Deriving with Derivatives

Optimizing Incremental Fixpoints for Higher-Order Flow Analysis

BENJAMIN QUIRING, University of Maryland, USA
DAVID VAN HORN, University of Maryland, USA

At the heart of efficient program analysis implementations are incremental solutions to fixpoint problems. These solutions can be interpreted as the derivative of the underlying analysis function. Methods that describe how to systematically derive higher-order analyses from program semantics, such as Abstracting Abstract Machines, *don't* shed light on how to efficiently *implement* those analyses. In this paper, we explore complementary techniques to optimize the derivative computation towards deriving efficient implementations. In particular, we use static specializations (by partial evaluation and rewriting) and dynamic specializations (in the form of tracking dependencies during the fixpoint), yielding efficient incremental fixpoints. We present how these optimizations apply to an example analysis of continuation-passing-style λ-calculus, and describe how they pair particularly well with tunable and optimized workset-based fixpoint methods. We demonstrate the efficacy of this approach on a flow analysis for the Standard ML language, yielding an average speed-up of 56x over an existing fixpoint method for higher-order analyses from the literature.

CCS Concepts: • **Theory of computation → Program analysis**.

Additional Key Words and Phrases: higher-order flow analysis, abstract interpretation, fixpoints, incremental computation

ACM Reference Format:
Benjamin Quiring and David Van Horn. 2024. Deriving with Derivatives: Optimizing Incremental Fixpoints for Higher-Order Flow Analysis. *Proc. ACM Program. Lang.* 8, ICFP, Article 261 (August 2024), 28 pages. https://doi.org/10.1145/3674650

1 Introduction

Underlying higher-order flow analyses is often an "analysis function" f which is *monotone*: given a larger input, its output is larger. The goal is to find a fixpoint of f which, under some finiteness conditions, always exists, and can be found by essentially iterating f on a starting point v_0,

$$v_0 \sqsubseteq \ldots \sqsubseteq v_{i+1} = v_i \sqcup f(v_i) \sqsubseteq \ldots \sqsubseteq v_n = v_{n+1},$$

where $\sqcup$ is a join operator that combines values together, like a set-union. The sequence of v_i is increasing, and the result of $f(v_i)$ is always incorporated into the next element.

For example, a standard abstract interpretation for a small-step semantics is to compute a representation of all reachable program states, represented by a set of finite abstractions of real program states. In the case of set-of-states, v_0 contains the initial program state and f is the function that, given a collection of states v, returns the successors of each state in v. The accumulated sets v_i are the explored states in a breadth-first search of the state graph, with v_i contained within v_{i+1}.

Authors' Contact Information: Benjamin Quiring, bquiring@umd.edu, Department of Computer Science, University of Maryland, College Park, MD, USA; David Van Horn, dvanhorn@cs.umd.edu, University of Maryland, College Park, MD, USA.

ACM 2475-1421/2024/8-ART261
https://doi.org/10.1145/3674650

Very often, directly computing the above increasing sequence is inefficient: f is repeatedly called on larger and larger inputs, performing similar work over and over. For example, in the set-of-states scenario, f processes the initial program state every single iteration. A more efficient approach is to only process the *newly-found* elements Δv at each iteration, which are the *changes* to the accumulated set v. When propagating changes, an *incremental* version of f is used, $\Delta f(v, \Delta v)$, which acts on both the input v and the *change* to the input Δv. The key property of the incremental version is that it is a form of derivative: it captures how f changes with respect to its input, $f(v) \sqcup \Delta f(v, \Delta v) = f(v \sqcup \Delta v)$. The incremental solution can be used to formulate the exact same increasing sequence, but written differently:

$$\begin{array}{ccccccccc} v_0 & \sqsubseteq & \ldots & \sqsubseteq & v_{i+1} = v_i \sqcup \Delta v_i & \sqsubseteq & \ldots & \sqsubseteq & v_n = v_{n+1} \\ \Delta v_0 & & \ldots & & \Delta v_{i+1} = \Delta f(v_i, \Delta v_i) & & & & \end{array}$$

These incremental solutions are behind analysis frameworks, such as the semi-naïve evaluation in Datalog [Abiteboul et al. 1995] and provide efficient implementations for fixpoints.

In the set-of-states example, the derivative only computes the state transitions from Δv, since the entirety of v was already processed. In this case, Δv_i is all states distance i from the initial state. Using the derivative, each state is processed in f at most once.

When describing program analyses for higher-order languages, it is very common to write down the semantics as a transition relation, giving rise to such a set-of-states analysis. In practice, there are too many states, and so techniques such as store-widening, which "globalizes" the memory store and share it between all states, are used to make the analysis feasible to compute (at the cost of precision) [Gilray et al. 2016; Van Horn and Might 2010]. Unfortunately, after providing the mathematical specification, discussion of how to implement such analyses in a performant manner is usually left out.

