Precise Reasoning with Structured Heaps and Collective Operations à la Map/Reduce

What Static Analysis can Learn From High-Performance DSLs

Grégory Essertel, Guannan Wei, Tiark Rompf
Purdue University

Abstract

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 correlating multiple temporal points in the dynamic execution with a single point in the program text without ambiguity.

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.

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 [5] and SeaHorn [17], as well as Facebook's acclaimed Infer tool [10].

```
// 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)
```

Figure 1. Challenge program: in constrast to state-of-the-art tools like CPAchecker, SeaHorn, or Facebook 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 over this sequence $s = \Sigma(i < n).\ y[i].\texttt{head} = \Sigma(i < n).\ i$. 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).

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.

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 [16], big-Σ and related operators have rapidly become an integral part of modern mathematics. 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 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 pose all kinds of challenges including the need for automatic inference of invariants.

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. But over the last decade, a thriving line of research has demonstrated that the same is 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 [8, 9, 32, 38, 40, 45–48]. 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 [1, 20, 27, 31, 34, 37, 39].

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 Heap Abstraction 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

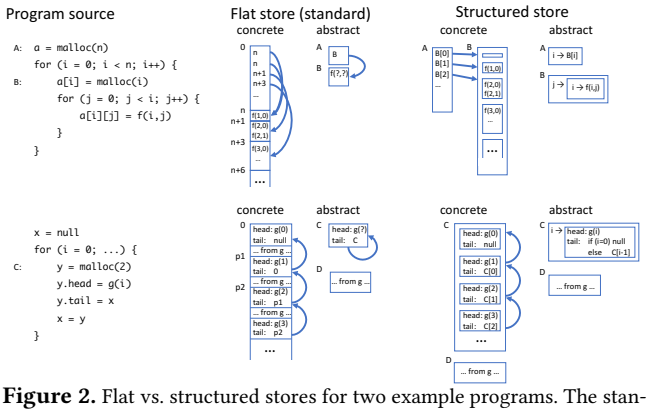

Figure 2. Flat vs. structured stores for two example programs. The standard flat store model assigns consecutive numeric addresses for each allocation. By contrast, addresses in our structured store model consist of a program point and the surrounding loop variables at the time of allocation. `A, B[1], 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.

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 summaries. Instead of abstracting dynamic allocations uniformly per program point, and having the abstract 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, similar to standard compilers. The solution is to *interleave* translation and simplification [25], 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 [1, 20, 27, 31, 34, 37, 39], but these systems (a) are specific to a given target DSLs, 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 [13, 51], 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 [55] and in polyhedral compilation [3]. 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).

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 [j ↦ ?, s ↦ ?], from 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: [j ↦$j(n-1)$, s ↦$s(n-1)$]

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 $0 \leq i < n$. Again, variable i is bound in the body of the term. As part of the simplification, β-reduction is performed for non-recursive functions:

```
let j = λ(i). if i ≥ 0 then j(i − 1) + 1 else 0 = λ(i). Σ(i₂ < i + 1). 1
let s = λ(i). if i ≥ 0 then s(i − 1) + j(i − 1) else 0
      = λ(i). Σ(i₃ < i + 1). j(i₃ − 1)
```

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

```
let j = λ(i). Σ(i₂ < i + 1). 1 = λ(i). if i ≥ 0 then i + 1 else 0
let s = λ(i). Σ(i₃ < i + 1). j(i₃ − 1) = λ(i). Σ(i₃ < i + 1). i₃
      = λ(i). if i ≥ 0 then (i + 1) ∗ i/2 else 0
let n = #(i). ¬(j(i) < k) = #(i). ¬(i + 1) < k
      = if k ≥ 0 then k else 0
```

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

```
[ j ↦ if k ≥ 0 then k else 0, s ↦ if k ≥ 0 then k∗(k−1)/2 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 sum
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 both 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 if 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 = λ(i). if i ≥ 0 then i + 1 else 0
let a = λ(i). if i ≥ 0 then a(i − 1)[i ↦ 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 = λ(i). if i ≥ 0 then a(i − 1)[i ↦ i] else []
      = λ(i). ⟨.⟩(i₂ < i + 1). i₂
```

