Combinator-Based Fixpoint Algorithms for Big-Step Abstract Interpreters

SVEN KEIDEL, TU Darmstadt, Germany
SEBASTIAN ERDWEG and TOBIAS HOMBÜCHER, JGU Mainz, Germany

Big-step abstract interpreters are an approach to build static analyzers based on big-step interpretation. While big-step interpretation provides a number of benefits for the definition of an analysis, it also requires particularly complicated fixpoint algorithms because the analysis definition is a recursive function whose termination is uncertain. This is in contrast to other analysis approaches, such as small-step reduction, abstract machines, or graph reachability, where the analysis essentially forms a finite transition system between widened analysis states.

We show how to systematically develop sophisticated fixpoint algorithms for big-step abstract interpreters and how to ensure their soundness. Our approach is based on small and reusable fixpoint combinators that can be composed to yield fixpoint algorithms. For example, these combinators describe the order in which the program is analyzed, how deep recursive functions are unfolded and loops unrolled, or they record auxiliary data such as a (context-sensitive) call graph. Importantly, each combinator can be developed separately, reused across analyses, and can be verified sound independently. Consequently, analysis developers can freely compose combinators to obtain sound fixpoint algorithms that work best for their use case. We provide a formal metatheory that guarantees a fixpoint algorithm is sound if its composed from sound combinators only. We experimentally validate our combinator-based approach by describing sophisticated fixpoint algorithms for analyses of Stratego, Scheme, and WebAssembly.

1 INTRODUCTION

Abstract interpretation [Cousot and Cousot 1977] is a methodology for defining sound static analyses. While in the past, many static analyses have been described as abstract interpreters in small-step style [Darais et al. 2015; Horn and Might 2010; Might and Shivers 2006a,b; Schmidt 1996; Sergey et al. 2013], more recently big-step abstract interpreters have been investigated more thoroughly [Bodin et al. 2019; Darais et al. 2017; Keidel and Erdweg 2019; Keidel et al. 2018; Wei et al. 2019]. Such big-step abstract interpreters can be simply described as recursive functions in any meta-language; we use Haskell as a meta-language throughout this paper.

Big-step abstract interpreters (sometimes called definitional abstract interpreters) look like the corresponding concrete interpreter, except they compute with abstract data. For example, consider the big-step abstract interpreter in Listing 1 that approximates values using intervals. Big-step abstract interpreters like this are easy to understand [Darais et al. 2017] and reason about [Keidel et al. 2018], while they seamlessly combine data-flow and control-flow information. However, our abstract interpreter does not terminate on all inputs, since it calls itself unconditionally in the last line of Listing 1. While non-termination is expected language behavior for concrete interpreters, it is undesirable for abstract interpreters. But how can we ensure our big-step abstract interpreter terminates?

ICFP'23, September 4–9, 2023, Seattle, Washington, United States

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
eval^ :: Funs → Env^ → Expr → Maybe Val^
eval^ funs env expr = case expr of
  Var x → lookup x env
  Num n → return (n,n)
  Add e1 e2 → do
    (i1,i2) ← eval^ funs env e1
    (j1,j2) ← eval^ funs env e2
    return (i1+j1,i2+j2)
  Call funName args → do
    let (params, body) = lookup funName funs
    vs ← for args (eval^ funs env)
    eval^ funs (Map.fromList (zip params vs)) body
```

```
Expr = Var String
     | Num Int
     | Add Expr Expr
     | Call String [Expr]
     | ...
Funs = Map [String] (String, Expr)
Env^ = Map String Val^
Val^ = Interval
```

Listing 1. A big-step abstract interpreter for a language with first-order functions. The abstract interpreter is sound but fails to terminate for diverging recursive functions.

In this work, we study fixpoint algorithms for big-step abstract interpreters (*big-step fixpoint algorithms* for short) and how to describe and reason about them modularly. Like most fixpoint algorithms, big-step fixpoint algorithms must apply the abstract interpreter repeatably until the analysis result is stable. However, the recursive definition of big-step abstract interpreters makes it difficult to ensure termination. In particular, it is insufficient to limit the number of call sites in the object language, because abstract interpreters still end up in an infinite recursive loop when calls recur. Schmidt [1995] was the first to propose a fixpoint algorithm for big-step abstract interpreters. We reformulate his work and combine it with the chaotic iteration strategy of Bourdoncle [1993] to derive a novel big-step fixpoint algorithm: a fixpoint algorithm based on chaotic iteration over strongly-connected subgraphs of the dynamically discovered graph-shaped trace of an abstract interpreter.

Our initial big-step fixpoint algorithm is sound and ensures termination, although its implementation and soundness proof are complex and monolithic. We argue that fixpoint algorithms should be described and proven sound modularly because practical analyses require specialized and fine-tuned fixpoint algorithms:

Specialized fixpoint algorithms. Different analyses and languages often require specialized fixpoint algorithms. For example, the Soot framework [Lam et al. 2011] describes 7 different fixpoint algorithms for different types of analyses: Two algorithms[1] for distributive analysis problems [Reps et al. 1995; Sagiv et al. 1995], two algorithms[2] for bidirectional analyses problems such as taint analysis [Lerch et al. 2014], two algorithms[3] for flow-sensitive analysis problems [Späth et al. 2016], and one algorithm[4] for context, flow and field-sensitive analysis problems [Späth et al. 2019].
However, it takes effort to develop and maintain many different fixpoint algorithms and their soundness proofs. Therefore many proofs become outdated over time and they become ineffective at guaranteeing soundness of the analysis.

Fine-tuning existing fixpoint algorithms. Fixpoint algorithms require continuous fine-tuning to yield satisfactory performance and precision. In particular, no fixpoint algorithm works best in all cases and configurability is key. For example, consider the continuous changes to

[1] https://github.com/Sable/heros/blob/develop/src/heros/solver/IFDSSolver.java and IDESolver.java
[2] https://github.com/Sable/heros/blob/develop/src/heros/solver/BiDiIFDSSolver.java and BiDiIDESolver.java
[3] https://github.com/Sable/heros/blob/develop/src/heros/fieldsens/FieldSensitiveIFDSSolver.java and FieldSensitiveBiDiIFDSSolver.java
[4] https://github.com/CodeShield-Security/SPDS/blob/master/WPDS/src/main/java/wpds/impl/WeightedPAutomaton.java

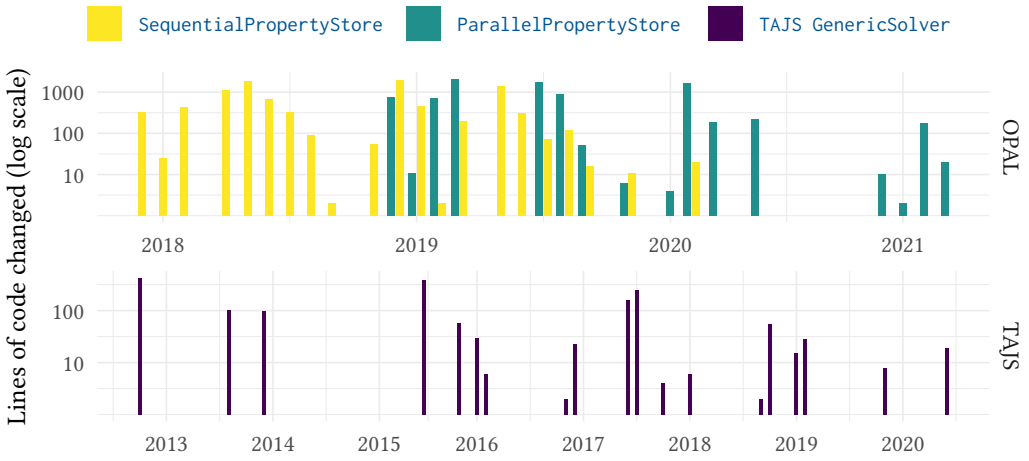


Fig. 1. Monthly changes to fixpoint algorithms in the OPAL and TAJS analysis frameworks.

the fixpoint algorithms in the OPAL [Helm et al. 2020] and TAJS [Jensen et al. 2009] analysis frameworks depicted in Figure 1.

Unfortunately, fine-tuning a single monolithic fixpoint algorithm has two problems. First, tuning it for one analysis may lead to regressions for other analyses that use the same algorithm. Second, every change to the fixpoint algorithm can introduce a soundness bug, yet reestablishing the soundness proof for every single change is infeasible in practice. These two problems cause framework developers either to avoid fine-tuning their fixpoint algorithms or to avoid proving them sound rigorously.

To support such scenarios, we propose a novel approach for the implementation of big-step fixpoint algorithms based on reusable and separately verifiable *fixpoint combinators*. Note that we mean *combinators* in the sense of parser combinators, where complex functions can be constructed by composing simpler functions.[5] For example, fixpoint combinators describe the order in which the program statements are analyzed, how deep recursive functions are unfolded or loops are unrolled, or they record auxiliary data such as a control-flow graph. A complete fixpoint algorithm can then be composed by choosing appropriate fixpoint combinators (each starting with φ):

$$\varphi_{\texttt{filter}}\ \texttt{isFunBody}\ (\varphi_{\texttt{unfold}}\ 3\ \varphi_{\texttt{stackWiden}} \circ \varphi_{\texttt{innermost}})$$

This fixpoint algorithm only applies to function bodies ($\varphi_{\texttt{filter}}\ \texttt{isFunBody}$) and unfolds the first 3 recursive function calls ($\varphi_{\texttt{unfold}}$) before it applies a widening operator on the stack ($\varphi_{\texttt{stackWiden}}$). In case of nested recursive function calls, the fixpoint algorithm stabilizes the analysis result of the innermost calls first ($\varphi_{\texttt{innermost}}$).

This modular description allows analysis developers to specialize and fine-tune fixpoint algorithms by reconfiguring individual combinators or adding specialized ones. For example, we can extend the fixpoint algorithm from above to record the call graph ($\varphi_{\texttt{CFG}}$) and to handle while-loops:

$$\varphi_{\texttt{filter}}\ \texttt{isFunBody}\ (\varphi_{\texttt{CFG}} \circ \varphi_{\texttt{unfold}}\ 3\ \varphi_{\texttt{stackWiden}} \circ \varphi_{\texttt{innermost}}) \circ$$
$$\varphi_{\texttt{filter}}\ \texttt{isWhileLoop}\ (\varphi_{\texttt{unroll}}\ 10\ \varphi_{\texttt{stackWiden}} \circ \varphi_{\texttt{outermost}})$$

This fixpoint algorithm seamlessly interleaves the intra-procedural analysis of loops with the inter-procedural analysis of recursive function calls. Both of these aspects can be individually

[5]Technically, we mean *non-standard* fixpoint combinators φ that do not necessarily satisfy the standard fixpoint property $\varphi(f) = f(\varphi(f))$, but rather $\varphi(f) \sqsupseteq f(\varphi(f))$, which is sufficient for soundness.

changed and fine-tuned by adding, replacing, and reordering fixpoint combinators. And of course we can use standard function abstraction to make parts of the fixpoint algorithm reusable.

We also modularize the soundness proofs of big-step fixpoint algorithms by developing a formal theory for fixpoint combinators. The modularization simplifies the effort and complexity of these soundness proofs and makes them composable. In particular, we prove that a modular fixpoint algorithm is sound if all of its combinators are sound. This not only simplifies the initial soundness proof for a fixpoint algorithm but also makes it easier to reestablish soundness after a change.

We demonstrate that our approach is feasible and useful by implementing it in Haskell as part of the *Sturdy* framework [Keidel and Erdweg 2019; Keidel et al. 2018].[6] We developed 12 fixpoint combinators and composed them to obtain fixpoint algorithms for 3 analyses of 3 different languages: WebAssembly, Stratego, and Scheme. We use these case studies to assess the language and analysis-independence, the precision, and the performance of the fixpoint algorithms. We find that the initial fixpoint algorithms perform poorly, but they can be easily specialized to the analysis without changing the implementation of any of the fixpoint combinators. We conclude that configurable fixpoint algorithms are necessary to allow analysis developers to fine-tune their analyses.

In summary, we make the following contributions:

- We combine prior work on big-step fixpoint algorithms [Schmidt 1995] and chaotic iteration [Bourdoncle 1993] to develop a novel big-step fixpoint algorithm (Section 2).
- We propose an approach to modularize the description of big-step fixpoint algorithms through sound and reusable fixpoint combinators (Section 3).
- We present a library of reusable fixpoint combinators that serve as building blocks for developing fixpoint algorithms (Section 4).
- We develop a formal theory for these combinators that allows us to prove their soundness separately and once and for all (Section 5).
- We demonstrate that our approach is feasible and useful by implementing it as part of the Sturdy framework (Section 6).

2 DESIGNING BIG-STEP FIXPOINT ALGORITHMS

In this section, we first describe conditions that guarantee the termination of big-step fixpoint algorithms. In the second half, we develop a big-step fixpoint algorithm that satisfies these conditions.

2.1 Enforcing Termination of Big-Step Fixpoint Algorithms

Schmidt [1995] introduced big-step abstract interpretation and showed how to compute its fixpoint. We reformulate his findings as three conditions that guarantee the termination of big-step fixpoint algorithms. Consider the analysis of the factorial function implemented in a language with first-order functions. The following diagram shows a big-step reduction trace of an abstract interpreter with unbounded recursion, where $\rho \vdash e \Downarrow v$ evaluates an expression e under environment ρ to an abstract value v. Such a trace looks similar to the trace of a concrete big-step interpreter, except that the values are intervals.

[6] The artifact is available at https://doi.org/10.5281/zenodo.7986916