This work instead focuses on how to implement the fixpoints for higher-order analyses, seeking to bridge the gap between the mathematical specification of program analyses and the actual steps needed to implement them efficiently. Concretely, this work focuses on what we refer to as "small-step" fixpoints, where each analysis step as well as the changes being propagated are small, which contrasts with the "big-step" approach described above, in which the changes are of any size and are entirely processed each iteration. For example, the big-step approach for the set-of-states analysis yields a breadth-first search, while the small-step approach allows for traversal of the graph in any order and closely resembles a workset algorithm: at each step, a single state is taken from the workset (which is the set of changes that have not yet been processed), stepped, and any newly-found states are added into the workset.

Using smaller changes provides more freedom to the analysis implementer (whether a human or an automated tool), and provides a more tunable implementation. This work examines the impact of using small step fixpoints and changes, which we observe can be used to *specialize* the derivative on the particular kind of small change being processed, leading to the optimization of the derivative computation and therefore the optimization of the overall fixpoint. Additionally, using a workset algorithm provide inherent freedom in the *order* that elements are handled; a clever workset algorithm can eliminate work by processing certain elements before others. Using specialized derivatives opens the path to a particularly useful optimization for higher-order program analysis based around tracking *dependencies* between various computations and changes in the fixpoint.

This paper contributes

- A language $\mathcal{M}$ that describes monotone computations on lattices including sets, pairs, and maps, and an associated denotational semantics, which can be used to implement fixpoints and describe standard static analyses. (Section 2)

x			Lattice Variables
u			Non-lattice variables
d			Non-lattice computations
e	$\triangleq$	**bot** | **join** e_1 e_2	Lattice operations
	|	**var** x	Variable references
	|	**pair** e_1 e_2 | **proj**$_1$ e | **proj**$_2$ e	Product Lattices
	|	**set-singleton** $\{d\}$ | **big-join** e_2 **for** u **in** e_1	Powerset Lattices
	|	**map-singleton** $[d \mapsto e]$ | **map-lookup** $e(d)$	Map Lattices
	|	**match** d **with** $(\text{in}_i\ u_i \mapsto e_i)_{i \in \{1,2\}}$	Match

Fig. 1. The syntax of the language $\mathcal{M}$ for monotone computations.

- A formulation of the derivative for computations in $\mathcal{M}$, which act incrementally on the input and an arbitrary change to the input. Derivatives can be derived symbolically, and so are easily constructed by automated tools or performance-oriented implementers. (Section 3)
- An example abstract interpretation for a CPS-style λ-calculus, which motivates how the derivative can be specialized and optimized for small changes, and how the small-step fixpoint computation can improve performance by changing the workset order. (Section 4)
- A discussion of dynamic dependency tracking in $\mathcal{M}$ and how it can be used to implement higher-order flow analysis very efficiently. (Section 5)

We conclude with an evaluation in Section 6 obtaining an average of 56x speedup between our derivative-based implementation and standard techniques for higher-order flow analysis fixpoints, as described in Glaze et al. [2013]. We conclude with a discussion of limitations in Section 7, and related work in Section 8. Overall, this work aims to show how implementers of higher-order static analyses can more easily use incremental solutions by focusing on small changes and specializing derivatives, which, when combined with the control that a "small-step" workset-based fixpoint provides, yield highly optimized implementations.

2 Lattices and Fixpoints

The goal of the following section is to define a language $\mathcal{M}$ (for *monotone*) that describes monotone computations. We provide a type system and denotational semantics for $\mathcal{M}$; evaluation in this language yields monotone *functions* which we can take fixpoints of. This language not only allows us to describe the monotone computations but also their incremental versions (*i.e.*, their derivatives) in one language. This language is functionally a subset of Datafun [Arntzenius and Krishnaswami 2019] without λs, nested fixpoints, or discretization.

A lattice L is partially ordered set with order $\sqsubseteq$ equipped with a join, or least upper bound operator, written $\sqcup$. [1] Additionally, the lattices in this work have a bottom element, written $\bot$, that is the least element in the lattice and the identity of the join operator. From the point of view of fixpoints, the important functions on lattices are monotone, or order-preserving: if $f : L_1 \to L_2$ and $v \sqsubseteq_1 v'$, then $f(v) \sqsubseteq_2 f(v')$. Figure 1 describes the syntax for the language $\mathcal{M}$.

As a first note, this language partitions computations (and variables) into *lattice* computations e (and variables x) which must be monotone, and *non-lattice* computations d (and variables u). The purpose of partitioning computations into lattice and non-lattice is to enforce monotonicity,

[1] As well as a meet (*syn.*greatest lower bound) operator. For this work, we limit discussion to join-semilattices. There is no theoretical problem with meets, they just add one more bit of unnecessary complexity.

and the language $\mathcal{M}$ restricts how these two can interact with each other. In particular, the key property is that non-lattice computations d are disallowed from referring to lattice variables x.

To get a feel for the distinction between lattice and non-lattice computations, consider the set-of-states example from the introduction, where the goal is to determine the reachable program states of a small-step semantics, which is determined via a graph traversal: all set operations (*e.g.*constructing singleton sets, unions) are monotone lattice computations, whereas the operations used to actually step individual states (investigating what instruction the state performs next, updating state-internal environments or memory, etc.), are non-lattice computations.

