the symbolic level. We use this in Figures 10, 11, 12.

While Section 3 has focused on a formal soundness property for `IMP`, we do not make such claims for the full C language. In particular, `SIGMA` does not accurately model integer overflow, pointer arithmetic (beyond arrays), floating point computation, concurrency, and undefined behavior.

Analysis and verification of C code using `SIGMA` can currently only be considered sound for programs that do not use such features. These restrictions are not unreasonable, and are, for example, reflected in certain categories of the SV-COMP verification benchmarks.

Figure 10 and 11 illustrate complex control flow within loops. In Figure 10, `SIGMA` manages to infer the polynomial form of the variable `agg` during the approximation phase. However, in Figure 11, there is no such polynomial form, thus `SIGMA` generates a generic sum for variable `a` and `b`. While this generic form does not provide a lot of information about either `a` or `b`, it can be used to prove, through the algebraic properties of the sum, that the condition `a + b == 3*n` is always evaluated to true.

```
int main() {
  // path p1
  int n = __VERIFIER_nondet_int();
  // path p2
  int m = __VERIFIER_nondet_int();
  int agg = 0;  int i = 0;
  __VERIFIER_assume(0 < m && 0 <= n);
  while (i < n) {
    // path p3(x18?) = <...>.while[x18?]
    if (i < m) agg += 3;
    else agg += 1;
    i += 1;
  }
  return 0;
}
```

```
Store σ_{x18?} after iteration x18? (constant values elided):
"&i" ↦ [ (x18? + 1) : "int" ]
"&agg" ↦ [ if ((x18? < rand?(p2))) { ((x18? * 3) + 3) }
           else { ((rand?(p2) * 2) + (x18? + 1)) } : "int" ]
Loop termination: u = #(x18?).!(x18? < rand?(p1))
Final store σ_f = σ_{u-1} =
"&n" ↦ [ rand?(p1) : "int" ]
"&m" ↦ [ rand?(p2) : "int" ]
"&i" ↦ [ rand?(p1) : "int" ]
"&agg" ↦ [ if ((rand?(p1) < (rand?(p2) + 1))) { rand?(p1) * 3 }
           else { ((rand?(p1) * 2) + rand?(p1)) } : "int" ]
"return" ↦ [ 0 : "int" ]
```

Fig. 10. SIGMA analysis result for a program with a conditional in a loop where the result can be expressed as a polynomial. Left: C source code. Right: the store inferred within the loop and the final store.

```
int main() {
  int i, n, a, b;
  i = 0; a = 0; b = 0;
  // path p1
  n = __VERIFIER_nondet_int();
  __VERIFIER_assume(n >= 0 && n <= 1000000);
  while (i < n) {
    // path p2(x11?) = <...>.while[x11?]
    if (__VERIFIER_nondet_int()) {
      a = a + 1;  b = b + 2;
    } else { a = a + 2;  b = b + 1; }
    i = i + 1;
  }
  __VERIFIER_assert(a + b == 3*n);
  return 0;
}
```

```
Final store:
"&i" ↦ [ rand?(p1) : "int" ]
"valid" ↦ 1
"&a" ↦ [ sum(rand?(p1)) { x11? =>
           if (rand?(p2(x11?))) 1 else 2
         } : "int" ]
"&n" ↦ [ rand?(p1) : "int" ]
"&b" ↦ [ sum(rand?(p1)) { x11? =>
           if (rand?(p2(x11?))) 2 else 1
         } : "int" ]
"return" ↦ [ 0 : "int" ]
```

Fig. 11. SIGMA analysis result for a program with a conditional in a loop where the result can not be expressed as a polynomial (sv-comp benchmark loop-lit/bhmr2007_true-unreach-call.c.i). We show the C source on the left. On the right, we show the final store.

```
int main() {
  // path p1
  int n = __VERIFIER_nondet_int();
  int agg = 0;
  int i = 0;
  __VERIFIER_assume(0 <= n);
  while (i < n) {
    agg += i; i += 4;
  }
  return 0;
}
```

```
Store σ_{x11?} after iteration x11? (constant values elided):
"&agg" ↦ [ (x11? * (x11? * 2)) + (x11? * 2)) : "int" ]
"&i" ↦ [ (x11? * 4) + 4: "int" ]
Loop termination: u = #(x11?).!((x11? * 4) < rand?(p1))
Final store σ_f = σ_{u-1} =
"&n" ↦ [ rand?(p1) : "int" ]
"&i" ↦ [ ((rand?(p1) + 3) / 4) * 4 : "int" ]
"&agg" ↦ [ (rand?(p1) + 3) / 4) * (((rand?(p1) + 3) / 4) * 2)
           + ((rand?(p1)+3)/4)* -2 : "int" ]
"return" ↦ [ 0 : "int" ]
```

Fig. 12. SIGMA analysis result for a program with a loop increment different from 1. We show the C source on the left. On the right, we show the store inferred within the loop and the final store.

6 EVALUATION

We evaluate SIGMA in three different categories: program verification, program equivalence, and transformation of legacy code to DSLs. We use an Intel Core i7-7700 CPU with 32GB of RAM running Ubuntu 16.04.3 LTS.