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 = λ(i). if i ≥ 0 then i + 1 else 0
let s = λ(i). if i ≥ 0 then s(i − 1) + a[j(i − 1)] else 0
      = λ(i). if i ≥ 0 then s(i − 1) + (⟨.⟩(i₂ < n+1). i₂)[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 = λ(i). if i ≥ 0 then s(i − 1) + i else 0
      = λ(i). if i ≥ 0 then (i + 1) ∗ i/2 else 0
```

Finally the state at the end of the program is:

```
[ l ↦ n, j ↦ n, a ↦ ⟨.⟩(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        // traverse list, compute sum
x := null; l := 0                   z: = x; s: = 0
while l < n do {                    while z != null do {
  y := new; y.head := l;              s := s + z.head;
  y.tail := x; x := y; l := l + 1     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 y = λ(i). if i ≥ 0 then &new:p[i] else ⊥
let x = λ(i). if i ≥ 0 then &new:p[i] else null
```

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

Expressions $e \in \text{Exp}$

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

$n \mid b \mid \&x$	Constant (nat, bool, addr)
$e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2$	Arithmetic
$e_1 < e_2 \mid e_1 = e_2 \mid e_1 \wedge e_2 \mid \neg e$	Boolean
$e_1[e_2]$	Field read

Statements $s \in \text{Stm}$

$s ::=$

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

Syntactic Sugar

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

Figure 3. IMP: Surface language syntax.

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 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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 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).

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:

$m \in \text{Option } T ::= \text{None} \mid \text{Some } \tau \text{ where } \tau \in T$
$x \leftarrow m;\ f(x) = m \gg\!= f$
$\gg\!= : \text{Option } T \to (T \to \text{Option } U) \to \text{Option } U$
$\text{getOrElse} : \text{Option } T \to T \to T$
$\text{toNat} : \text{Val} \to \text{Option Nat}$
$\text{toBool} : \text{Val} \to \text{Option Bool}$
$\text{toLoc} : \text{Val} \to \text{Option Loc}$

Primitive iteration:

$\# : (\text{Nat} \rightharpoonup \text{Bool}) \to \text{Nat}$
$\#f = g(0) \text{ where } g(i) = \text{if } f(i) \text{ then } i \text{ else } g(i+1)$

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

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

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

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

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

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

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

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 $x :=$ new, as well as array or field references $e_1[e_2]$ 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 due to type errors or undefined fields at runtime. 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.

First, store addresses are not flat but have structure. For dynamic allocations, rule (ENew) deterministically assigns a fresh store address $\&\text{new}{:}c$ 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 [42, 43], 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.

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 $n = \#(\lambda i. \ldots)$.

Expressions $g \in \text{Fxp}$

$g ::=$			
$n \mid b \mid l \mid []$	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
$w ::=$	Value		
$n \mid b \mid l$	Constant	$[n_0 \mapsto w_0, \ \ldots]$	Object

Figure 6. `FUN`: Target language syntax.

Translation

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

Figure 7. `FUN`: Math 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.

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 (Adequancy)**.** *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 in that we can directly read it as a translation from `IMP` to math. By mapping the math 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, etc.