```
fact(n) =
  if(n==0) 1
  else fact(n-1) * n
```

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\mathtt{n} \mapsto [0,\infty] \vdash \mathtt{if(n == 0)\ 1\ else\ fact(n-1) * n} \Downarrow ?}}{\mathtt{n} \mapsto [1,\infty] \vdash \mathtt{fact(n-1)} \Downarrow ?}}{\mathtt{n} \mapsto [1,\infty] \vdash \mathtt{fact(n-1) * n} \Downarrow ?} \qquad \mathtt{n} \mapsto [0,0] \vdash \mathtt{1} \Downarrow [1,1]}{\mathtt{n} \mapsto [0,\infty] \vdash \mathtt{if(n == 0)\ 1\ else\ fact(n-1) * n} \Downarrow ?}$$
$$\mathtt{n} \mapsto [0,\infty] \vdash \mathtt{fact(n)} \Downarrow ?$$

The analysis starts at the call `fact(n)`, where `n` is bound to the interval $[0,\infty]$ in the environment. Because the interval $[0,\infty]$ contains 0 and other numbers, the abstract interpreter has to evaluate both branches of the conditional `if(n == 0)` and join the results. Whereas the analysis of the first branch terminates after only one step, the second branch diverges while recurrently calling the factorial function with the same environment over and over again (see highlighted calls). We write the question mark symbol to represent that the abstract interpreter diverged and did not produce a result. This leads us to the first condition:

Condition 1 *A big-step fixpoint algorithm has to detect recurrent recursive calls and cut off recursion to avoid non-termination.*

Detecting recurrent calls allows the fixpoint algorithm to iterate that part of the computation that spans the initial call and the recurrent call. One way of detecting recurrent recursive calls is to remember the calls of the abstract interpreter on each branch of the derivation tree. Each call consists of the inputs of the abstract interpreter, e.g., an expression and an environment. By remembering the calls, we can easily detect a diverging call, if the exact same call occured earlier, further down the derivation branch.

However, this way of detecting recurrent recursive calls is insufficient. For example, consider the analysis of the factorial function for negative arguments. Clearly, the factorial function does not terminate for negative arguments, and we expect the abstract interpreter to return an analysis result that represents non-termination. Instead, the abstract interpreter itself diverges:

```
fact(n) =
  if(n==0) 1
  else fact(n-1) * n
```

$$\dfrac{\dfrac{\dfrac{\vdots}{\mathtt{n} \mapsto [-\infty,-2] \vdash \mathtt{if(n == 0)\ 1\ else\ fact(n-1) * n} \Downarrow ?}}{\mathtt{n} \mapsto [-\infty,-1] \vdash \mathtt{if(n == 0)\ 1\ else\ fact(n-1) * n} \Downarrow ?}}{\mathtt{n} \mapsto [-\infty,-1] \vdash \mathtt{fact(n)} \Downarrow ?}$$

The abstract interpreter analyzes the factorial function with smaller and smaller intervals, because factorial decrements its argument on every recursive call. Even though the intervals become smaller, the chain of recursive calls is still infinite. Therefore, the fixpoint algorithm never encounters a recurrent recursive call. This means that a fixpoint algorithm that satisfies the first condition still may not terminate. This leads us to the second condition:

Condition 2 *A big-step fixpoint algorithm has to ensure that all possibly infinite call chains have a recurrent call.*

In other words, all call chains are either finite or repeat themselves after finitely many calls. This ensures that a fixpoint algorithm can find a recurrent call even in infinite call chains.

While the first and second condition concern the inputs of the abstract interpreter, the third condition concerns its outputs. To illustrate this condition, consider an interval analysis of the multiplication function on Peano numbers, where we initially bind `m` to $[1,\infty]$ and `n` to $[1,1]$.

```
mult(m,n) =
  if(m==1) n
  else mult(m-1,n) + n
```

$$\dfrac{\dfrac{\dfrac{}{\mathtt{n} \mapsto [1,1] \vdash \mathtt{n} \Downarrow [1,1]} \quad \dfrac{\dfrac{\dfrac{}{\mathtt{m} \mapsto [2,\infty], \mathtt{n} \mapsto [1,1] \vdash \mathtt{mult}(\mathtt{m}-1,\mathtt{n}) \Downarrow X}}{\mathtt{m} \mapsto [2,\infty], \mathtt{n} \mapsto [1,1] \vdash \mathtt{mult}(\mathtt{m}-1,\mathtt{n})+\mathtt{n} \Downarrow X + [1,1]}}{\mathtt{m} \mapsto [1,\infty], \mathtt{n} \mapsto [1,1] \vdash \mathtt{if}(\mathtt{m}==1)\ \mathtt{n}\ \mathtt{else}\ \mathtt{mult}(\mathtt{m}-1,\mathtt{n})+\mathtt{n} \Downarrow [1,1] \sqcup (X + [1,1])}}{\mathtt{m} \mapsto [1,\infty], \mathtt{n} \mapsto [1,1] \vdash \mathtt{mult}(\mathtt{m},\mathtt{n}) \Downarrow ?}$$

The right-hand side branch of the derivation tree contains a recurrent call of `mult`. In this example, we represent the result of the recurrent call with a symbolic variable X. By tracing back the result to the initial call of `mult`, we obtain the recursive equation $X = [1,1] \sqcup (X + [1,1])$. An established technique for solving such an equation is to start with the empty interval $\bot$ and then to proceed iteratively until reaching a fixpoint [Cousot and Cousot 1992]. However, starting with $\bot$, this technique does not reach a fixpoint in a finite number of steps for our example:

$$X_0 = \bot \quad X_1 = [1,1] \sqcup (X_0 + [1,1]) = [1,1] \quad X_2 = [1,1] \sqcup (X_1 + [1,1]) = [1,2] \quad \ldots$$

This example shows that even if a big-step fixpoint algorithm ensures and detects recurrent calls, it still might iterate on the analysis result indefinitely. This leads us to the third condition:

Condition 3 *A big-step fixpoint algorithm may only iterate the results a finite number of times.*

Our three conditions guarantee termination:

Theorem 2.1 (Termination). *If a big-step fixpoint algorithm satisfies the three termination conditions and all reduction rules have a finite branching factor, then the big-step fixpoint algorithm terminates.*

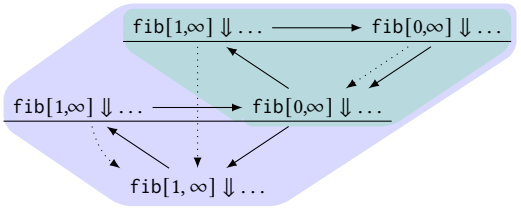
Proof. *Condition 1* and *2* ensure that each infinite call chain is eventually cut off at a recurrent call and hence is finite. *Condition 3* ensures that the fixpoint algorithm iterates on the analysis result for each node of the tree finitely many times. Finite call chains, finite iteration, and the finite branching factor of the rules guarantee that the big-step derivation tree is finite. Therefore, the fixpoint algorithm terminates. □

To summarize, a big-step fixpoint algorithm terminates if it satisfies the termination conditions. In the following subsection, we describe a big-step fixpoint algorithm that satisfies the termination conditions.

2.2 A Big-Step Fixpoint Algorithm that iterates on Strongly-Connected Subgraphs

In this section, we describe a novel fixpoint algorithm for big-step abstract interpreters that iterates on the strongly-connected subgraph of the graph-shaped trace of the abstract interpreter. The fixpoint algorithm targets the simple functional language from Section 2.1, but we generalize it in Section 3 by making it language-independent and modular.

```
fib(n) = if(n==0) 0
         else if(n==1) 1
         else fib(n-1) + fib(n-2)
```



The fixpoint algorithm iterates on the strongly-connected subgraphs (SCGs) of the graph-shaped trace of the abstract interpreter [Bourdoncle 1993]. An SCG is a set of calls from which it is possible to reach all other calls in the same set. SCGs in the abstract interpreter trace occur if the analyzed program has cyclic dependencies, such as loops or recursive functions. For example, consider the graph-shaped trace to the right for the analysis of the Fibonacci function starting at call `fib`[1,∞]. We write `fib`[i,j] as a shorthand for a call $[\mathtt{n} \mapsto [i,j]] \vdash \mathtt{fib}(\mathtt{n})$. The graph has an *outer*

SCG and an *inner* SCG indicated by the differently shaded areas. The solid arrows indicate calls and returns in the order in which they are executed by the abstract interpreter. The dotted arrows indicate recurrent calls. A call is recurrent when that same call is already on the call stack. In our example, we have three recurrent calls and the dotted arrow indicates the outermost dominator. We explain later in Figure 2 how this trace is computed in more detail.

To compute a fixpoint, the algorithm has to iterate on all calls in the body of an SCG. The order in which the fixpoint algorithm iterates over the calls does not matter for soundness [Bourdoncle 1993], but affects performance and precision of the analysis. In this section we present an algorithm that prioritizes calls in the *innermost* SCGs, before iterating on the outer SCGs. The trace of the abstract interpreter only becomes known while the analysis is running. Hence, the fixpoint algorithm cannot compute SCGs a priori and instead it must discover SCGs on the fly while the analysis is running. To detect SCGs, our algorithm tracks recurrent calls, because some recurrent calls are the entry calls of SCGs. For example, in the trace of the Fibonacci function above the recurrent call $\texttt{fib}[0,\infty]$ points to the entry call of the inner SCG (rightmost dotted arrow), whereas the recurrent calls of $\texttt{fib}[1,\infty]$ point to the entry call of the outer SCG. To detect the *innermost* SCGs, the algorithm looks for the first recurrent call that it encounters upon returning.

Listing 2 shows the adapted abstract interpreter $\widehat{\texttt{eval}}$ and the main fixpoint algorithm $\texttt{fix}_{\texttt{monolithic}}$. Instead of calling itself recursively like in Listing 1, the abstract interpreter $\widehat{\texttt{eval}}$ calls $\texttt{fix}_{\texttt{monolithic}}$ to evaluate subexpressions, and $\texttt{fix}_{\texttt{monolithic}}$ calls $\widehat{\texttt{eval}}$ mutual recursively. This allows us to encapsulate the fixpoint logic in $\texttt{fix}_{\texttt{monolithic}}$, whereas $\widehat{\texttt{eval}}$ captures the rest of the abstract language semantics, which we do not show for brevity. Our fixpoint algorithm uses three data structures: A map $\widehat{\texttt{Stack}}$ to detect recurrent calls, storing for each expression `Expr` the abstract environment $\widehat{\texttt{Env}}$ under which the expression is evaluated. A map $\widehat{\texttt{Cache}}$ to iterate on analysis results, storing for each abstract $\widehat{\texttt{Call}}$ the abstract value $\widehat{\texttt{Val}}$ to which the call evaluated. A set $\widehat{\texttt{SCG}}$ to detect which calls need to be iterated, containing recurrent recursive calls.

The algorithm first checks in line 9, if the expression is a function body and hence a potentially diverging call. If the expression is not a function body (e.g., a numeric operator), no iteration is necessary to find a fixpoint and we can simply call $\widehat{\texttt{eval}}$. This not only saves analysis time, but also reduces the size of the stack and cache tremendously. If the expression is a function body, the algorithm then checks if the cache contains a stable analysis result for the call and returns this result to avoid redundant reanalysis (line 10). Analysis results are stable if they do not grow anymore when reevaluated and if they solely depend on other stable analysis results. If the cache only contains an unstable or no analysis result, the algorithm checks if the call `(env,expr)` is a recurrent call by searching for it on the stack. In case of a recurrent call, the algorithm satisfies *Condition 1* by either returning the unstable analysis result (line 11) or returning $\bot$ (line 12). Furthermore, since the analysis result needs to be iterated on, the algorithm adds its call to the SCG set. If the call does not appear on the stack, the algorithm calls a recursive helper function `iterate` (line 13) that iterates the analysis result until it stabilizes.

Function `iterate` is responsible for iterating on calls in SCGs. The first line of `iterate` applies a widening operator [Cousot and Cousot 1992] $\nabla_{\texttt{Stack}}$ to the stack and the call. This widening operator ensures that all infinite non-repeating stacks eventually have a recurrent call (*Condition 2*). We explain this operator in more detail below. In line 19, the algorithm calls the abstract interpreter $\widehat{\texttt{eval}}$ with the widened inputs. The algorithm then iterates on the call, in case the call is a head of an SCG (line 20), or otherwise simply returns the result of $\widehat{\texttt{eval}}$ (line 28). In line 22, the algorithm uses a widening operator for values $\nabla_{\widehat{\texttt{Val}}}$ to ensure that the analysis result does not grow indefinitely (*Condition 3*). If the widened value is strictly greater than the cached value, the algorithm keeps iterating (line 26). Otherwise, if the widened value did not grow anymore, the algorithm terminates