Verification. We compare SIGMA with CPAchecker [Beyer and Keremoglu 2011] and SeaHorn [Gurfinkel et al. 2015] on programs from or similar to the SV-COMP benchmarks [Beyer 2012]. We used the programs from the *loop-lit*, *loop-invgen*, and *recursive-simple-*∗ categories of SV-COMP. CPAchecker won the 2018 SV-COMP competition, and both state-of-the-art tools scored highly in previous years. The goal is to assess the reachability of a given function call *__VERIFIER_error()* (an assertion evaluated to false triggers a call to this function as well). The expected result of the analysis is encoded in the filename, e.g., *false-unreach-call* means that the call marked unreachable can actually be executed, whereas a *true-unreach-call* means that the error can never be triggered. The example in Figure 11 is a program from the *loop-lit* category.

Name	CPAchecker	SeaHorn	SIGMA
simple_built_from_end_true-unreach-call.i	TIMEOUT	250	273
list_addnat_false-unreach-call.i	2890	190	215
list_addnat_true-unreach-call.i	305560	170	215
loop_addnat_false-unreach-call.i	2830	190	285
loop_addnat_true-unreach-call.i	TIMEOUT	200	285
loop_addsubnat_false-unreach-call.i	3140	210	364
loop_addsubnat_true-unreach-call.i	TIMEOUT	230	364
nestedloop_mul1_true-unreach-call.i	OUT OF MEMORY	7280	405
nestedloop_mul2_true-unreach-call.i	TIMEOUT	240	365

Fig. 13. Results are in ms (TIMEOUT is set at 900s). The red cells indicate incorrect results (false positives).

First, we highlight nine challenging programs of our benchmark: three programs operate on singly linked lists, four programs use more than one non-nested loop and two other programs have nested loops. The results for these programs are shown in Figure 13. At first glance we can see two distinct behaviors between CPAchecker and SeaHorn. CPAchecker, while being quite slow, never gives an incorrect answer. SeaHorn, on the other hand, is fast and can sometimes give an incorrect (false positive) result. All three tools manage to handle the *false-unreachable-call* case, which can be seen as the easy problem as it only requires to find a counterexample. However in the case of true-unreachable, the prover needs to check all possible values. The very big search space explains CPAchecker's timeouts. For SeaHorn, the true case appears to be difficult, as the internal logic may overapproximate the problem and give an incorrect (false positive) result. SIGMA, by contrast, is very precise and can verify all the examples while being almost as fast as SeaHorn.

In order to have a more general idea of the performance of SIGMA, we look at three SV-COMP suites: *loop-lit*, *loop-invgen*, and *recursive-simple-*∗. Figure 14 shows a summary view of the running time of the analysis, and compares SIGMA to CPAChecker and SeaHorn. In these graphs, inspired by a similar visualization by [Zhu et al. 2018], each dot represents one program. The dots are placed at coordinates representing their analysis times. The samples under the identity line are the programs where SIGMA is faster then the other tool. The complete list of programs is shown in Figure 15. On the *loop-** benchmark, which is composed of programs with loops and computation over scalar values, SIGMA performs very well. It proves most of the reachability goals quickly. Out of the 37 programs, it times out on 3 programs that have complex control flow (jumps from within the then branch to the else branch). There are also 4 programs where SIGMA cannot make a decision; this is when the valid flag is not simplified to 0 or 1. This situation is indicated by a yellow box in Figure 15. For the *recursive-simple-*∗ benchmark, even though SIGMA is not designed to handle recursion, on

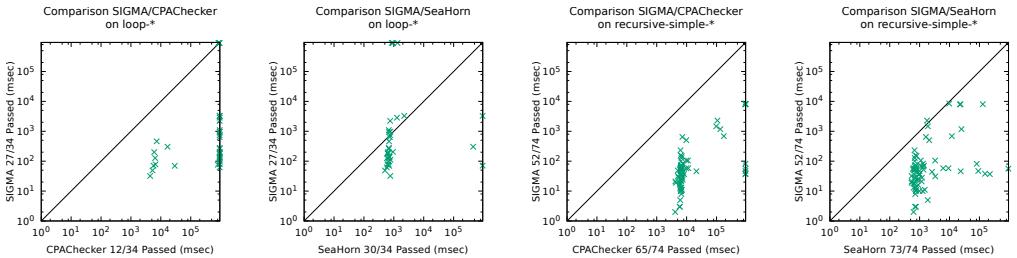

Fig. 14. Verification time of CPAChecker vs SIGMA, and SeaHorn vs SIGMA. Each point represents a program and is placed at coordinates (analysis time CPACHecker/SeaHorn, analysis time SIGMA).

many of the programs, the verification is successful. SIGMA manages to verify the assertions through pure symbolic execution driven by inlining of function calls. We did not implement any additional analysis for recursive functions. For the programs where it fails, the execution dependents on an unknown value thus a simple symbolic execution can not terminate and SIGMA gives up after exhausting its internal inlining limit.

Program Equivalence. Since SIGMA computes exact symbolic descriptions of the program state post-execution, it can also be used to test program equivalence. In the absence of side effects that read external input, we can run SIGMA on both programs and verify that the post-execution states have the same symbolic representation, up to symbolic rewriting. Let s_1 and s_2 be the two symbolic states, then we want to test whether $s_1 = s_2$ simplifies to true. Since errors are explicit at the symbolic level, this test not only applies to programs that independently verify, but can also relate the error behavior of two programs. In our evaluation, we manage to prove equivalence between the two functions below: one using a while loop, and the other one using GOTOs. For example, given the same random input, these two code snippets produce the same symbolic forms for the return value:

```
int main() {
  int a = nondet_int();
  int b = 0;
  while (b < a)
    b = b + 1;
  return b;
}
```

```
int main() {
  int a = nondet_int();
  int b = 0; goto cond;
more: b = 1 + b;
cond: if (b < a) goto more;
  return b;
}
```

This method can be generalized to global variables and even I/O. In the case of I/O, the different streams of input data can be modeled as data stored on the heap (see earlier discussion of `rand?(p)`, and the symbolic forms need to match to prove equivalence.