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.

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:

$$w[\&\text{new:fst}.p] \rightarrow w[\text{fst}][\&\text{new}:p]$$
$$w[\&\text{new:snd}.p] \rightarrow w[\text{snd}][\&\text{new}:p]$$
$$w[\&\text{new:while}[n].p] \rightarrow w[\text{while}][n][\&\text{new}:p]$$
$$\cdots$$

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.

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 (a.k.a., 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, we report a verification failure.

Structural Equivalences

$$a[k \mapsto u][j] \equiv \text{if } j = k \text{ then } u \text{ else } a[j]$$
$$C[\text{if } c \text{ then } u \text{ else } v] \equiv \text{if } c \text{ then } C[u] \text{ else } C[v]$$
$$\text{letrec } f = (\lambda(i).[a \mapsto u, b \mapsto v]) \text{ in } e \equiv \text{letrec } f = (\lambda(i).[a \mapsto f_a(i), \mapsto f_b(i)]) \; f_a = (\lambda(i).u), f_b = (\lambda(i).v) \text{in } e$$

Collective Forms ...

$$\text{letrec } f = (\lambda(i).\ \text{if } 0 \le i \text{ then } f(i-1)[i \mapsto g_i] \text{ else } []) \text{ in } f(a) \equiv \langle . \rangle (i < a+1).\ g_i$$
$$\text{letrec } f = (\lambda(i).\ \text{if } 0 \le i \text{ then } f(i-1) + g_i \text{ else } 0) \text{ in } f(a) \equiv \Sigma(i < a+1).\ g_i$$
$$\Sigma(i < n).\ i \equiv \text{if } n \ge 0 \text{ then } n/2 * (n-1) \text{ else } 0$$
$$\#(i).\ \neg(i < a) \equiv \text{if } a \ge 0 \text{ then } a \text{ else } 0$$

Figure 8. Selected equivalences (non-exhaustive) that can be used as simplification rules, using a variety of deterministic or nondeterministic rewriting strategies.

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 $0 * e \rightarrow 0$ or 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.

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 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.

Fundamentallly, 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 [25].

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` represenation contains mutual recursion between variables:

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

Here, 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.

Fortunately, speculative rewriting provides a solution: 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.

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` as the current symbolic value of k: `let k = λ(i).n, let j = λ(i).0, let s = λ(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?

Our strategy 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

Before Loop	Before Iter i	After: Expected	After: Actual
y_0	$\hat{f}(i-1)$	$\hat{f}(i)$	$\Delta(\hat{f}(i-1))$
k = n	n	n	n
j = 0	0	0	1
s = 0	0	0	0
k = n	n	n	n
j = 0	i-1	i	i
s = 0	0	0	i-1
k = n	n	n	n
j = 0	i-1	i	i
s = 0	((i-2)* (i-1))/2	((i-1)* i)/2	((i-1)* i)/2

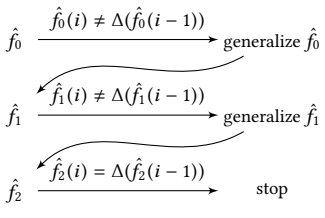

Figure 9. Fixpoint iteration for running example, iterations 0 (top) to 2 (bottom), converging to a 2nd-degree polynomial for s. The generalization treats different data types differently: (1) try a higher degree of polynomial for numerics, (2) apply generalization to fields recursively for records,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

the observed values for $i = 0$ and has derivative $d_j = 1$.

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

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.

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. The fallback (always valid) is to create a recursive definition:

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

Therefore, we can view the function space as partially ordered, from optimistic to pessimistic: $\hat{f}_0 \sqsubset \hat{f}_1 \cdots \sqsubset \hat{f}_\omega$. 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; }
```