```
1  Stack^ = Map Expr Env^      Cache^  = Map Call^ (Stable,Val^)    SCG^ = Set Call^
2  Call^  = (Env^,Expr)        Stable = Stable | Unstable
3
4  eval^ :: Call^ → Stack^ → Cache^ → (Val^, Cache^, SCG^)
5  eval^ (env,expr) stack cache = case expr of ... fix_monolithic (env',expr') stack cache ...
6
7  fix_monolithic :: Call^ → Stack^ → Cache^ → (Val^, Cache^, SCG^)
8  fix_monolithic call stack cache
9    | not (isFunctionBody call)                  = eval^ call stack cache
10   | call ∈ cache && cachedValStable == Stable   = (val_cached, cache, ∅)
11   | call ∈ cache && cachedValStable == Unstable = (val_cached, cache, {call})
12   | call ∉ cache && call ∈ stack                = (⊥,        cache, {call})
13   | call ∉ cache && call ∉ stack                = iterate call stack cache
14   where (cachedValStable,val_cached) = cache(call)
15
16 iterate :: Call^ → Stack^ → Cache^ → (Val^, Cache^, SCG^)
17 iterate call stack cache_1 =
18   let (stack_widened, call_widened) = stack ∇_Stack^ call
19       (val_new, cache_2, scg)     = eval^ call_widened stack_widened cache_1
20   if call_widened ∈ scg then
21     let val_old    = if call_widened ∈ cache_2 then cache_2(call_widened) else ⊥
22         val_widened = val_old ∇_Val^ val_new
23         stable    = if val_widened ⊑ val_old && size scg == 1 then Stable else Unstable
24         cache_3    = cache_2[call_widened ↦ (stable,val_widened)]
25     if val_old ⊏ val_widened
26     then iterate call stack cache_3
27     else (val_widened, cache_3, scg \ {call_widened})
28   else (val_new, cache_2, scg)
29
30 ∇_Stack^ :: Stack^ → Call^ → (Stack^, Call^)
31 ∇_Stack^ stack (env_1, expr)
32   | expr ∈ dom stack && env_1 ⊑ env_2 = (stack,                    (env_2, expr))
33   | expr ∈ dom stack && env_1 ⋢ env_2 = (stack[expr ↦ env_widened], (env_widened, expr))
34   | expr ∉ dom stack                 = (stack[expr ↦ env_1],      (env_1, expr))
35   where env_2 = stack(expr)
36         env_widened = env_2 ∇_Env^ env_1
37
38 ∇_Env^ :: Env^ → Env^ → Env^
39 ∇_Val^ :: Val^ → Val^ → Val^
```

Listing 2. Big-step fixpoint algorithm iterating on the innermost strongly-connected subgraph. The code uses common mathematical notation for operations on maps and sets for readability. In particular, the notation cache(call) looks up the key call in the map cache and the notation cache[call ↦ res] updates the map entry call to res. Furthermore, {call} refers to the singleton set with the element call.

the iteration, returns the widened value, and removes the call from the SCG since it does not require iteration anymore (line 27).

The widening operator ∇_{Stack} ensures that all infinite non-repeating stacks eventually have a recurrent call (*Condition 2*). If the expression appeared on the stack and the environment of the call is smaller than the environment on the stack (line 32), the stack widening operator introduces a recurrent call by reusing the environment on the stack. If the environment on the stack is not an upper bound of the environment in the call (line 33), the stack widening operator applies a

```
1   ↳ ⟨fib[1,∞], ρ1, σ1⟩
2       ↳ ⟨fib[1,∞], ρ2, σ1⟩ ↲ ⟨⊥, σ1, θ1⟩
3       ↳ ⟨fib[0,∞], ρ2, σ1⟩4
4           ↳ ⟨fib[1,∞], ρ3, σ1⟩ ↲ ⟨⊥, σ1, θ1⟩
5           ↳ ⟨fib[0,∞], ρ3, σ1⟩ ↲ ⟨⊥, σ1, θ2⟩
6           ↲ ⟨[0,0]⊔[1,1]⊔(⊥+⊥), σ1, θ3⟩
7       ⟨fib[0,∞], ρ2, σ2⟩  -- iterate
8           ↳ ⟨fib[1,∞], ρ3, σ2⟩ ↲ ⟨⊥, σ2, θ1⟩
9           ↳ ⟨fib[0,∞], ρ3, σ2⟩ ↲ ⟨[0,1], σ2, θ2⟩
10          ↲ ⟨[0,0]⊔[1,1]⊔(⊥+[0,1]), σ2, θ3⟩
11      ↲ ⟨[1,1]⊔(⊥+[0,1]), σ2, θ1⟩
12  ⟨fib[1,∞], ρ1, σ3⟩1  -- iterate
13      ↳ ⟨fib[1,∞], ρ2, σ3⟩ ↲ ⟨[1,1], σ3, θ1⟩
14      ↳ ⟨fib[0,∞], ρ2, σ3⟩4
15          ↳ ⟨fib[1,∞], ρ3, σ3⟩ ↲ ⟨[1,1], σ3, θ1⟩
16          ↳ ⟨fib[0,∞], ρ3, σ3⟩ ↲ ⟨[0,1], σ3, θ2⟩
17          ↲ ⟨[0,0]⊔[1,1]⊔([1,1]+[0,1]), σ3, θ3⟩
18      ⟨fib[0,∞], ρ2, σ4⟩  -- iterate
19          ↳ ⟨fib[1,∞], ρ3, σ4⟩ ↲ ⟨[1,1], σ4, θ1⟩
20          ↳ ⟨fib[0,∞], ρ3, σ4⟩ ↲ ⟨[0,∞], σ4, θ2⟩
21          ↲ ⟨[0,0]⊔[1,1]⊔([1,1]+[0,∞]), σ4, θ3⟩
22      ↲ ⟨[1,1]⊔([1,1]+[0,∞]), σ4, θ1⟩
23  ⟨fib[1,∞], ρ1, σ5⟩  -- iterate
24      ...
25      ↲ ⟨[1,1]⊔([1,∞]+[0,∞]), σ6, θ1⟩
```

Stacks:

$\rho_1 = \varnothing$

$\rho_2 = \{\texttt{fib}[1,\infty]\}$

$\rho_3 = \{\texttt{fib}[1,\infty], \texttt{fib}[0,\infty]\}$

Caches:

$\sigma_1 = \varnothing$

$\sigma_2 = \sigma_1[\texttt{fib}[0,\infty] \mapsto (\texttt{Unstable}, [0,1])]$

$\sigma_3 = \sigma_2[\texttt{fib}[1,\infty] \mapsto (\texttt{Unstable}, [1,1])]$

$\sigma_4 = \sigma_3[\texttt{fib}[0,\infty] \mapsto (\texttt{Unstable}, [0,\infty])]$

$\sigma_5 = \sigma_4[\texttt{fib}[1,\infty] \mapsto (\texttt{Unstable}, [1,\infty])]$

$\sigma_6 = \sigma_5[\texttt{fib}[1,\infty] \mapsto (\texttt{Stable}, [1,\infty])]$

SCGs:

$\theta_1 = \{\texttt{fib}[1,\infty]\}$

$\theta_2 = \{\texttt{fib}[0,\infty]\}$

$\theta_3 = \{\texttt{fib}[1,\infty], \texttt{fib}[0,\infty]\}$

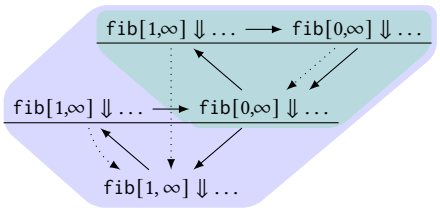


Fig. 2. Example trace of the abstract interpreter analyzing the Fibonacci function. Arrow ↳ represents a call, arrow ↲ a return, and arrow an iteration. The highlighting indicates which results changed between consecutive iteration. The indentation level indicates the depth of the stack.

widening operator $\nabla_{\widehat{\text{Env}}}$ to both environments. Operator $\nabla_{\widehat{\text{Env}}}$ computes an upper bound of both environments and ensures that the environment under which an expression is evaluated cannot grow infinitely. Lastly, in case the expression did not occur on the stack (line 34), the operator adds the call to the stack without changing it.

We illustrate how this algorithm works at an example of the analysis of the Fibonacci function. Figure 2 shows a trace of the abstract interpreter starting at `fib[1,∞]`. To make the internals of the fixpoint algorithm visible, we write ↳ $\langle e, \rho, \sigma\rangle$ for a call e with stack ρ and cache σ. Furthermore, we write ↲ $\langle v, \sigma, \theta\rangle$ for a return from a call with result value v, output cache σ, and SCG θ. Sometimes we show intermediate steps in the abstract interpretation, where we allow v to be an expression that evaluates to an interval. The highlighting indicates which analysis results changed between consecutive iterations.

The algorithm alternatingly iterates the innermost and the outermost SCG shown in the bottom right of Figure 2. In the beginning the algorithm explores the recursive calls of the fibonacci function until it hits the recurrent calls (lines 2, 4, and 5 in Figure 2). In these cases the algorithm returns $\bot$ to avoid non-termination and adds the call to the SCG set (line 12 in Listing 2). Later the algorithm returns to the call of `fib[0,∞]` (line 6 in Figure 2) and iterates because the call is in SCG set θ_2 and the result interval has grown from $\bot$ to [0,1] (lines 20 and 25 in Listing 2). The following iteration propagates the new analysis result `fib[0,∞]` $\mapsto$ [0,1] throughout the inner SCG. After returning to call `fib[0,∞]` (line 10 in Figure 2), the result did not grow and the algorithm returns to the

surrounding call `fib`[1,∞], removing call `fib`[0,∞] from the SCG set (line 27 in Listing 2). Since the result for `fib`[1,∞] has grown from ⊥ to [1,1], the algorithm iterates again and propagates the new analysis result `fib`[1,∞] ↦ [1,1] throughout the outer SCG. After returning to call `fib`[0,∞] (line 17 in Figure 2), the result has grown from [0,1] to [0,2] and hence the algorithm widens the result to $[0,1]\nabla_{\widehat{\texttt{Val}}}[0,2] = [0,\infty]$ (line 22 in Listing 2). Since the result is greater after widening, the algorithm iterates the call `fib`[0,∞] again until it does not grow anymore. Finally, after one more iteration of call `fib`[1,∞], the result [1,∞] does not grow anymore (line 25 in Figure 2) and the algorithm sets the cache entry to stable (line 23 in Listing 2).

In summary, we developed a big-step fixpoint algorithm that iterates on the strongly-connected subgraph of the graph-shaped trace and satisfies the termination conditions. We prove soundness of this algorithm in Section 5.

3 MODULAR DESCRIPTION OF BIG-STEP FIXPOINT ALGORITHMS

In the previous section, we discussed a big-step fixpoint algorithm that iterates on the strongly-connected subgraphs of the graph-shaped trace. Even though the initial fixpoint algorithm works and is sound, it is hard to specialize and fine-tune it. We can solve these problems by *modularizing* the description of big-step fixpoint algorithms, which we discuss in this section. We illustrate the modular description by refactoring the function $\texttt{fix}_{\texttt{monolithic}}$ into smaller reusable fixpoint combinators. Additionally, this modularization will enable us to prove soundness of the fixpoint algorithm modularly, which we discuss in Section 5.

Language-Independence. The problem that makes function $\texttt{fix}_{\texttt{monolithic}}$ language-dependent is that it refers to the abstract interpreter $\widehat{\texttt{eval}}$, environments, expressions, and values from the analyzed language directly. To make the algorithm language-independent, we first remove references to language-specific types. As first step, we replace the inputs (`Env`,`Expr`) and outputs $\widehat{\texttt{Val}}$ of the abstract interpreter with the type variables `a` and `b`.

```
Stack a = Set a          Cache a b = Map a b          SCG a = Set a
```

As second step, we remove the reference to $\widehat{\texttt{eval}}$ by turning it into an open-recursive style and passing its body as an argument to $\texttt{fix}_{\texttt{monolithic}}$. This allows us to implement $\texttt{fix}_{\texttt{monolithic}}$ independently of the analyzed language.