In addition, we have verified that SIGMA can demonstrate equivalence of programs in the style of Section 2: (1) a program that computes the sum of n nondeterministic inputs in a loop; (2) a program that stores these value in an array and then computes the sum; (3) a program that stores these values in a linked list and then computes the sum.

Legacy Code to DSLs. SIGMA can also be used to analyze legacy optimized C code and translate it to high-level performance-oriented DSLs. In addition, the symbolic representation obtained from the analysis can be used to better understand the program behavior. An interesting case is Stencil codes, which are patterns for updating array elements according to their neighbors. Many stencil codes are written in Fortran or C and heavily optimized for performance on a particular architecture, which precludes porting to GPUs, parallelizing and distributing across a cluster, or in general foward-porting the code to other emerging architectures.

Name	CPAchecker	SeaHorn	SIGMA
linv/apache-escape-[...]_true-unreach-call.i	848638	1329	TIMEOUT
linv/apache-get-tag_true-unreach-call.i	TIMEOUT	851	TIMEOUT
linv/down_true-unreach-call.i	TIMEOUT	650	98
linv/fragtest_simple_true-unreach-call.i	TIMEOUT	766	978
linv/half_2_true-unreach-call.i	TIMEOUT	717	142
linv/heapsort_true-unreach-call.i	TIMEOUT	2237	3275
linv/id_build_true-unreach-call.i	6632	661	78
linv/id_trans_false-unreach-call.i	7268	692	462
linv/large_const_true-unreach-call.i	16748	448736	305
linv/MADWiFi-[...]_true-unreach-call.i	TIMEOUT	754	2232
linv/nest-if3_true-unreach-call.i	TIMEOUT	746	740
linv/nested6_true-unreach-call.i	TIMEOUT	1272	2853
linv/nested9_true-unreach-call.i	TIMEOUT	TIMEOUT	3267
linv/NetBSD_loop_true-unreach-call.i	TIMEOUT	631	177
linv/sendmail-[...]_true-unreach-call.i	TIMEOUT	680	279
linv/seq_true-unreach-call.i	TIMEOUT	681	1101
linv/SpamAssassin-loop_true-unreach-call.i	867493	923	TIMEOUT
linv/string_concat-noarr_true-unreach-call.i	TIMEOUT	732	636
linv/up_true-unreach-call.i	TIMEOUT	621	183
llit/afnp2014_true-unreach-call.c.i	29254	700	70
llit/bhmr2007_true-unreach-call.c.i	TIMEOUT	769	112
llit/cggmp2005_true-unreach-call.c.i	4391	761	32
llit/cggmp2005_variant_true-unreach-call.c.i	TIMEOUT	583	62
llit/cggmp2005b_true-unreach-call.c.i	5998	628	201
llit/css2003_true-unreach-call.c.i	6469	664	127
llit/ddlm2013_true-unreach-call.c.i	TIMEOUT	934	205
llit/gj2007_true-unreach-call.c.i	5244	504	49
llit/gj2007b_true-unreach-call.c.i	TIMEOUT	714	136
llit/gr2006_true-unreach-call.c.i	5692	TIMEOUT	71
llit/gsv2008_true-unreach-call.c.i	TIMEOUT	679	952
llit/hhk2008_true-unreach-call.c.i	TIMEOUT	620	79
llit/jm2006_true-unreach-call.c.i	802315	625	76
llit/jm2006_variant_true-unreach-call.c.i	TIMEOUT	590	126
llit/mcmillan2006_true-unreach-call.c.i	TIMEOUT	683	259
rec/afterrec_2calls_false-unreach-call.c	5296	673	29
rec/afterrec_2calls_true-unreach-call.c	4377	534	19
rec/afterrec_false-unreach-call.c	5041	659	49
rec/afterrec_true-unreach-call.c	4151	629	2
rec/fibo_10_false-unreach-call.c	9751	2100	504
rec/fibo_10_true-unreach-call.c	9716	9217	58
rec/fibo_15_false-unreach-call.c	107274	1772	2292
rec/fibo_15_true-unreach-call.c	134059	25115	1182
rec/fibo_20_false-unreach-call.c	TIMEOUT	22148	7934
rec/fibo_20_true-unreach-call.c	TIMEOUT	129764	8049
rec/fibo_25_false-unreach-call.c	TIMEOUT	81518	82
rec/fibo_25_true-unreach-call.c	TIMEOUT	223645	37
rec/fibo_2calls_10_false-unreach-call.c	9731	1096	106
rec/fibo_2calls_10_true-unreach-call.c	11053	3298	105
rec/fibo_2calls_15_false-unreach-call.c	97812	1887	1466
rec/fibo_2calls_15_true-unreach-call.c	175109	11984	686
rec/fibo_2calls_2_false-unreach-call.c	5143	538	32
rec/fibo_2calls_2_true-unreach-call.c	4213	549	21
rec/fibo_2calls_20_false-unreach-call.c	TIMEOUT	9728	8441
rec/fibo_2calls_20_true-unreach-call.c	TIMEOUT	23486	8053
rec/fibo_2calls_25_false-unreach-call.c	TIMEOUT	91274	47
rec/fibo_2calls_25_true-unreach-call.c	TIMEOUT	154521	38
rec/fibo_2calls_4_false-unreach-call.c	6500	763	46
rec/fibo_2calls_4_true-unreach-call.c	6322	980	10
rec/fibo_2calls_5_false-unreach-call.c	6751	790	57
rec/fibo_2calls_5_true-unreach-call.c	6531	1265	10
rec/fibo_2calls_6_false-unreach-call.c	7328	883	24
rec/fibo_2calls_6_true-unreach-call.c	6679	1349	52
rec/fibo_2calls_8_false-unreach-call.c	8138	946	37
rec/fibo_2calls_8_true-unreach-call.c	7730	2611	44
rec/fibo_5_false-unreach-call.c	6557	795	60
rec/fibo_5_true-unreach-call.c	6134	1873	5
rec/fibo_7_false-unreach-call.c	7209	1190	28
rec/fibo_7_true-unreach-call.c	6745	3332	31
rec/id_b2_o3_true-unreach-call.c	6330	705	65
rec/id_b3_o2_false-unreach-call.c	6915	790	149
rec/id_b3_o5_true-unreach-call.c	6076	668	51
rec/id_b5_o10_true-unreach-call.c	6223	596	53
rec/id_i10_o10_false-unreach-call.c	6513	956	35
rec/id_i10_o10_true-unreach-call.c	6067	678	18
rec/id_i15_o15_false-unreach-call.c	6640	1169	21
rec/id_i15_o15_true-unreach-call.c	6118	631	13
rec/id_i20_o20_false-unreach-call.c	6827	1306	66
rec/id_i20_o20_true-unreach-call.c	6197	624	18
rec/id_i25_o25_false-unreach-call.c	6728	1484	11
rec/id_i25_o25_true-unreach-call.c	6540	681	35
rec/id_i5_o5_false-unreach-call.c	6327	817	24
rec/id_i5_o5_true-unreach-call.c	5832	706	3
rec/id_o10_false-unreach-call.c	7026	996	38
rec/id_o100_false-unreach-call.c	11357	6152	58
rec/id_o1000_false-unreach-call.c	TIMEOUT	TIMEOUT	56
rec/id_o20_false-unreach-call.c	7386	1305	83
rec/id_o200_false-unreach-call.c	21269	23597	46
rec/id_o3_false-unreach-call.c	6454	777	69
rec/id2_b2_o3_true-unreach-call.c	6274	739	55
rec/id2_b3_o2_false-unreach-call.c	6476	723	74
rec/id2_b3_o5_true-unreach-call.c	6169	670	57
rec/id2_b5_o10_true-unreach-call.c	6313	619	59
rec/id2_i5_o5_false-unreach-call.c	6384	731	23
rec/id2_i5_o5_true-unreach-call.c	6038	895	16
rec/sum_10x0_false-unreach-call.c	6437	985	47
rec/sum_10x0_true-unreach-call.c	6678	754	13
rec/sum_15x0_false-unreach-call.c	6805	1125	68
rec/sum_15x0_true-unreach-call.c	6457	745	10
rec/sum_20x0_false-unreach-call.c	6859	1393	71
rec/sum_20x0_true-unreach-call.c	6693	716	8
rec/sum_25x0_false-unreach-call.c	7189	1625	648
rec/sum_25x0_true-unreach-call.c	6853	760	11
rec/sum_2x3_false-unreach-call.c	6182	717	12
rec/sum_2x3_true-unreach-call.c	5959	755	3
rec/sum_non_eq_false-unreach-call.c	6739	687	127
rec/sum_non_eq_true-unreach-call.c	6480	685	100
rec/sum_non_false-unreach-call.c	6228	675	232
rec/sum_non_true-unreach-call.c	6352	727	157