In the same way as we are speculating 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 subsequent 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 structure is key for reasoning about programs such as 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 a standard algorithm [33]. 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.

Another feature that is required for realistic analysis is dealing with nondeterministic input, i.e., havoc. SIGMA models havoc in a very similar manner as dynamic allocations: each call to havoc is parameterized with the program context, so that the results of different havoc calls can be uniquely identified even on the symbolic level.

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 code, 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.

6 Evaluation

We evaluate SIGMA in three different categories: program properties 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 [5] and SeaHorn [17] on programs from or similar to the SV-COMP benchmarks [4]. CPAchecker won the 2018 SV-COMP competition, and both state-of-the-art tools scored highly in previous years. We use nine programs for our benchmark: three programs operate on singly linked lists, four programs are using more than one non-nested loops and two other programs

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

Figure 10. Results are in ms (TIMEOUT is set at 900s). The red cells indicate incorrect results (false positives).

have nested loops. The goal is to assess the reachability of a given function call. 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.

The results are shown in Figure 10. 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 counter example. 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.

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, 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.

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. SIGMA will generate the following closed form for the computation of a single iteration:

```
int i = 0; int ai = a[0];   // Extracted FUN code of the
a[0] = (1 + a[1])/2;        // corresponding C code:
while (i < n-1) {           let a = ⟨.⟩(i < n).
  int tmp = (ai + a[i+1])/2;  if (i == 0) then (1 + a[i+1])/2
  ai = a[i]; a[i++] = tmp;    else if (i == n-1) then (a[i-1] + 1)/2
}                             else (a[i-1] + a[i+1])/2
a[n-1] = (ai + 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 [20, 27], which supports parallel CPU, GPU, and cluster execution. The process is easily generalized to multi-dimension 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. 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 [20, 27], or to transform imperative Java code to Hadoop for cluster execution [1]. Another line of work uses symbolic execution to parallelize user-defined aggregations [34]. 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 [31]. There is also work on synthesizing MapReduce programs from sketches [44], on defining language subsets that are guaranteed to have an efficient translation [37], 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 [39].

High-Performance DSLs Some notable works in the DSL space include Delite [9, 24, 40, 48], Halide [32], and Accelerate [28, 49, 50, 54]. Most of these systems come with expressive, functional IRs. Some systems focus explicitly on the intermediate layers, for example Lift [45, 46], PENCIL [2], or the parallel action language [26].

Analysis and Optimization Our optimistic fixpoint approach is directly inspired by Lerner et al.'s work on composing dataflow analyses and transformations [25]. 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) [53] 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. [53] 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 improve precision.

Recurrence Analysis Analyzing integer recurrences has been an active topic of research. Some recent works include compositional recurrence analysis (CRA) [15, 21], which aims to derive closed forms for recurrence equations and inequations. The approach is based on an algebraic representation of path expressions [52], refered to as Newtonian program analysis [35]. Earlier works include abstract acceleration of general linear loops [19], and a study of algebraic reasoning about P-solvable loops [23]. Efficient integer linear inequality solvers have been available for some time [12, 30].

Heap Abstraction Recent work on efficient and precise points-to analysis models the heap by merging equivalent automata [51]. Other works use structured heaps to model container data structures [13], and some techniques have been proposed for heap abstractions that enable sparse global analyses for C-like languages [29], 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 [22]. Our simplificaton rules that break apart heap objects to expose their fields are inspired by such techniques. Abstracting abstract machines [18] 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 [6].

Shape analysis [41] abstractly represents the set of possible runtime stores by shape graphs, with nodes statically representing sets of runtime objects. In all instantiations of the framework we are aware of, these shape graphs are statically associated to program points (modulo context- or flow-sensitivity), which leads to the problem of being unable to precisely distinguish objects allocated in different loop iterations. Our paper presents a solution to this problem, representing all objects allocated in a loop as a sequence (a collective form) indexed by the (symbolic) loop var. We believe that this solution could be integrated in standard shape analysis to increase precision.

Shape analysis approaches based on separation logic [7, 36] improve precision and scale to large codebases [11, 14], implemented, e.g., in Facebook's Infer tool [10]. With its support for reasoning about linked list and related structures via bi-abduction [11], 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 represenations, 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 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.

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 [55] and on polyhedral compilation [3]. The main difference 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.

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.

References

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

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

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

[8] 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.

[9] 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*.

[10] 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.

[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.

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

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

[14] 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.

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

[16] 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.

[17] 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.

[18] David Van Horn and Matthew Might. 2011. Abstracting abstract machines: a systematic approach to higher-order program analysis. *Commun. ACM* 54, 9 (2011), 101–109.

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

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

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

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

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

[24] 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.

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

[26] 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.

[27] 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.

[28] Ryan Newton and Teresa Ko. 2009. Experience report: embedded, parallel computer-vision with a functional DSL. In *ICFP*. ACM, 59–64.

[29] 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.

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

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

[32] 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.

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

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

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

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

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

[38] 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)*.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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

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

[45] 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.

[46] 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.

[47] 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.

[48] 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)*.

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

[50] 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.

[51] 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.

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

[53] 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).

[54] 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.

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