```
eval :: (Env,Expr) ⇓ Val
eval = fix_monolithic (λeval_rec ((env,expr),stack,cache) →
                        case expr of ... eval_rec ((env',expr'),stack,cache) ...)

fix_monolithic :: (a ⇓ b → a ⇓ b) → a ⇓ b
```

To this end, we introduced a type (⇓) to represent the type of fixpoint computation:

```
a ⇓ b = (a, Stack a, Cache a b) → (Cache a b, SCG a, b)
```

We refrain from showing the new code of $\texttt{fix}_{\texttt{monolithic}}$ until after the second refactoring.

Reusable Fixpoint Combinators. To make the fixpoint algorithm easier to specialize to a new analysis, we make two more changes. First, instead of implementing one single monolithic fixpoint algorithm, we split its functionality across multiple smaller fixpoint combinators $\varphi_1, \ldots \varphi_n$. These combinators are then called by a function `fix` in a round-robin fashion, such that each combinator has the chance to affect the fixpoint computation:

```
eval = fix_φ (λeval_rec ...)

fix_φ :: (a ⇓ b → a ⇓ b) → a ⇓ b      φ :: a ⇓ b → a ⇓ b
fix_φ f = φ (f (fix_φ f))             φ f (call,stack,cache) = φ1(φ2(...φn(f)...)) (call,stack,cache)
```

In particular, $\texttt{fix}_{\varphi}$ $(\lambda\widehat{\texttt{eval}}_{\texttt{rec}}\ldots)$ invokes combinator φ. Combinator φ first invokes combinator φ_1, which then may invoke φ_2, and so on, until eventually φ_n calls $(\lambda\widehat{\texttt{eval}}_{\texttt{rec}}\ldots)$ $(\texttt{fix}_{\varphi}$ $(\lambda\widehat{\texttt{eval}}_{\texttt{rec}}\ldots))$ and the cycle repeats.

Even though this design of fixpoint combinators allows us to separate concerns, their type `a ⇓ b` is not fully extensible, as some combinators may need some extra data not present in the stack or the cache. Therefore, as second change, we generalize the type `a ⇓ b` to an arrow type `c a b` [Hughes 2000]. The arrow type reads as "some effectful computation `c` that takes values of type `a` as input and produces values of type `b` as output." Arrows allow us to implement fixpoint combinators without having to refer to a specific type of fixpoint computation. They are particularly useful for implementing big-step fixpoint algorithms, because they cleanly separate the inputs of an effectful computation from the outputs. Moreover, they have proven useful for modularizing other parts of the abstract interpreter [Keidel and Erdweg 2019, 2020; Keidel et al. 2018].

Refactoring the fixpoint algorithm. Based on these principles, we now refactor the fixpoint algorithm $\texttt{fix}_{\texttt{monolithic}}$ into three reusable combinators $\varphi_{\texttt{innermost}}$, $\varphi_{\texttt{filter}}$, and $\varphi_{\texttt{stackWiden}}$.

The combinator $\varphi_{\texttt{innermost}}$ is a stripped down version of the $\texttt{fix}_{\texttt{monolithic}}$ algorithm and only satisfies *Condition 1* and *3*.

```
1  φinnermost :: ... ⇒ c a b → c a b
2  φinnermost f = proc call → do
3    (stable,resultcached) ←Cache.lookup≺ call
4    if stable
5    then return ≺ resultcached
6    else do
7      recurrentCall ← Stack.elem ≺ call
8      if recurrentCall then do
9        SCG.add ≺ call
10       return ≺ resultcached
11     else iterate f ≺ call
12
13
14
15
16
```

```
17 iterate :: ... ⇒ c a b → c a b
18 iterate f = proc call → do
19   resultnew ← Stack.push f ≺ call
20   callInSCG ← SCG.elem ≺ call
21   if callInSCG then do
22     (grown,resultwidened)
23       ← Cache.update ≺ (call,resultnew)
24     if grown then iterate f ≺ call
25     else do
26       sizeSCG ← SCG.size ≺ ()
27       if sizeSCG == 1
28         then Cache.setStable ≺ call
29         else return ≺ ()
30       SCG.remove ≺ call
31       return ≺ resultwidened
32   else return ≺ resultnew
```

The combinator is parameterized by operations to access and modify the stack, cache and SCG contained in the effectful arrow computation. Furthermore, the code uses the following arrow notation: The keyword `proc x` introduces a new arrow computation that binds its argument to the variable `x`. The syntax `y ← f ≺ x` calls an arrow computation `f` with the argument `x` and binds the result to the variable `y`. Lastly, the keyword `return ≺ x` returns `x` as result of the arrow computation, but does not exit the surrounding `proc` like regular returns.

The combinator $\varphi_{\texttt{innermost}}$ first looks up the call in the cache. If the cached result is stable, the combinator simply returns the cached entry (line 5). Otherwise, the combinator looks up the call on the stack (line 7). In case of a recurrent call (line 8), the algorithm adds the call to the SCG and returns the cached entry. Otherwise, if the call did not appear on the stack (line 11), the algorithm calls the recursive helper function `iterate` that updates the analysis result until it does not grow anymore. The function `iterate` first calls the computation `f` while adding the current call to the stack (line 19). Afterwards, it checks if the call occurred in the SCG (line 20) and hence needs to be iterated on. If the call occurred in the SCG, function `iterate` updates the cache with the new result (line 23). The operation `Cache.update` simultaneously updates the cache, widens the new

result against an existing entry and checks if the result is stable or has grown. If the analysis result has grown the function iterates again (line 24). Otherwise, it removes the call from the SCG and returns the widened result (line 31). If the SCG consist of only a single element, i.e. the current call, then the current result only depends on other stable analysis results.

To address *Condition 2*, we implement a fixpoint combinator that applies a widening operator to the current stack and call:

```
φstackWiden :: ... ⇒ (stack → a → (stack,a)) → c a b → c a b
φstackWiden ∇Stack f = proc call → do
  stack ← Stack.ask ≺ ()
  let (stackwidened,callwidened) = stack ∇Stack call
  Stack.local f ≺ (stackwidened, callwidened)
```

Combinator $\varphi_{\texttt{stackWiden}}$ first accesses the stack contained in the arrow computation. It then applies the stack widening operator ($\nabla_{\widehat{\texttt{Stack}}}$) to this stack and current call. Afterwards, it passes the widened call to the computation `f` and sets the new stack.

Lastly, the higher-order fixpoint combinator $\varphi_{\texttt{filter}}$, inspired by Wei et al. [2019] `fix_select`, filters out calls not relevant to the rest of the fixpoint algorithm:

```
φfilter :: ... ⇒ (a → Boolean) → (c a b → c a b) → (c a b → c a b)
φfilter predicate φ f = proc call →
  if predicate call
  then φ f ≺ call
  else f ≺ call
```

The combinator $\varphi_{\texttt{filter}}$ either calls the combinator φ whenever the predicate holds, or skips the combinator φ when the predicate does not hold.

With these three fixpoint combinators, we can recreate the fixpoint algorithm $\texttt{fix}_{\texttt{monolithic}}$ from the previous section:

$$\texttt{fix}_{\texttt{monolithic}} = \texttt{fix}_{\varphi}\ (\widehat{\lambda\texttt{eval}}_{\texttt{rec}} \ldots) \qquad \varphi = \varphi_{\texttt{filter}}\ \texttt{isFunctionBody}\ (\varphi_{\texttt{stackWiden}}\ \nabla_{\widehat{\texttt{Stack}}} \circ \varphi_{\texttt{innermost}})$$

The execution order of the combinators is from outside inwards: First $\varphi_{\texttt{filter}}$ gets control, then $\varphi_{\texttt{stackWiden}}$, then $\varphi_{\texttt{innermost}}$, and finally $(\widehat{\lambda\texttt{eval}}_{\texttt{rec}} \ldots)$ before the cycle repeats.

To summarize, in this section we proposed a modular description of fixpoint algorithms. In particular, we describe fixpoint algorithms with reusable fixpoint combinators, where each combinator captures a certain aspect of the fixpoint algorithm.

4 A LIBRARY OF REUSABLE FIXPOINT COMBINATORS

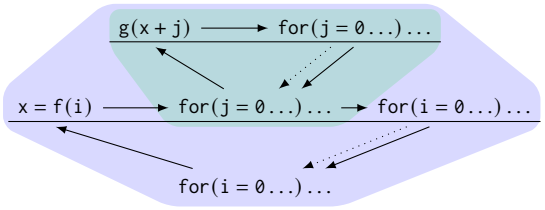
In the previous section, we described a framework for developing modular fixpoint algorithms by the means of fixpoint combinators. Based on this framework, we develop a library of reusable fixpoint combinators in this section, which serve as building blocks for fixpoint algorithms.

4.1 Iteration Strategy Combinators

An iteration strategy determines the order in which statements are analyzed. For example, the combinator $\varphi_{\texttt{innermost}}$ of Section 3 iterates on the innermost SCGs of the graph-shaped trace. Furthermore, an iteration strategy cuts off recurrent recursive calls to enforce termination, at the cost of precision.

Iterating on the outermost SCGs. For some programs, it can be faster to iterate on the outer SCGs first. For example, consider the analysis of the follow program:

```
for(i = 0; i < m; i++) {
  x = f(i);
  for(j = 0; j < n; j++) {
    g(x + j);
  }
}
```



If `f(i)` yields the analysis result $\top$ after the second iteration of the outer loop, then we waste analysis time trying to analyze `g(x + j)` precisely in the first iteration of the inner loop.

We implement this iteration strategy with the fixpoint combinator $\varphi_{\texttt{outermost}}$:

```
1  φ_outermost :: ... ⇒ c a b → c a b
2  φ_outermost = ... iterate ...
3
4  iterate = proc call → do
5    result_new ← Stack.push f ≺ call
6    callInSCG ← SCG.elem ≺ call
7    sizeSCG ← SCG.size ≺ ()
8
9    if callInSCG && sizeSCG == 1 then do
10   (grown,result_widened)
11     ← Cache.update ≺ (call,result_new)
12   if grown then iterate f ≺ call
13   else do
14     SCG.remove ≺ call
15     return ≺ result_widened
16 else return ≺ result_new
```

The combinator $\varphi_{\texttt{outermost}}$ is similar to $\varphi_{\texttt{innermost}}$, except that it returns to the head of the outermost subgraph before iterating. The combinator identifies heads of the outermost subgraph by checking that the size of the SCG set is 1 (line 9). This check works because the combinator removes each call from the SCG set when returning (line 14) and if this set has only a single call left, this call must be the head of an outermost SCG.

Iterating on the topmost call. The combinators $\varphi_{\texttt{innermost}}$ and $\varphi_{\texttt{outermost}}$ iterate on the SCGs of the graph-shaped trace of the abstract interpreter. However, calculating the SCGs induces an overhead, which slows down the analysis especially for programs, that only consists of a single large SCG. In these cases, it can be faster to only iterate on the topmost call of the abstract interpreter. We implement this iteration strategy with the following fixpoint combinator, which is inspired by an existing big-step fixpoint algorithm [Darais et al. 2017]:

```
1  φ_topmost f = proc call → do
2    callInCache ← Cache.elem ≺ call
3    if callInCache then do
4      (_,result) ← Cache.lookup ≺ call
5      return ≺ result
6    else do
7      Cache.initialize ≺ call
8      iterate f ≺ call
9
10
11
12

13 iterate f = proc call → do
14   res_new ← f ≺ call
15   res_widened ← Cache.update ≺ (call,res_new)
16   topmost ← isTopmostCall ≺ call
17   if topmost then do
18     stable ← Cache.isStable ≺ ()
19     if stable then return ≺ res_widened
20     else do
21       call_widened ←
22         Cache.nextIteration ≺ call
23       iterate ≺ call_widened
24   else return ≺ result_widened
```

The combinator $\varphi_{\texttt{topmost}}$ neither requires a stack nor SCGs. Instead, it uses the cache to detect recurrent calls (line 2). On the topmost call of the abstract interpreter, the combinator compares the cache of the current iteration to the cache of the previous iteration (line 18). If the cache of the current iteration has grown, the combinator keeps iterating with a new empty cache (line 22). The

`nextIteration` operation additionally widens the call, which allows data like a monotone store to be passed along between iterations.

Mixing iteration strategies. Our case studies show that it is difficult to find a single iteration strategy that works best for all programs (Section 6.3). To this end, the following combinator mixes two iteration strategies φ_1 and φ_2 on a case-by-case basis:

```
φalternative :: ... ⇒ c a Boolean → (c a b→c a b) → (c a b→c a b) → (c a b→c a b)
φalternative predicate φ1 φ2 f = proc call → do
  b ← predicate ≺ call
  if b
  then φ1 f ≺ call
  else φ2 f ≺ call
```

The predicate is an effectful computation, which allows it to dynamically adapt the iteration strategy. For example, the predicate may first try both strategies φ_1 and φ_2 once to decide afterwards based on collected performance metrics.

4.2 Recursion Depth Combinators

Recursion depth operators control how deep an abstract interpreter recurses on a program. For example, the combinator $\varphi_{\texttt{stackWiden}}$ of Section 3 limits the recursion depth to be finite by widening calls on the stack.

Call-site context sensitivity. A popular technique to limit the recursion depth of the abstract interpreter is to join all calls with the same k-truncated call string at the cost of precision [Shivers 1991]. We implement this technique with the following combinator:

```
1  φcallsiteSensitive :: ... ⇒ Int → c a b → c a b
2  φcallsiteSensitive k f = proc call → do
3    stack ← Stack.ask ≺ ()
4    let ctx = truncate k (getLabels stack)
5    call_cached ← Context.lookup ≺ ctx
6    if call ⊑ call_cached then
7      f ≺ call_cached
8    else do
9      call_widenend ← Context.update ≺ (ctx,call)
10     f ≺ call_widenend
```

The combinator $\varphi_{\texttt{callsiteSensitive}}$ uses a context cache that remembers the call for the current context (line 5). If the context cache contains a larger call than the current call, the combinator calls computation `f` with the cached call (line 7). Otherwise, the operation `Cache.update` widens the current and cached call and calls computation `f` with the widened call (line 10). Note that we do not need to change the analysis itself to integrate call-site sensitivity, unlike Shivers [1991]. Instead, we simply add this combinator to the fixpoint algorithm.

Stack unfolding. The following combinator improves precision by unfolding the first few calls on the stack to prevent joining:

```
φunfold :: ... ⇒ Int → (c a b → c a b) → (c a b → c a b)
φunfold limit φ f = proc call → do
  n ← Stack.size ≺ ()
  if n ≤ limit then f ≺ call
  else φ f ≺ call -- Stack size exceeds limit
```

The combinator $\varphi_{\texttt{unfold}}$ recursively calls computation `f` as long as the stack size is below a certain limit and falls back to the combinator φ if the stack size exceeds the limit. We can integrate this combinator into a fixpoint algorithm by applying it to a stack widening combinator ($\varphi_{\texttt{unfold}}$ 10 ($\varphi_{\texttt{stackWiden}}$ $\nabla_{\texttt{Stack}}$)). This prevents joining of the stack for the first 10 recursive calls.

Loop unrolling. Another common technique to improve precision is to unroll the first few iterations of a loop to prevent joining [Mauborgne and Rival 2005]. Big-step abstract interpreters analyze the same loop statement multiple times recursively:

```
eval = fix_φ (λeval_rec → proc statement → case statement of
  WhileLoop condition body → eval_rec ≺ If condition (Sequence body (WhileLoop condition body))
    ...)
```

With this observation, we implement a combinator that only joins after the same call appeared a certain number of times on the stack:

```
φ_unroll :: ... ⇒ Int → (c a b → c a b) → (c a b → c a b)
φ_unroll limit φ f = proc call → do
  n ← Stack.getCallCount ≺ call
  if n ≤ limit then do
    Stack.incrementCallCount ≺ call
    f ≺ call
  else φ f ≺ call -- Call count exceeds limit
```

Similar to the previous combinator, we can integrate this combinator by applying it to a stack widening combinator: $\varphi_{\texttt{unroll}}\ 10\ (\varphi_{\texttt{stackWiden}}\ \nabla_{\texttt{Stack}})$

4.3 Tracing Combinators

Tracing combinators run alongside the main fixpoint algorithm to record auxiliary data like a trace, without affecting precision.

Recording a control-flow graph. The following tracing combinator records a control-flow graph (CFG) of the program, which describes the order in which statements are evaluated:

```
φ_CFG f = proc call → do
  predecessor ← getPredecessor ≺ ()
  CFG.addEdge ≺ (predecessor, call)
  withNewPredecessor f ≺ call
```

Since the control-flow of a program is encoded implicitly in the big-step abstract interpreter, all the combinator needs to do is to add an edge to the CFG between the most recently evaluated call `predecessor` and the active `call`. Afterwards, the combinator passes control to computation `f`, remembering the active call as new predecessor.

We can integrate this combinator into an existing fixpoint algorithm by adding it to the front ($\varphi_{\texttt{CFG}} \circ \varphi_{\texttt{filter}}$ `isFunctionBody` ($\ldots \varphi_{\texttt{innermost}} \ldots$)). In this case, the CFG contains all statements as nodes. We can control the granularity of the CFG by changing the position of the $\varphi_{\texttt{CFG}}$ combinator ($\varphi_{\texttt{filter}}$ `isFunctionBody` ($\varphi_{\texttt{CFG}} \circ \ldots \varphi_{\texttt{innermost}} \ldots$)). In this case, the CFG only contains function calls, in other words, the CFG is an interprodcedural call graph.

Debugging static analyses. The following tracing combinator allows debugging of analyses with a graphical debugger [Pree 2020]:

```
1  φ_debug isBreakpoint f = proc call → do
2    if isBreakpoint call then do
3      cfg ← CFG.get ≺ ()
4      stack ← Stack.elems ≺ ()
5      command ← Client.send ≺ (call,cfg,stack)
6      case command of
7        Step → breakAtNextStatement ≺ ()
8        ...
9    f ≺ call
```

The fixpoint combinator runs as a server within the fixpoint algorithm and sends information to a graphical debugging client in a browser. On a break point the combinator sends the current stack

and the CFG to the debugging client (line 5). After the client returned a debugging command, the combinator executes the command and calls computation `f`. This resumes the analysis until the combinator hits the next break point.

5 MODULAR SOUNDNESS PROOFS OF BIG-STEP FIXPOINT ALGORITHMS

In this section, we develop a formal theory to prove soundness of big-step fixpoint algorithms that consist of fixpoint combinators. In particular, we prove that a modular fixpoint algorithm is sound if all of its combinators are sound.

To this end, we first review a soundness proof of monolithic fixpoint algorithms, which we later extend for modular algorithms:

Proposition 5.1 (Soundness of Monolithic Fixpoint Algorithms [Cousot and Cousot 1992]). *Let* $\widehat{\texttt{eval}} : \widehat{D} \to \widehat{D}$ *be an abstract interpreter and* $\texttt{eval} : D \to D$ *be the monotone collecting semantics of the concrete interpreter over two complete lattices* $(\widehat{D}, \sqsubseteq, \sqcup)$ *and* $(D, \sqsubseteq, \sqcup)$*. Furthermore, let* $\gamma : \widehat{D} \to D$ *be a monotone concretization function such that* $\forall x \in D.\, \widehat{x} \in \widehat{D}.\, x \sqsubseteq \gamma(\widehat{x}) \implies \texttt{eval}(x) \sqsubseteq \gamma(\widehat{\texttt{eval}}(\widehat{x}))$*. A fixpoint algorithm for the abstract interpreter is sound if it yields a post-fixpoint of* $\widehat{\texttt{eval}}$*, i.e. an element* $\widehat{p} \in \widehat{D}$ *with* $\widehat{\texttt{eval}}(\widehat{p}) \sqsubseteq \widehat{p}$*.*

Proof.

$$
\begin{aligned}
& \widehat{\texttt{eval}}(\widehat{p}) \sqsubseteq \widehat{p} && (\widehat{p} \text{ is a post-fixpoint of } \widehat{\texttt{eval}}) \\
\implies & \gamma(\widehat{\texttt{eval}}(\widehat{p})) \sqsubseteq \gamma(\widehat{p}) && (\gamma \text{ is monotone}) \\
\implies & \texttt{eval}(\gamma(\widehat{\texttt{eval}}(\widehat{p}))) \sqsubseteq \gamma(\widehat{\texttt{eval}}(\widehat{p})) && (\widehat{\texttt{eval}} \text{ sound w.r.t. } \texttt{eval}) \\
\implies & \text{lfp}(\texttt{eval}) \sqsubseteq \gamma(\widehat{\texttt{eval}}(\widehat{p})) && (\text{Tarski's fixpoint theorem [Tarski 1955]})
\end{aligned}
$$

□

This proposition requires that the collecting semantics of the concrete interpreter is monotone, such that the least fixpoint $\text{lfp}(\texttt{eval})$ exists. Monotonicity of the collecting semantics follows directly by Scott-continuity of the denotational semantics of the concrete interpreter [Streicher 2006].

We build on this idea to develop a soundness composition theorem for modular big-step fixpoint algorithms. Let us first assume that the modular fixpoint algorithm is built from fixpoint combinators over the following grammar:

$$
\begin{aligned}
\varphi ::= {} & \varphi \text{ atomic} && \text{(atomic combinators)} \\
\mid {} & \varphi \circ \varphi && \text{(combinator composition)} \\
\mid {} & \varphi(\varphi \ldots \varphi) && \text{(higher-order combinators)}
\end{aligned}
$$

This grammar allows us to formulate the following soundness lemmas for each type of combinator:

$$
\begin{aligned}
\text{atomic } \varphi \textit{ sound} \quad & \text{if} \quad \forall \text{ monotone } \widehat{f}.\ \forall \widehat{x}.\ \varphi(\widehat{f}(\widehat{x})) \sqsubseteq \widehat{x} \implies \widehat{f}(\widehat{x}) \sqsubseteq \widehat{x} \\
\varphi_1 \circ \varphi_2 \textit{ sound} \quad & \text{if} \quad \varphi_1 \textit{ sound} \wedge \varphi_2 \textit{ sound} \\
\text{higher-order } \varphi \textit{ sound} \quad & \text{if} \quad \forall \varphi_1 \ldots \varphi_n.\ \varphi_1 \textit{ sound} \wedge \ldots \varphi_n \textit{ sound} \\
& \qquad \implies \varphi(\varphi_1 \ldots \varphi_n) \textit{ sound}
\end{aligned}
$$

That is, an atomic fixpoint combinator φ is sound if all post-fixpoints of $\varphi \circ \widehat{f}$ are also post-fixpoints of $\widehat{f}$, for any $\widehat{f}$. All other types of combinator preserve this post-fixpoint property.

These soundness lemmas allow us to prove soundness of modular fixpoint algorithms once and for all with the following theorem:

Theorem 5.2 (Soundness of Modular Fixpoint Algorithms). *A modular fixpoint algorithm* $\mathtt{fix}_\varphi(\widehat{\mathtt{eval}})$ *is sound, if all of its combinators* φ *are sound.*

Proof. We prove by structural induction over φ that $\forall \widehat{f}, \widehat{x}.\, \varphi(\widehat{f}(\widehat{x})) \sqsubseteq \widehat{x} \implies \widehat{f}(\widehat{x}) \sqsubseteq \widehat{x}$. By definition of fix, we get $\varphi(\widehat{\mathtt{eval}}(\mathtt{fix}_\varphi(\widehat{\mathtt{eval}}))) = \mathtt{fix}_\varphi(\widehat{\mathtt{eval}})$, which satisfies the precondition above. We conclude $\widehat{\mathtt{eval}}(\mathtt{fix}_\varphi(\widehat{\mathtt{eval}})) \sqsubseteq \mathtt{fix}_\varphi(\widehat{\mathtt{eval}})$, which shows that $\mathtt{fix}_\varphi(\widehat{\mathtt{eval}})$ is a post-fixpoint of $\widehat{\mathtt{eval}}$. Thus the fixpoint algorithm $\mathtt{fix}_\varphi(\widehat{\mathtt{eval}})$ is sound by Proposition 5.1. □

This way of proving soundness of modular fixpoint algorithms is more flexible than a monolithic proof because it allows us to reorder and add new combinators without invalidating the soundness proof.

5.1 Soundness Proof Strategies for Fixpoint Combinators

In this subsection, we prove three theorems that guarantee soundness of three classes of combinators. The soundness proofs are split into two parts: One part proves soundness of classes of combinators that satisfy certain properties (Theorem 5.3 and 5.4) and one part shows combinator implementations actually satisfy these properties (Corollary 5.4 and 5.6).

Extensive Fixpoint Combinators. A fixpoint combinator φ is extensive iff for all monotone $\widehat{f}$ and for all $\widehat{x}$, it holds $\widehat{f}(\widehat{x}) \sqsubseteq \varphi(\widehat{f}(\widehat{x}))$. This is the case if $\varphi(\widehat{f}(\widehat{x})) \prec a$ calls $\widehat{f}(\widehat{x}) \prec a'$ with a greater argument $a' \sqsupseteq a$ such that $\widehat{f}(\widehat{x}) \prec a \sqsubseteq \widehat{f}(\widehat{x}) \prec a' \sqsubseteq \varphi(\widehat{f}(\widehat{x})) \prec a$.

Theorem 5.3. *An extensive fixpoint combinator* φ *is sound.*

Proof. We assume $\varphi(\widehat{f}(\widehat{x})) \sqsubseteq \widehat{x}$ for all monotone $\widehat{f}$ and for all $\widehat{x}$. By extensivity, we conclude $\widehat{f}(\widehat{x}) \sqsubseteq \varphi(\widehat{f}(\widehat{x})) \sqsubseteq \widehat{x}$. □

Corollary 5.4. *Combinators* $\varphi_{\mathtt{stackWiden}}$ *and* $\varphi_{\mathtt{CFG}}$ *are sound.*

Proof. By Theorem 5.3 it suffices to show that the combinators are extensive. Combinator $\varphi_{\mathtt{stackWiden}}(\nabla_{\widehat{\mathtt{Stack}}})(\widehat{f})$ is extensive because it calls $\widehat{f}$ with an upper bound of the current input. Calling $\varphi_{\mathtt{stackWiden}}(\nabla_{\widehat{\mathtt{Stack}}})(\widehat{f})$ is greater than just calling $\widehat{f}$ by monotonicity of $\widehat{f}$. Hence it follows $\widehat{f} \sqsubseteq \varphi_{\mathtt{stackWiden}}(\nabla_{\widehat{\mathtt{Stack}}})(\widehat{f})$. Combinator $\varphi_{\mathtt{CFG}}(\widehat{f})$ is extensive because it calls function $\widehat{f}$ with the same argument it was supplied with. □

Interleaving Fixpoint Combinators. A higher-order fixpoint combinator $\varphi(\varphi_1, \varphi_2)(\widehat{f}(\widehat{x}))$ is interleaving iff $\varphi_1(\widehat{f}(\widehat{x})) \sqsubseteq \varphi(\varphi_1, \varphi_2)(\widehat{f}(\widehat{x}))$ or $\varphi_2(\widehat{f}(\widehat{x})) \sqsubseteq \varphi(\varphi_1, \varphi_2)(\widehat{f}(\widehat{x}))$. This is the case if φ either calls combinator φ_1 or calls combinator φ_2. This observation leads us to our second theorem:

Theorem 5.5 (Soundness of Interleaving Fixpoint Combinators). *Let* φ *be a higher-order fixpoint combinator, such that for all* φ_1, φ_2*, it holds* $\forall x. \varphi_1(x) \sqsubseteq \varphi(\varphi_1, \varphi_2)(x) \vee \varphi_2(x) \sqsubseteq \varphi(\varphi_1, \varphi_2)(x)$*, then* φ *is sound.*

Proof. We assume $\varphi(\varphi_1, \varphi_2)(\widehat{f}(\widehat{x})) \sqsubseteq \widehat{x}$ for a given $\widehat{f}, \widehat{x}$. We assume that both φ_1 and φ_2 are sound by the soundness definition of the higher-order combinator φ. If $\varphi_1(\widehat{f}(\widehat{x})) \sqsubseteq \varphi(\varphi_1, \varphi_2)(\widehat{f}(\widehat{x}))$, then $\widehat{f}(\widehat{x}) \sqsubseteq \widehat{x}$ follows immediately by soundness of φ_1. The other case is analogous. □

Corollary 5.6. *The combinators* $\varphi_{\mathtt{alternative}}$, $\varphi_{\mathtt{filter}}$ *and* $\varphi_{\mathtt{unroll}}$ *are sound by Theorem 5.5.*

Proof. By Theorem 5.5, it suffices to show that the combinators are interleaving. The combinator $\varphi_{\mathtt{alternative}}(P, \varphi_1, \varphi_2)(\widehat{f}(\widehat{x}))$ either calls $\varphi_1(\widehat{f}(\widehat{x}))$ or $\varphi_2(\widehat{f}(\widehat{x}))$ depending on the predicate P. Hence $\varphi_1(\widehat{f}(\widehat{x})) \sqsubseteq \varphi_{\mathtt{alternative}}(P, \varphi_1, \varphi_2)(\widehat{f}(\widehat{x}))$ or $\varphi_2(\widehat{f}(\widehat{x})) \sqsubseteq \varphi_{\mathtt{alternative}}(P, \varphi_1, \varphi_2)(\widehat{f}(\widehat{x}))$. Furthermore, combinator $\varphi_{\mathtt{filter}}$ is interleaving because $\varphi_{\mathtt{filter}}(P, \varphi_1) = \varphi_{\mathtt{alternative}}(P, \varphi_1, \varphi_{\mathtt{id}})$,

where $\varphi_{\texttt{id}}(\widehat{f}(\widehat{x})) \prec a = \widehat{f}(\widehat{x}) \prec a$ is the identity combinator. Finally, $\varphi_{\texttt{unroll}}$ is interleaving because $\varphi_{\texttt{unroll}}(n, \varphi_2) = \varphi_{\texttt{alternative}}(P_n, \varphi_{\texttt{id}}, \varphi_2)$, where predicate P_n is true after n recursive calls. □

Cache-Based Fixpoint Combinators. Lastly, we prove soundness of the cache-based fixpoint combinators like $\varphi_{\texttt{innermost}}$:

Theorem 5.7. *The fixpoint combinator $\varphi_{\texttt{innermost}}$ is sound.*

Proof. We assume $\varphi_{\texttt{innermost}}(\widehat{f}(\widehat{x})) \sqsubseteq \widehat{x}$ for all monotone $\widehat{f}$ and for all $\widehat{x}$ and $\texttt{call}$.

The key insight is that unstable intermediate results only occur within SCGs and $\varphi_{\texttt{innermost}}$ does not return until all calls within the SCG have reached a post-fixpoint. To this end, we first prove that either $\widehat{f}(\widehat{x}) \prec \texttt{call} \sqsubseteq \widehat{x} \prec \texttt{call}$ or that the $\texttt{call}$ appears on the stack and in the resulting SCG set.

- If $\texttt{result}_{\texttt{cached}}$ is unstable and the $\texttt{call}$ does not occur on the stack, combinator $\varphi_{\texttt{innermost}}$ iterates until $\texttt{result}_{\texttt{new}}$ does not grow anymore (line 24 in $\varphi_{\texttt{innermost}}$) and $\texttt{result}_{\texttt{new}} \sqsubseteq \texttt{result}_{\texttt{cached}}$. In this case, combinator $\varphi_{\texttt{innermost}}$ removes the $\texttt{call}$ from the SCG set and returns $\texttt{result}_{\texttt{widened}}$ (line 31). By the assumption, it follows that $\texttt{return} \prec \texttt{result}_{\texttt{widened}} \sqsubseteq \widehat{x} \prec \texttt{call}$. By transitivity we conclude $\widehat{f}(\widehat{x}) \prec \texttt{call} \sqsubseteq \texttt{return} \prec \texttt{result}_{\texttt{new}} \sqsubseteq \texttt{return} \prec \texttt{result}_{\texttt{widened}} \sqsubseteq \widehat{x} \prec \texttt{call}$.
- If $\texttt{result}_{\texttt{cached}}$ is unstable and the call occurs on the stack, combinator $\varphi_{\texttt{innermost}}$ adds the call to the SCG (line 9).
- If $\texttt{result}_{\texttt{cached}}$ is stable, combinator $\varphi_{\texttt{innermost}}$ simply returns the cached result (line 5). By the assumption, it follows that $\texttt{return} \prec \texttt{result}_{\texttt{cached}} \sqsubseteq \widehat{x} \prec \texttt{call}$. Furthermore, the $\texttt{Cache.update}$ operation only marks a result as stable if $\widehat{f}(\widehat{x}) \prec \texttt{call} \sqsubseteq \texttt{return} \prec \texttt{result}_{\texttt{cached}}$, as explained in the previous case. By transitivity we conclude $\widehat{f}(\widehat{x}) \prec \texttt{call} \sqsubseteq \texttt{return} \prec \texttt{result}_{\texttt{cached}} \sqsubseteq \widehat{x} \prec \texttt{call}$.

When we run a fixpoint algorithm including $\varphi_{\texttt{innermost}}$, we initialize the stack to be empty. This means that $\texttt{call}$ cannot appear on the stack and hence $\widehat{f}(\widehat{x}) \prec \texttt{call} \sqsubseteq \widehat{x} \prec \texttt{call}$ has to be true. □

Theorem 5.8. *The fixpoint combinator $\varphi_{\texttt{outermost}}$ is sound.*

Proof. The argument for $\varphi_{\texttt{outermost}}$ is similar to $\varphi_{\texttt{innermost}}$. The only difference is that $\varphi_{\texttt{outermost}}$ waits until the SCG contains only a single element. But this does not change that it only returns from an SCG until all calls within reached a post-fixpoint. □

To summarize, in this section we presented a way to prove soundness of fixpoint algorithms that consist of fixpoint combinators. In particular, a fixpoint algorithm is sound, if all of its combinators are sound. This simplifies the soundness proof, as it suffices to prove each combinator sound individually. Furthermore, we proved soundness of three classes of fixpoint combinators.

6 CASE STUDIES

In this section, we evaluate the feasibility of our framework for modular fixpoint algorithms. In particular, we integrated the combinators of Section 4 into the Sturdy library [Keidel and Erdweg 2019; Keidel et al. 2018] and used them to develop fixpoint algorithms for the following analyses:

- A static type analysis [Keidel and Erdweg 2020] for Stratego [Visser et al. 1998], a domain-specific dynamically-typed language for program transformations,
- a dead-code constant-propagation analysis for WebAssembly, a low-level bytecode that runs in the browser [Haas et al. 2017],
- a k-CFA [Shivers 1991] for Scheme [Abelson et al. 1998], a dynamically-typed programming language with first-class functions and mutable state.

The goal of these case studies is to asses the language and analysis-independence, the precision, and the performance of the fixpoint combinators.

6.1 Static Type Analysis for Stratego

In our first case study, we implemented a fixpoint algorithm for a static type analysis [Keidel and Erdweg 2020] for Stratego [Visser et al. 1998], a domain-specific dynamically-typed language for developing program transformations. Stratego is difficult to type statically because of features like generic program traversals that temporarily produce ill-typed programs.

The abstract interpreter takes a Stratego program called a *strategy*, a strategy environment, a term environment, and a program term that is transformed. Furthermore, the abstract interpreter returns as output a list of errors, an updated term environment and resulting term.