Fig. 15. Results are in ms (TIMEOUT is set at 900s). The red cells indicate incorrect results (false positives), and the yellow cells indicate when SIGMA could not decide (valid flag was not simplified to 0 or 1).

A simple example is Jacobi iteration on a one-dimensional array. At each iteration, the algorithm updates each location with the arithmetic mean of its left-hand side and right-hand side values. For the out-of-bound locations, the default value is 1. From the C program on the left, SIGMA will generate the closed form on the right for the computation of a single iteration:

```
int i = 0; int ai = a[0];
a[0] = (1 + a[1])/2;
while (i < n-1) {
  int tmp = (ai + a[i+1])/2;
  ai = a[i]; a[i++] = tmp;
}
a[n-1] = (ai + 1)/2;
```

```
// Extracted FUN code of the
// corresponding C code:
let a = ⟨.⟩(i < n).
  if (i == 0) then (1 + a[i+1])/2
  else if (i == n-1) then (a[i-1] + 1)/2
  else (a[i-1] + a[i+1])/2
```

From the derived closed form, the Jacobi algorithm is immediately apparent, despite the use of temporaries and loop-carried dependencies in the C source. The closed form FUN code is easily

mapped to a high-performance DSL such as Halide [Kamil et al. 2016; Mendis et al. 2015], which supports parallel CPU, GPU, and cluster execution. The process readily generalizes to multi-dimensional Jacobi iteration.

7 RELATED WORK

Decompiling to High-Level Languages. A key ingredient of our approach is to transform—in a sense, "decompile"—a low-level language into a comparatively higher level language. Our approach has been greatly inspired by previous work in this direction. Some classics are the GOTO-elimination algorithm by Ramshaw [1988] (newer work in this direction includes [Yakdan et al. 2015]), and the realization that compilers or analyzers for imperative languages based on SSA form are essentially using a functional intermediate language [Appel 1998]. Various compiler frameworks (e.g., [Bergstra et al. 1996]) use rewriting rules to drive simplification and analysis, though these approaches typically do not address challenges such as those introduced by dynamic allocation.

More recently, there has been a flurry of work that aims to translate low-level imperative code to high-performance DSLs. Some works are based on a technique described as verified lifting, which is used to transform stencil codes to the Halide DSL [Kamil et al. 2016; Mendis et al. 2015], or to transform imperative Java code to Hadoop for cluster execution [Ahmad and Cheung 2016]. Another line of work uses symbolic execution to parallelize user-defined aggregations [Raychev et al. 2015]. An approach closely related to ours transforms Java code to a functional IR and then to Apache Spark, after a rewriting and simplification process that, e.g., maps loop-carried dependencies to group-by operations [Radoi et al. 2014]. There is also work on synthesizing MapReduce programs from sketches [Smith and Albarghouthi 2016], on defining language subsets that are guaranteed to have an efficient translation [Rompf and Brown 2017], and work in the space of just-in-time compilers to reverse-engineer Java bytecode at runtime and redirect imperative API calls to embedded DSLs [Rompf et al. 2014].

High-Performance DSLs. Some notable works in the DSL space include Delite [Brown et al. 2011; Lee et al. 2011; Rompf et al. 2011; Sujeeth et al. 2011], Halide [Ragan-Kelley et al. 2013], and Accelerate [Chakravarty et al. 2011; McDonell et al. 2015; Svensson et al. 2014, 2015; Vollmer et al. 2015]. Most of these systems come with expressive, functional IRs. Some systems focus explicitly on the intermediate layers, for example Lift [Steuwer et al. 2015, 2017], PENCIL [Baghdadi et al. 2015], or the parallel action language [Llopard et al. 2017].

Analysis and Optimization. Our optimistic fixpoint approach is directly inspired by Lerner et al.'s work on composing dataflow analyses and transformations [Lerner et al. 2002]. Related work has aimed to automatically prove the correctness of compiler optimizations [Lerner et al. 2003], and on generating compiler optimizations from proofs of equivalence and before/after examples [Tate et al. 2010]. A related line of work models a space of possible program transformations given by equivalence rules through the notion of equality saturation, based on a program equivalence graphs (PEGs) [Tate et al. 2011] as IR. The PEG model has heavily inspired early versions of our work. The reasoning-by-rewriting approach and the avoidance of phase-ordering issues is similar, as is the overall goal of a flexible semantics-preserving representation as a basis for various kinds of analysis. However there are important differences: PEGs do not include collective forms except the `pass` operator, which is similar to our #. The θ operator in the PEG model describes standard recurrences, not collective forms. Tate et al. [2011] also do not discuss specifics about heap-allocated data, and the accompanying Java analysis tool Peggy maps all heap objects into a single summarization object. Thus, while PEGs can express program equivalence in general, Peggy could not prove the equivalences in Section 6, nor verify the linked list program in Figure 1. The two innovations we

propose, collective forms and structured heaps, could be implemented without difficulty in a PEG setting, and potentially improve precision.

Recurrence Analysis. Analyzing integer recurrences has been an active topic of research. Some recent works include compositional recurrence analysis (CRA) [Farzan and Kincaid 2015; Kincaid et al. 2017], which aims to derive closed forms for recurrence equations and inequations. The approach is based on an algebraic representation of path expressions [Tarjan 1981], refered to as Newtonian program analysis [Reps et al. 2016]. Earlier works include abstract acceleration of general linear loops [Jeannet et al. 2014], and a study of algebraic reasoning about P-solvable loops [Kovács 2008]. Efficient integer linear inequality solvers have been available for some time [Dillig et al. 2011a; Pugh 1991]. Aligators [Henzinger et al. 2010] is a tool from the static analysis community, representative for highlighting some of the limitations. Given a simple loop as input, Aligators can extract quantified scalar invariants, using a recurrence solving technique similar to the one used in our framework. However, like many other tools, Aligators has limited applicability in that it only handles linear recurrences (polynomials of degree 1), does not handle nested loops, does not provide collective forms such as symbolic sums, does not handle dynamic allocation of arrays, and does not appear to support complex or nested conditions inside loops. Many compilers provide some form of recurrence analysis as part of their optimization suite, often based on Bachmann et al. [1994]'s chains of recurrences (CoR) model, and sometimes called *scalar evolution*. An example is the SCEV pass in LLVM. These analyses are able to infer closed-form representations for simple counting loops but are limited in ways similar to tools like Aligators with respect to dynamic allocations, collective forms, and complex expressions.