```
eval^ :: ... ⇒ c (Strategy, StratEnv^, TermEnv^, Term^) (Errors^, Either^ () (TermEnv^, Term^))
eval^ = fix_(φfilter isStrategyBody (φstackWiden ∇Stack^ ∘ φoutermost)) ...
```

If the strategy fails to match a term pattern, the abstract interpreter returns the empty tuple instead. Type-checking Stragego program transformations is not trivial since generic program traversals may produce intermediate terms that are ill-sorted. To solve this, the analysis uses an abstract domain that is able to represent ill-sorted terms:

```
Term^ = Sorted Sort | MaybeSorted (Powerset (Constructor,Term^))
```

This abstract domain is infinite since abstract terms can grow infinitely deep. To this end, a widening operator cuts-off terms at a specified depth, trying to type check deeper terms to determine their sort.

The fixpoint algorithm uses the outermost iteration strategy and applies a stack widening operator because the abstract term domain and term environment are infinite. The stack widening operator replaces the current call with the topmost call on the stack that is greater. To debug the analysis during development we added a tracing combinator $\varphi_{\texttt{trace}}$ within the $\varphi_{\texttt{filter}}$ expression to print a trace of analyzed strategies and their abstract term arguments. Furthermore, we occasionally moved the tracing combinator to the outside of the $\varphi_{\texttt{filter}}$ expression for a more fine grained trace that contains all substrategies as well.

We tested the abstract interpreter and its fixpoint algorithm on a test suite with 61 test cases including 3 existing program transformations: an desugaring of arrows (665 lines of code)[7] [Paterson 2001], a normalization of arrows to causal commutative normal form (490 loc) [Liu et al. 2009], and an interpreter for PCF (61 loc) [Plotkin 1977]. We found that in all of these test cases the abstract interpreter terminates with a sound analysis result. Furthermore, the results where precise enough to validate the well-typedness of the 3 program transformations.

This case study shows that the modular fixpoint algorithm is precise enough to yield usable analysis results.

6.2 Dead-Code Constant-Propagation Analysis for WebAssembly

In our second case study, we implemented a fixpoint algorithm for a dead-code constant-propagation analysis for WebAssembly (Wasm), a low-level bytecode that runs in the browser [Haas et al. 2017]. This analysis can be used to reduce the size of the executables that are sent to the browser.

The abstract interpreter takes as input a list of instructions, a list of return types, a module instance, an operand stack, a frame of local variables, function tables, module memories, a global state and a set of errors:

```
eval^ :: ... ⇒ c ([Instr],[Type],ModuleInst,OperandStack^,Frame^,Tables^,Memories^,GlobalState^,Errors^)
                 (OperandStack^,Frame^,Memories^,GlobalState^,Errors^)
eval^ = fix_((φCFG ∘ φfilter isLoopOrCall φinnermost) ...
```

[7] Counted with `wc -l` because the standard tool `cloc` does not support Stratego code.

The abstract interpreter abstracts values with a finite constant abstract domain, i.e., abstract values are either 32 or 64-bit integers, 32 or 64-bit floating point numbers, or $\top$. Furthermore, operand stack and call frame are abstracted with lists of abstract values, because their shape is statically known [Haas et al. 2017]. Moreover, the linear memory is abstracted with a byte-indexed vector of abstract values.

The fixpoint algorithm uses the $\varphi_{\texttt{innermost}}$ iteration strategy and applies it to loops and function calls. The combinator $\varphi_{\texttt{CFG}}$ records an inter-procedural control-flow graph (CFG) which allows us to find dead-code, i.e., code that will never be executed. In particular, we remove all instructions that do not appear in the CFG because they cannot be executed. This works because the control-flow of the abstract interpreter overapproximates the control-flow of the concrete interpreter and an instruction not analyzed by the abstract interpreter cannot be executed by the concrete interpreter.

Our Wasm analysis and the fixpoint combinators have been reimplemented in Scala [Brandl et al. 2023]. The analysis has been evaluated on 1458 binaries of the WasmBench benchmark suite [Hilbig et al. 2021]. With a timeout of 60 seconds, the dead-code constant-propagation analysis terminates on average in 5s and eliminates 20% of the program code. In contrast, the industry-standard Binaryen terminates on average in 0.1s, but only eliminates 9% of the program code.

This case study demonstrates two points:

- Our fixpoint combinators are meta-language, object-language, and analysis independent.
- Our fixpoint combinators scale to develop fixpoint algorithms for real-world languages.

6.3 k-CFA for Scheme

For our third case study, we implemented an inter-procedural control-flow analysis (k-CFA) [Shivers 1991] and static type analysis for Scheme [Abelson et al. 1998], a dynamically-typed real-world programming language with first-class functions and mutable state. The abstract interpreter takes as input a list of expression, an environment, store and errors and returns as output an abstract value, store and errors:

$\widehat{\texttt{Env}} = \texttt{Map String } \widehat{\texttt{Addr}} \qquad \widehat{\texttt{Store}} = \texttt{Map } \widehat{\texttt{Addr}}\ \widehat{\texttt{Val}} \qquad \widehat{\texttt{Errors}} = \mathcal{P}(\texttt{String})$

$\widehat{\texttt{eval}} :: \ldots \Rightarrow \texttt{c ([Expr], } \widehat{\texttt{Env}}\texttt{, } \widehat{\texttt{Store}}\texttt{, } \widehat{\texttt{Errors}}\texttt{) (}\widehat{\texttt{Val}}\texttt{, } \widehat{\texttt{Store}}\texttt{, } \widehat{\texttt{Errors}}\texttt{)}$
$\widehat{\texttt{eval}} = \texttt{fix}_{(\varphi_{\texttt{filter}}\ \texttt{isApplication}\ (\varphi_{\texttt{recordCallSite}}\ k)\ \circ\ \varphi_{\texttt{filter}}\ \texttt{isFunctionBody}\ \varphi_{\texttt{topmost}})}\ \cdots$

The abstract domain includes a set-based abstraction for closures and quoted symbols, a shape abstraction for lists, a constant abstraction for booleans, and type-based abstraction for all other datatypes:

$\widehat{\texttt{Val}} = \top \mid \texttt{ClosureVal (Powerset (Expr,}\widehat{\texttt{Env}}\texttt{))} \mid \texttt{ListVal } \widehat{\texttt{List}} \mid \texttt{BoolVal } \widehat{\texttt{Bool}} \mid \texttt{NumVal } \widehat{\texttt{Num}} \mid \ldots$
$\widehat{\texttt{List}} = \texttt{Nil} \mid \texttt{Cons (Powerset } \widehat{\texttt{Addr}}\texttt{) (Powerset } \widehat{\texttt{Addr}}\texttt{)} \mid \texttt{NilOrCons (Powerset } \widehat{\texttt{Addr}}\texttt{) (Powerset } \widehat{\texttt{Addr}}\texttt{)}$
$\widehat{\texttt{Bool}} = \top \mid \texttt{True} \mid \texttt{False}$
$\widehat{\texttt{Num}} = \top \mid \texttt{IntVal} \mid \texttt{FloatVal}$

Even though Scheme is a dynamically-typed language, the abstract domain above typically used for statically-typed languages. Specifically, two abstract values of different types join to $\top$. This choice of abstract domain is precise enough to soundly compute the control-flow of all but one benchmark programs we discuss below, but performs better than a set-based abstraction.

As fixpoint algorithm, we first developed an initial algorithm that we later specialize. The initial fixpoint algorithm uses $\varphi_{\texttt{topmost}}$ as a baseline iteration strategy because it does not compute SCGs similar to Shivers's k-CFA. Furthermore, the algorithm uses the combinator $\varphi_{\texttt{recordCallSite}}$ to record the k most recent call sites, which we use as abstract addresses.

While the initial fixpoint algorithm terminates and is sound, it converges to a fixpoint very slowly. The reason is that the algorithm "forgets" about recent store and error updates when it

returns a cached result. To address this problem, we specialize the fixpoint algorithm to use a different cache that respects the part of the input and output that only ever grows. The following code shows the lookup operation of the cache:

```
Cache.lookup :: ... ⇒ c (a,s) (Stable,(b,s))
Cache.lookup = proc (call,monotone_new) → do
  cache ← getCache ≺ ()
  if call ∈ cache then do
    let (stable_old,result,monotone_old) = cache(call)
    let stable = if monotone_old ⊏ monotone_new then stable_old else Unstable
    return ≺ (stable, (result, monotone_new))
  else return ≺ (Unstable, (⊥, monotone_new))
```