Heap Abstraction. Recent work on efficient and precise points-to analysis models the heap by merging equivalent automata [Tan et al. 2017]. Other works use structured heaps to model container data structures [Dillig et al. 2011b], and some techniques have been proposed for heap abstractions that enable sparse global analyses for C-like languages [Oh et al. 2012], similar in spirit to SSA form. While SSA is typically used for local variables, techniques under the umbrella name Array SSA exist to extend sparse reasoning to heap data [Knobe and Sarkar 1998]. Our simplificaton rules that break apart heap objects to expose their fields are inspired by such techniques. Abstracting abstract machines [Horn and Might 2010] described different kinds of allocation policies parameterized by an abstract clock. This line of work has been inspirational for our structured heap representation, which differs in modeling the heap structure after the syntactic structure of the program. Many other directions exist, e.g., predicate abstraction for heap manipulation programs [Bingham and Rakamaric 2006].

Shape analysis [Sagiv et al. 2002] provides a parametric framework for specifying different abstract interpretations. In each instantiation of the framework, a set of possible runtime stores is represented by a set of 3-valued logical structures. An individual in a 3-valued structure represents a set of runtime objects: each individual represents all objects in a runtime store that have the same values for a chosen set of properties of objects. (Different instantiations of the framework are created by making different choices of which object-properties to use.) A 3-valued structure does not represent a static partition of the runtime objects; for instance, in a loop the properties of a given object o can change from iteration to iteration, and hence the individual that represents o would be different on different iterations. Stated another way, a given individual in a 3-valued structure can represent different objects when considered to be the abstraction of the runtime stores that arise on different iterations. Our paper takes a different approach, by indexing objects via an abstract notion of time: all objects allocated in a loop are considered to be a sequence (i.e., a collective form) indexed by the (symbolic) loop variable. It remains to be seen whether the two approaches could be combined, and what the advantages of such a combination might be.

Shape analysis approaches based on separation logic [Brookes and O'Hearn 2016; Reynolds 2002] improve precision and scale to large codebases [Calcagno et al. 2011; Distefano et al. 2006], implemented, e.g., in Facebook's Infer tool [Calcagno et al. 2015]. With its support for reasoning about linked list and related structures via bi-abduction [Calcagno et al. 2011], Infer should in principle come close to verifying programs like the one in Figure 1; however it still fails on this particular example and several variations we tried on the public Infer web interface. Since Infer does not compute precise symbolic representations, however, it is unsuited for tasks like translating legacy code to DSLs (Section 6). An interesting avenue for future research is how our heap representation can form a basis for and interact with separation predicates. This could, e.g., enable support for modular analyses that use a precise partial heap model within a function, and approximate separation predicates for function contracts.

Gopan et al. [2005] extend the ideas from shape analysis à la Sagiv et al. [2002] to indexed elements in arrays, thereby creating a parametric framework for creating abstract interpreters that can establish certain kinds of relationships among array elements. Their approach is based on splitting the collection of array elements into partitions based on index values that satisfy common properties, e.g., $< i$, $= i$, or $> i$, where i is a loop-counter variable. Additional predicates are introduced to hold onto invariants of elements that have been coalesced into a single partition. Instantiations of the framework are capable of establishing that (i) an array-initialization loop initializes all elements of an array (and that certain numeric constraints hold on the values of the initialized elements); (ii) an array-copy loop copies all elements from one array to another; and (iii) an insertion-sort routine sorts all of the elements of an array.

The idea of representing program values in terms of an execution context that captures the current loop iteration is also present in previous work on dynamic program analysis [Xin et al. 2008] and on polyhedral compilation [Benabderrahmane et al. 2010]. The main difference in our work is that we push the indexing idea all the way into the store model and allocation scheme, which permits effective static reasoning about dynamic allocations and linked data structures, and that we use the indexing scheme as a basis for a generic symbolic representation and static analysis.

Semantics. The proofs and formal models presented in this paper are largely standard, but make key use of induction over a numeric "fuel metric" that bounds the amount of work (in this case, loop iterations) a program is allowed to do. Such techniques enable effective proofs for functional formulations of big-step semantics and have only recently received wide-spread interest [Amin and Rompf 2017; Owens et al. 2016]. Existentially quantifying over the number of loop iterations in our `IMP` semantics is very similar to a recent proposal by Siek [2016, 2017].

8 CONCLUSIONS

In this paper, we identified two key limitations of current progam analysis techniques: (1) the low-level and inherently *scalar* description of program entities, and (2) collapsing information per program point, and projecting away the dimension of time. As a remedy, we proposed first-class collective operations, and a novel structured heap abstraction that preserves a symbolic dimension of time. We have elaborated both in a sound formal model, and in a prototype tool that analyzes C code. The paper includes an experimental evaluation that demonstrates competitive results on a series of benchmarks. Given its semantics-preserving nature, our implementation is not limited to analysis for verification, but our benchmarks also include checking program equivalence, and translating legacy C code to high-performance DSLs.

ACKNOWLEDGMENTS

The authors thank Suresh Jagannathan for providing extensive comments on draft versions of this paper. This work was supported in part by NSF awards 1553471, 1564207, 1918483, DOE award DE-SC0018050, as well as gifts from Google, Facebook, and VMware.

REFERENCES

Maaz Bin Safeer Ahmad and Alvin Cheung. 2016. Leveraging Parallel Data Processing Frameworks with Verified Lifting. In *SYNT/CAV (EPTCS)*, Vol. 229. 67–83.

Nada Amin and Tiark Rompf. 2017. Type soundness proofs with definitional interpreters. In *POPL*. ACM, 666–679.

Andrew W. Appel. 1998. SSA is Functional Programming. *SIGPLAN Notices* 33, 4 (1998), 17–20.

Olaf Bachmann, Paul S. Wang, and Eugene V. Zima. 1994. Chains of Recurrences - a Method to Expedite the Evaluation of Closed-form Functions. In *ISSAC*. ACM, 242–249.

Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F. Donaldson, Jeroen Ketema, Javed Absar, Sven van Haastregt, Alexey Kravets, Anton Lokhmotov, Robert David, and Elnar Hajiyev. 2015. PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming. In *PACT*. IEEE Computer Society, 138–149.

Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. 2010. The Polyhedral Model Is More Widely Applicable Than You Think. In *CC (Lecture Notes in Computer Science)*, Vol. 6011. Springer, 283–303.

Jan A. Bergstra, T. B. Dinesh, John Field, and Jan Heering. 1996. A Complete Transformational Toolkit for Compilers. In *ESOP (Lecture Notes in Computer Science)*, Vol. 1058. Springer, 92–107.

Dirk Beyer. 2012. Competition on Software Verification - (SV-COMP). In *TACAS (Lecture Notes in Computer Science)*, Vol. 7214. Springer, 504–524.

Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *CAV (Lecture Notes in Computer Science)*, Vol. 6806. Springer, 184–190.

Jesse D. Bingham and Zvonimir Rakamaric. 2006. A Logic and Decision Procedure for Predicate Abstraction of Heap-Manipulating Programs. In *VMCAI (Lecture Notes in Computer Science)*, Vol. 3855. Springer, 207–221.

Stephen Brookes and Peter W. O'Hearn. 2016. Concurrent separation logic. *SIGLOG News* 3, 3 (2016), 47–65.

Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Arvind K. Sujeeth, Christopher De Sa, Christopher R. Aberger, and Kunle Olukotun. 2016. Have abstraction and eat performance, too: optimized heterogeneous computing with parallel patterns. In *CGO*. ACM, 194–205.

Kevin J. Brown, Arvind K. Sujeeth, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2011. A Heterogeneous Parallel Framework for Domain-Specific Languages. *20th International Conference on Parallel Architectures and Compilation Techniques*.

Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NFM (Lecture Notes in Computer Science)*, Vol. 9058. Springer, 3–11.

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6 (2011), 26:1–26:66.

Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *DAMP*. ACM, 3–14.

Edsger Wybe Dijkstra. 1976. *A Discipline of Programming* (1st ed.). Prentice Hall PTR, Upper Saddle River, NJ, USA.

Isil Dillig, Thomas Dillig, and Alex Aiken. 2011a. Cuts from proofs: a complete and practical technique for solving linear inequalities over integers. *Formal Methods in System Design* 39, 3 (2011), 246–260.

Isil Dillig, Thomas Dillig, and Alex Aiken. 2011b. Precise reasoning for programs using containers. In *POPL*. ACM, 187–200.

Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2006. A Local Shape Analysis Based on Separation Logic. In *TACAS (Lecture Notes in Computer Science)*, Vol. 3920. Springer, 287–302.

Robert A. Van Engelen, Johnnie Birch, Yixin Shou, Burt Walsh, and Kyle A. Gallivan. 2004. A unified framework for nonlinear dependence testing and symbolic analysis. In *ICS*. ACM, 106–115.

Azadeh Farzan and Zachary Kincaid. 2015. Compositional Recurrence Analysis. In *FMCAD*. IEEE, 57–64.

Joseph Fourier. 1820. Extrait d'une mémoire sur le refroidissement séculaire du globe terrestre. *Bulletin des Sciences par la Société Philomathique de Paris, April 1820* (1820), 58–70.

Denis Gopan, Thomas W. Reps, and Shmuel Sagiv. 2005. A framework for numeric analysis of array operations. In *POPL*. ACM, 338–350.

Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *CAV (1) (Lecture Notes in Computer Science)*, Vol. 9206. Springer, 343–361.

Thomas A. Henzinger, Thibaud Hottelier, Laura Kovács, and Andrey Rybalchenko. 2010. Aligators for Arrays (Tool Paper). In *LPAR (Yogyakarta) (Lecture Notes in Computer Science)*, Vol. 6397. Springer, 348–356.

David Van Horn and Matthew Might. 2010. Abstracting abstract machines. In *ICFP*. ACM, 51–62.

Kenneth E. Iverson. 1980. Notation as a Tool of Thought. *Commun. ACM* 23, 8 (1980), 444–465.

Bertrand Jeannet, Peter Schrammel, and Sriram Sankaranarayanan. 2014. Abstract acceleration of general linear loops. In *POPL*. ACM, 529–540.

Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified lifting of stencil computations. In *PLDI*. ACM, 711–726.

Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas Reps. 2017. Compositional Recurrence Analysis Revisited. In *PLDI*.

Kathleen Knobe and Vivek Sarkar. 1998. Array SSA Form and Its Use in Parallelization. In *POPL*. ACM, 107–120.

Laura Kovács. 2008. Reasoning Algebraically About P-Solvable Loops. In *TACAS (Lecture Notes in Computer Science)*, Vol. 4963. Springer, 249–264.

HyoukJoong Lee, Kevin J. Brown, Arvind K. Sujeeth, Hassan Chafi, Tiark Rompf, Martin Odersky, and Kunle Olukotun. 2011. Implementing Domain-Specific Languages for Heterogeneous Parallel Computing. *IEEE Micro* 31, 5 (2011), 42–53.