```
MonotoneCache a b m =
    Map a (Stable,b,m)
```

The lookup operation of $\widehat{\texttt{MonotoneCache}}$ always returns the new and greater element $\texttt{monotone}_{\texttt{new}}$. The cached element $\texttt{monotone}_{\texttt{old}}$ is only kept to determine if the result is stable. Returning the old cached element $\texttt{monotone}_{\texttt{old}}$ would forget about the store and error updates.

We integrate this improvement into the initial fixpoint algorithm by replacing the cache that is contained in the arrow computation. Furthermore, we use the combinator $\varphi_{\texttt{transform}}$ to group the parts of the input and output that grow monotonically:

$$\begin{aligned}\varphi_{\texttt{transform}}\ & [([\texttt{Expr}], \widehat{\texttt{Env}}, \widehat{\texttt{Store}}, \widehat{\texttt{Errors}}) \simeq (([\texttt{Expr}], \widehat{\texttt{Env}}), (\widehat{\texttt{Store}}, \widehat{\texttt{Errors}}))] \\ & [(\widehat{\texttt{Val}}, \widehat{\texttt{Store}}, \widehat{\texttt{Error}}) \simeq (\widehat{\texttt{Val}}, (\widehat{\texttt{Store}}, \widehat{\texttt{Error}}))] \\ & (\varphi_{\texttt{filter}}\ \texttt{isApplication}\ (\varphi_{\texttt{recordCallSite}}\ k)\ \circ\ \varphi_{\texttt{filter}}\ \texttt{isFunctionBody}\ \varphi_{\texttt{topmost}})\end{aligned}$$

The combinator $\varphi_{\texttt{transform}}$ applies two isomorphisms to the inputs and outputs of the abstract interpreter such that variable `call` has type $([\texttt{Expr}], \widehat{\texttt{Env}})$ and variable $\texttt{monotone}_{\texttt{new}}$ has type $(\widehat{\texttt{Store}}, \widehat{\texttt{Error}})$ within the `Cache.lookup` operation above.

While the monotone cache improves the performance of the fixpoint algorithm, there is still room for fine-tuning the iteration strategy. In particular, we evaluate 3 different iteration strategies by analyzing Scheme programs of the *Gabriel* [Gabriel 1985] and *Scala-AM* benchmark suite [Es et al. 2019]. The Gabriel benchmark suite contains 9 Scheme files from 17 up to 562 lines of code (loc) with an average of 137 loc.[8] The Scala-AM benchmark suite contains 5 Scheme files from 10 up to 40 loc with an average of 26 loc. The analysis is precise enough to soundly analyze the control-flow of all benchmark programs, except for *dderiv* where the analysis tries to call a closure which is $\top$.

Figure 3 shows the speedups over the baseline iteration strategy $\varphi_{\texttt{topmost}}$ of our initial fixpoint algorithm. Benchmarks like *cpstak* and *diviter* have an SCG at the very top of the program. This means there is no significant difference between the iteration orders and the overhead of computing the SCGs slows down the iteration strategies $\varphi_{\texttt{innermost}}$ and $\varphi_{\texttt{outermost}}$. On other benchmarks like *destruc*, *takl*, and *rsa* the SCGs are smaller and do not span the entire program. In these cases the iteration strategies $\varphi_{\texttt{innermost}}$ and $\varphi_{\texttt{outermost}}$ get an considerable speedup. The results show that no single iteration strategy performs best for all analyzed programs and further fine-tuning is needed.

Lastly, we assess the potential performance overhead caused by the modularization of the fixpoint algorithm. In particular, we inspected the low-level code of the fixpoint algorithm generated by the Haskell compiler GHC. The GHC compiler first inlines the definitions of the fixpoint combinators and arrow computation and then optimizes the residual code. The result is a pure function close to a hand-written monolithic fixpoint algorithm, meaning that modularization has no performance penalty.

[8] Counted with `cloc` (https://github.com/AlDanial/cloc).

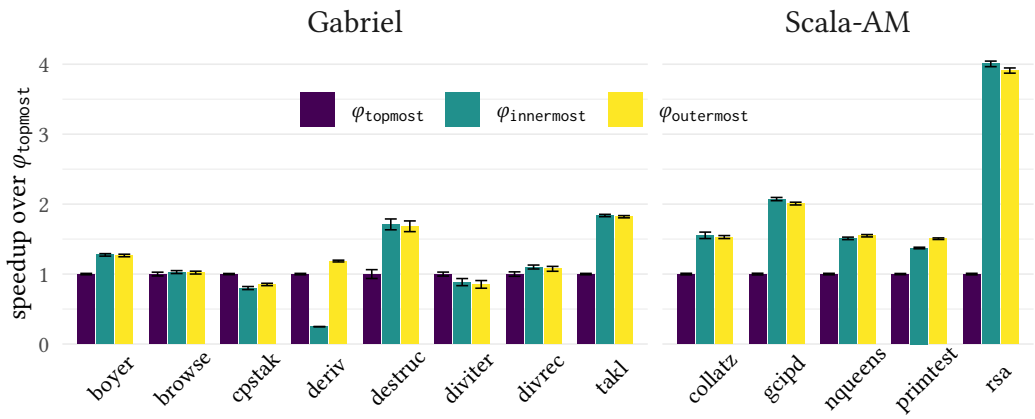


Fig. 3. Normalized running times of different iteration strategies for a 0CFA analysis for Scheme. The plot shows the speedup of each iteration strategy over the baseline $\varphi_{\texttt{topmost}}$ (higher is better). The error bars show the standard deviation of the ratio distribution for the normalized running time.

To summarize, the k-CFA case study demonstrates three points:

- We were able to specialize the initial fixpoint algorithm without changing code of existing fixpoint combinators.
- We compared the performance of 3 different iteration strategies and concluded that no iteration strategy performs best for all programs. This shows that further fine-tuning of the iteration strategy is needed.
- The Haskell compiler optimizes the modularized fixpoint algorithm by inlining and produces code close to hand-written monolithic fixpoint algorithm. Thus, modularization does not inflict a performance penalty.

7 RELATED WORK

The focus of this work is the modular description of fixpoint algorithms for big-step abstract interpreters. In this section, we discuss work related to our approach presented in this paper.

Modularizing the Definition and Soundness Proofs of Big-Step Abstract Interpreters. There have been several works that modularized different parts of the definition and soundness proofs of big-step abstract interpreters. Keidel et al. [2018] describe an approach that modularizes the concrete and abstract language semantics and its soundness proof with *arrows* [Hughes 2000]. In particular, the concrete and abstract semantics is derived from the same *generic interpreter* that is composed of a number of primitive operations over values, stores, exceptions, etc. The benefit of this approach is that it guarantees that an entire analysis is sound, as long as each operation is sound. However, Keidel et al. [2018] do not show a fixpoint algorithm nor do they describe how a fixpoint algorithm should be implemented.

Bodin et al. [2019] describe a similar approach that derives both the concrete and abstract semantics from the same *skeletal semantics*. However, compared to arrows used by Keidel et al. [2018], they use a more liberal algebra called *skeletons*, which consists of hooks, filters, and branching operations. Yet, they provide similar soundness guarantees: an entire analysis derived from a skeletal semantics is sound, as long as all of its operations are sound. Bodin et al. [2019, Section 5.4] define the abstract semantics as the greatest fixpoint of the abstract collecting semantics. However, they

do not show an algorithm that computes this fixpoint, nor do they explain how such an algorithm can be described modularly.

Keidel and Erdweg [2019] describe an approach that modularizes the *effects* of the analyzed language, such as exceptions and store mutations. More specifically, the approach captures the analysis of each effect with an *analysis component* which consists of a concrete and abstract *arrow transformer*. This approach simplifies the analysis of languages with multiple effects that interact with each other. Keidel and Erdweg define a single analysis for the fixpoint algorithm. However, they do not describe the fixpoint algorithm itself, nor do they describe how it can be decomposed further. In the present work, we make use of arrows and arrow transformers to modularize the description of fixpoint algorithms by the means of sound and reusable fixpoint combinators. We use arrows to describe fixpoint combinators that are independent of the type of the fixpoint computation. This allows us to change the type of the fixpoint computation, without needing to change the definition of the fixpoint combinators.

Darais et al. [2017] describe an approach that derives several *collecting semantics* from the same generic semantics with different combinators. These combinators, for example, collect a trace of the abstract interpreter, they collect expressions that are dead code, or they compute a fixpoint. These combinators inspired the style of fixpoint combinators we present in this paper, in that our fixpoint combinators have the same type as Darais et al. combinators. However, Darais et al. do not describe a formal theory for these combinators which makes it hard to reason about their soundness. In this work, we developed a framework for modular fixpoint algorithms that is based on fixpoint combinators. This framework allows us to describe a family of fixpoint algorithms that can be configured and fine-tuned more easily, as we show in our evaluation. Furthermore, we developed a formal theory about these algorithms which allows us to prove their soundness compositionally.

Fixpoint Algorithms for Big-Step Abstract Interpreters. The space of fixpoint algorithms for big-step abstract interpreters has not been extensively studied yet. Schmidt [1995, 1998] describes one of the first fixpoint algorithms for big-step abstract interpreters that operates on the derivation tree. The fixpoint algorithm unfolds the abstract derivation tree until each branch either terminates or repeats itself. The algorithm detects recurrent calls of the abstract interpreter by memoizing parts of the abstract derivation tree. If the algorithm finds a recurrent node in a branch, it cuts off recursion to avoid non-termination which satisfies *Condition 1*. Furthermore, the fixpoint algorithm satisfies *Condition 2* by joining the environments of repeating expressions with a widening operator that ensures that infinite recursive call chains have a recurrent call. However, many details about how this algorithm actual could be implemented are missing. Specifically, Schmidt does not explain how SCGs are calculated and on which calls the algorithm iterates. Instead, the algorithm generates a number of recursive equations, which then can be solved with an arbitrary iteration order to calculate the fixpoint. We combine Schmidt's solutions to the termination conditions to implement our initial fixpoint algorithm $\texttt{fix}_{\texttt{monolithic}}$ in Section 2, which we later modularize. However, instead of generating recursive equations, our algorithm $\texttt{fix}_{\texttt{monolithic}}$ specifies an iteration order, i.e., the algorithm iterates on the innermost SCGs of the trace of the abstract interpreter [Bourdoncle 1993].

Darais et al. [2017] present another fixpoint algorithm for big-step abstract interpreters, similar to *parallel fixpoint iteration*. We implemented this algorithm in Section 4 with the combinator $\varphi_{\texttt{topmost}}$. The algorithm uses two caches to remember the analysis result of two consecutive fixpoint iterations. The algorithm then iterates over the entire program, updating the cache of the most recent iteration. If none of the caches change anymore, the algorithm has reached a fixpoint and terminates. The algorithm satisfies *Condition 1* by detecting recurrent calls if they have an existing cache entry. However, the algorithm does not satisfy the other two conditions which means that it does not terminate for infinite abstract domains.

Chaotic Fixpoint Iteration. We focus on chaotic fixpoint iteration because it is the most popular algorithm to solve a set of recursive equations in abstract interpretation [Amato et al. 2016; Bourdoncle 1993; Geser et al. 1994; Kim et al. 2020]. Chaotic iteration strategies iterate on small parts of the analyzed program and hence are typically more efficient than parallel iteration strategies [Darais et al. 2017], which iterate on the entire analyzed program. On the downside, chaotic iteration strategies are more complicated to implement, because they need to keep track of SCGs. However, we were able to encapsulate this complexity within reusable combinators, which are easier to use.

Bourdoncle [1993] presents a chaotic iteration order that is based on a weak topological ordering of the control-flow graph. The iteration order is computed before running the analysis, which requires knowing the control-flow graph ahead of time. The iteration order improves the precision as it reduces the number of widening points to the heads of SCGs. Bourdoncle [1993]'s work inspired the design of the fixpoint combinators $\varphi_{\texttt{innermost}}$ and $\varphi_{\texttt{outermost}}$ that we developed in this paper, as they use the same widening points. However, in contrast, our fixpoint combinators compute the iteration order dynamically while the analysis is running. This means our fixpoint algorithms do not need to know the control-flow graph ahead of time and can dynamically fine-tune and adapt the iteration order if needed.

Fixpoint Algorithms for Small-Step Abstract Interpreters. In contrast to big-step abstract interpreters, static analyses in small-step style have a longer history of research [Horn and Might 2010; Might and Shivers 2006a; Sergey et al. 2013; Shivers 1991]. Similar to big-step abstract interpreters, small-step abstract interpreters also seamlessly combine data-flow and control-flow information. However, they describe the abstract semantics as a small-step relation. A fixpoint algorithm for such interpreters explores the finite state space of the small-step relation. Unfortunately, it is unclear how small-step fixpoint algorithms apply to big-step abstract interpreters, because of differences in the style of semantics: While small-step abstract interpreters use continuations to explicitly model control of the interpreter as part of the state space, big-step abstract interpreters leverage the control of the meta-language (e.g., Haskell). This means that big-step abstract interpreters cannot ensure termination simply by making the state space finite, because their interpreter function may diverge nonetheless. To this end, big-step fixpoint algorithms must detect recurrent recursive calls and iterate on them which is not necessary for small-step algorithms.

8 CONCLUSION

In this paper, we studied the modular description of fixpoint algorithms for big-step abstract interpreters. We identified three conditions that guarantee the termination of big-step fixpoint algorithms. Based on these conditions, we developed a fixpoint algorithm for big-step abstract interpreters that iterates on the strongly-connected subgraphs of the graph-shaped trace. However, since the algorithm consists of a single monolithic function, it is hard to extend, configure and adapt the fixpoint algorithm. To this end, we refactored the algorithm into small reusable fixpoint combinators which allow us to change the algorithm by rearranging and adding new combinators. Furthermore, the combinators simplify the soundness proof, as each combinator can be proved sound individually once and for all. Moreover, our evaluation demonstrates that our approach describes an entire family of fixpoint algorithms for different languages and analyses that can be easily extended, adapted and configured. Lastly, the fixpoint combinators have been reimplemented in Scala and used to develop fixpoint algorithms that scale to analyze real-world WebAssembly programs.

ACKNOWLEDGEMENTS

We thank Raphaël Monat whose encouragement was decisive to resubmit this work. We thank Katharine Brandl for developing an analysis for WebAssembly in the Sturdy framework, whose fixpoint algorithm we present as a case study in this paper. Furthermore, we thank Dominik Helm for testing our artifact and Daniel Jünger, Marie Liebig, André Pacak, David Richter, Jeff Smits, Tamás Szabó for helpful discussions and feedback. Finally, we thank all anonymous reviewers for helpful discussions and feedback.

REFERENCES

Harold Abelson, R. Kent Dybvig, Christopher T. Haynes, Guillermo Juan Rozas, N. I. Adams IV, Daniel P. Friedman, Eugene E. Kohlbecker, Guy L. Steele Jr., David H. Bartley, Robert H. Halstead Jr., Don Oxley, Gerald J. Sussman, G. Brooks, Chris Hanson, Kent M. Pitman, and Mitchell Wand. 1998. Revised Report on the Algorithmic Language Scheme. *High. Order Symb. Comput.* 11, 1 (1998), 7–105.

Gianluca Amato, Francesca Scozzari, Helmut Seidl, Kalmer Apinis, and Vesal Vojdani. 2016. Efficiently intertwining widening and narrowing. *Sci. Comput. Program.* 120 (2016), 1–24. https://doi.org/10.1016/j.scico.2015.12.005

Martin Bodin, Philippa Gardner, Thomas P. Jensen, and Alan Schmitt. 2019. Skeletal semantics and their interpretations. *Proc. ACM Program. Lang.* 3, POPL (2019), 44:1–44:31. https://doi.org/10.1145/3290357

François Bourdoncle. 1993. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and their Applications*. Springer, 128–141.

Katharina Brandl, Sebastian Erdweg, Sven Keidel, and Nils Hansen. 2023. Modular Abstract Definitional Interpreters for WebAssembly. In *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 388–400.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. 238–252.

Patrick Cousot and Radhia Cousot. 1992. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In *Programming Language Implementation and Logic Programming, 4th International Symposium, PLILP'92, Leuven, Belgium, August 26-28, 1992, Proceedings*. 269–295.

David Darais, Nicholas Labich, Phuc C. Nguyen, and David Van Horn. 2017. Abstracting definitional interpreters (functional pearl). *PACMPL* 1, ICFP (2017), 12:1–12:25.

David Darais, Matthew Might, and David Van Horn. 2015. Galois transformers and modular abstract interpreters: reusable metatheory for program analysis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. 552–571.

Noah Van Es, Quentin Stiévenart, and Coen De Roover. 2019. Garbage-Free Abstract Interpretation Through Abstract Reference Counting. In *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom (LIPIcs, Vol. 134)*, Alastair F. Donaldson (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 10:1–10:33. https://doi.org/10.4230/LIPIcs.ECOOP.2019.10

Richard P. Gabriel. 1985. *Performance and Evaluation of LISP Systems*. Massachusetts Institute of Technology, USA.

Alfons Geser, Jens Knoop, Gerald Lüttgen, Bernhard Steffen, and Oliver Ruthing. 1994. Chaotic fixed point iterations.

Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 185–200. https://doi.org/10.1145/3062341.3062363

Dominik Helm, Florian Kübler, Michael Reif, Michael Eichberg, and Mira Mezini. 2020. Modular collaborative program analysis in OPAL. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 184–196. https://doi.org/10.1145/3368089.3409765

Aaron Hilbig, Daniel Lehmann, and Michael Pradel. 2021. An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases. In *WWW '21: The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021*, Jure Leskovec, Marko Grobelnik, Marc Najork, Jie Tang, and Leila Zia (Eds.). ACM / IW3C2, 2696–2708. https://doi.org/10.1145/3442381.3450138

David Van Horn and Matthew Might. 2010. Abstracting abstract machines. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*. 51–62.

John Hughes. 2000. Generalising monads to arrows. *Sci. Comput. Program.* 37, 1-3 (2000), 67–111.

Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5673)*, Jens Palsberg and Zhendong Su (Eds.). Springer, 238–255. https://doi.org/10.1007/978-3-642-03237-0_17

Sven Keidel and Sebastian Erdweg. 2019. Sound and Reusable Components for Abstract Interpretation. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 176:1–176:28. https://doi.org/10.1145/3360602

Sven Keidel and Sebastian Erdweg. 2020. A Systematic Approach to Abstract Interpretation of Program Transformations. In *Verification, Model Checking, and Abstract Interpretation - 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16-21, 2020, Proceedings*. 136–157. https://doi.org/10.1007/978-3-030-39322-9_7

Sven Keidel, Casper Bach Poulsen, and Sebastian Erdweg. 2018. Compositional soundness proofs of abstract interpreters. *Proc. ACM Program. Lang.* 2, ICFP (2018), 72:1–72:26. https://doi.org/10.1145/3236767

Sung Kook Kim, Arnaud J. Venet, and Aditya V. Thakur. 2020. Deterministic parallel fixpoint computation. *PACMPL* 4, POPL (2020), 14:1–14:33. https://doi.org/10.1145/3371082

Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, Vol. 15.

Johannes Lerch, Ben Hermann, Eric Bodden, and Mira Mezini. 2014. FlowTwist: efficient context-sensitive inside-out taint analysis for large codebases. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 98–108. https://doi.org/10.1145/2635868.2635878

Hai Liu, Eric Cheng, and Paul Hudak. 2009. Causal commutative arrows and their optimization. In *Proceedings of International Conference on Functional Programming (ICFP)*, Vol. 44. ACM, 35–46.

Laurent Mauborgne and Xavier Rival. 2005. Trace Partitioning in Abstract Interpretation Based Static Analyzers. In *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3444)*, Shmuel Sagiv (Ed.). Springer, 5–20. https://doi.org/10.1007/978-3-540-31987-0_2

Matthew Might and Olin Shivers. 2006a. Environment analysis via Delta CFA. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. 127–140.

Matthew Might and Olin Shivers. 2006b. Improving flow analyses via GammaCFA: abstract garbage collection and counting. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*. 13–25.

Ross Paterson. 2001. A New Notation for Arrows. In *Proceedings of International Conference on Functional Programming (ICFP)*. ACM, 229–240.

Gordon D. Plotkin. 1977. LCF Considered as a Programming Language. *Theor. Comput. Sci.* 5, 3 (1977), 223–255.

Tomislav Pree. 2020. *Debugging Static Analyses in Sturdy*. Master's thesis. Johannes Gutenberg University.

Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, Ron K. Cytron and Peter Lee (Eds.). ACM Press, 49–61. https://doi.org/10.1145/199448.199462

Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. 1995. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. In *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22-26, 1995, Proceedings (Lecture Notes in Computer Science, Vol. 915)*, Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach (Eds.). Springer, 651–665. https://doi.org/10.1007/3-540-59293-8_226

David A. Schmidt. 1995. Natural-Semantics-Based Abstract Interpretation (Preliminary Version). In *Static Analysis, Second International Symposium, SAS'95, Glasgow, UK, September 25-27, 1995, Proceedings (Lecture Notes in Computer Science, Vol. 983)*, Alan Mycroft (Ed.). Springer, 1–18. https://doi.org/10.1007/3-540-60360-3_28

David A. Schmidt. 1996. Abstract Interpretation of Small-Step Semantics. In *Analysis and Verification of Multiple-Agent Languages, 5th LOMAPS Workshop, Stockholm, Sweden, June 24-26, 1996, Selected Papers*. 76–99.

David A. Schmidt. 1998. Trace-Based Abstract Interpretation of Operational Semantics. *LISP Symb. Comput.* 10, 3 (1998), 237–271.

Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. 2013. Monadic Abstract Interpreters. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 12 pages.

Olin Shivers. 1991. *Control-flow analysis of higher-order languages*. Ph. D. Dissertation. Carnegie Mellon University.

Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, flow-, and field-sensitive data-flow analysis using synchronized Pushdown systems. *Proc. ACM Program. Lang.* 3, POPL (2019), 48:1–48:29. https://doi.org/10.1145/3290361

Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *30th European Conference on Object-Oriented Programming, ECOOP 2016,*

July 18-22, 2016, Rome, Italy (LIPIcs, Vol. 56), Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 22:1–22:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.22

Thomas Streicher. 2006. *Domain-theoretic foundations of functional programming*. World Scientific.

Alfred Tarski. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.* 5, 2 (1955), 285–309.

Eelco Visser, Zine-El-Abidine Benaissa, and Andrew P. Tolmach. 1998. Building Program Optimizers with Rewriting Strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*. 13–26.

Guannan Wei, Yuxuan Chen, and Tiark Rompf. 2019. Staged abstract interpreters: fast and modular whole-program analysis via meta-programming. *PACMPL* 3, OOPSLA (2019), 126:1–126:32.