Sorin Lerner, David Grove, and Craig Chambers. 2002. Composing dataflow analyses and transformations. In *POPL*. ACM, 270–282.

Sorin Lerner, Todd D. Millstein, and Craig Chambers. 2003. Automatically proving the correctness of compiler optimizations. In *PLDI*. 220–231.

Ivan Llopard, Christian Fabre, and Albert Cohen. 2017. From a Formalized Parallel Action Language to Its Efficient Code Generation. *ACM Trans. Embedded Comput. Syst.* 16, 2 (2017), 37:1–37:28.

Trevor L. McDonell, Manuel M. T. Chakravarty, Vinod Grover, and Ryan R. Newton. 2015. Type-safe runtime code generation: accelerate to LLVM. In *Haskell*. ACM, 201–212.

Charith Mendis, Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao, and Saman P. Amarasinghe. 2015. Helium: lifting high-performance stencil kernels from stripped x86 binaries to Halide DSL code. In *PLDI*. ACM, 391–402.

Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. 2012. Design and implementation of sparse global analyses for C-like languages. In *PLDI*. ACM, 229–238.

Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional Big-Step Semantics. In *ESOP (Lecture Notes in Computer Science)*, Vol. 9632. Springer, 589–615.

William Pugh. 1991. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *SC*. ACM, 4–13.

Cosmin Radoi, Stephen J. Fink, Rodric M. Rabbah, and Manu Sridharan. 2014. Translating imperative code to MapReduce. In *OOPSLA*. ACM, 909–927.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*. ACM, 519–530.

Lyle Ramshaw. 1988. Eliminating go to's while preserving program structure. *J. ACM* 35, 4 (1988), 893–920.

Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. 2015. Parallelizing user-defined aggregations using symbolic execution. In *SOSP*. ACM, 153–167.

Thomas W. Reps, Emma Turetsky, and Prathmesh Prabhu. 2016. Newtonian program analysis via tensor product. In *POPL*. ACM, 663–677.

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. IEEE Computer Society, 55–74.

Tiark Rompf and Kevin J. Brown. 2017. Functional parallels of sequential imperatives (short paper). In *PEPM*. ACM, 83–88.

Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. 2013. Optimizing Data Structures in High-Level Programs *(POPL)*.

Tiark Rompf, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, and Kunle Olukotun. 2014. Surgical precision JIT compilers. In *PLDI*. ACM, 41–52.

Tiark Rompf, Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2011. Building-Blocks for Performance Oriented DSLs. In *DSL (EPTCS)*, Vol. 66. 93–117.

Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. 2002. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.* 24, 3 (2002), 217–298.

Jack Schwartz. 1970. Set theory as a language for program specification and programming. *Courant Institute of Mathematical Sciences, New York University* 12 (1970), 193–208.

Jeremy G. Siek. 2016. Denotational Semantics of IMP without the Least Fixed Point. http://siek.blogspot.ch/2016/12/denotational-semantics-of-imp-without.html.

Jeremy G. Siek. 2017. Declarative semantics for functional languages: compositional, extensional, and elementary. *CoRR* abs/1707.03762 (2017). arXiv:1707.03762 http://arxiv.org/abs/1707.03762

Calvin Smith and Aws Albarghouthi. 2016. MapReduce program synthesis. In *PLDI*. ACM, 326–340.

Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code. In *ICFP*. ACM, 205–217.

Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: a functional data-parallel IR for high-performance GPU code generation. In *CGO*. ACM, 74–85.

Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Trans. Embedded Comput. Syst.* 13, 4s (2014), 134:1–134:25.

A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, Michael Wu, A. R. Atreya, M. Odersky, and K. Olukotun. 2011. OptiML: an Implicitly Parallel Domain-Specific Language for Machine Learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML)*.

Bo Joel Svensson, Mary Sheeran, and Ryan R. Newton. 2014. Design exploration through code-generating DSLs. *Commun. ACM* 57, 6 (2014), 56–63.

Bo Joel Svensson, Michael Vollmer, Eric Holk, Trevor L. McDonell, and Ryan R. Newton. 2015. Converting data-parallelism to task-parallelism by rewrites: purely functional programs across multiple GPUs. In *FHPC/ICFP*. ACM, 12–22.

Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and precise points-to analysis: modeling the heap by merging equivalent automata. In *PLDI*. ACM, 278–291.

Robert Endre Tarjan. 1981. Fast Algorithms for Solving Path Problems. *J. ACM* 28, 3 (1981), 594–614.

Ross Tate, Michael Stepp, and Sorin Lerner. 2010. Generating compiler optimizations from proofs. In *POPL*. 389–402.

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2011. Equality Saturation: A New Approach to Optimization. *Logical Methods in Computer Science* 7, 1 (2011).

Michael Vollmer, Bo Joel Svensson, Eric Holk, and Ryan R. Newton. 2015. Meta-programming and auto-tuning in the search for high performance GPU code. In *FHPC/ICFP*. ACM, 1–11.

Bin Xin, William N. Sumner, and Xiangyu Zhang. 2008. Efficient program execution indexing. In *PLDI*. ACM, 238–248.

Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. 2015. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantic-Preserving Transformations. In *NDSS*. The Internet Society.

He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A data-driven CHC solver. In *PLDI*. ACM, 707–721.