There are five categories of lattice computations in $\mathcal{M}$. The first category is available for all lattices: every lattice has a bottom element and a join operator, which are represented by the expressions **bot** and **join** e_1 e_2, respectively. The second category is variable reference **var** x: lattice computations have lattice-valued inputs which are named by lattice variables x, and need to reference these inputs. The remaining three categories each deal with a particular type of lattice former: product lattices, powerset lattices, and map lattices. This work focuses on these three lattice "building blocks" because they are commonly found in static analysis and are required to describe the motivating examples for our optimizations later in the paper.

Product lattices **Pair** L_1 L_2 are formed by taking pairs from two lattices L_1 and L_2. The bottom element is the pair of the respective bottoms in both components, and the join operation simply distributes into each component. The $\mathcal{M}$ language has an introduction form for pairs **pair** e_1 e_2, and two elimination forms: $\mathbf{proj}_1$ e for the first projection and $\mathbf{proj}_2$ e for the second projection.

Powerset lattices **Set** U consist of the subsets of a set U. The bottom element is the empty set and join is union. The lattice language has an introduction form for singleton sets, **set-singleton** $\{d\}$, which constructs a singleton containing the result of the non-lattice computation d. There is also an elimination form **big-join** e_2 **for** u **in** e_1, which evaluates e_2 with a different binding of u for each element in the result of e_1, and then joins all results together.

Importantly, the variable u is a non-lattice variable. Non-lattice variables can only refer to non-lattice values, and can only be referenced in non-lattice computations. Symmetrically, lattice variables x can only refer to lattice values, and cannot be referenced in non-lattice computations.

Map lattices **Map** U L consist of functions, often represented by (finite) maps, from any (finite) set U to any lattice L. The bottom element is the function which sends every input to L's bottom, and the join operator joins the results at each input: $(m \sqcup m')(x) = m(x) \sqcup m'(x)$. The lattice language has an introduction form for "singleton maps," **map-singleton** $[d \mapsto e]$, which are bottom-valued for all but a pointed input (the result of the computation d) which maps to the result of e. The lattice language also has an elimination form for maps **map-lookup** $e(d)$, which evaluates the resulting map of e at the result of the non-lattice computation d.

Finally, **match** d **with** $(\mathrm{in}_i\ u_i \mapsto e_i)_{i \in \{1,2\}}$ describes pattern matching on sums in the non-lattice fragment, where each branch is still a *lattice* computation. This form is monotone because the discriminant d cannot depend on lattice inputs, and each branch e_i must be monotone. When writing program analyses, these kind of sums are necessary to, for example, provide different cases for different expression variants.

2.1 Well-formed Computations

Before describing a denotational semantics for the monotonic language we describe a type system for it to ensure computations are well-defined. After all, it does not make sense to join two values together if they are not in the same lattice, or take the first projection of a set. The type system is in Figure 2. This type system has the property that well-typed programs yield well-defined evaluations in the denotational semantics. Just as lattice and non-lattice computations and variables

$$
\begin{array}{lcll}
U & & & \text{Non-lattice types} \\
L & \triangleq & \mathbf{Pair}\ L_1\ L_2 & \text{Product Lattices} \\
 & | & \mathbf{Set}\ U & \text{Powerset Lattices} \\
 & | & \mathbf{Map}\ U\ L & \text{Map Lattices} \\
\Gamma & \triangleq & \cdot \mid \Gamma, x : L & \text{Lattice type environments} \\
\Sigma & \triangleq & \cdot \mid \Sigma, u : U & \text{Non-lattice type environments} \\
\Gamma; \Sigma \vdash e : L & & & \text{Lattice typing judgements} \\
\Sigma \vdash d : U & & & \text{Non-lattice typing judgements}
\end{array}
$$

All Lattices

$$\frac{\Gamma(x) = L}{\Gamma; \Sigma \vdash \mathbf{var}\ x : L} \qquad \frac{}{\Gamma; \Sigma \vdash \mathbf{bot} : L} \qquad \frac{\Gamma; \Sigma \vdash e_1 : L \quad \Gamma; \Sigma \vdash e_2 : L}{\Gamma; \Sigma \vdash \mathbf{join}\ e_1\ e_2 : L}$$

Product Lattices

$$\frac{\Gamma; \Sigma \vdash e_1 : L_1 \quad \Gamma; \Sigma \vdash e_2 : L_2}{\Gamma; \Sigma \vdash \mathbf{pair}\ e_1\ e_2 : \mathbf{Pair}\ L_1\ L_2} \qquad \frac{\Gamma; \Sigma \vdash e : \mathbf{Pair}\ L_1\ L_2}{\Gamma; \Sigma \vdash \mathbf{proj}_i\ e : L_i}$$

Powerset Lattices

$$\frac{\Sigma \vdash d : U}{\Gamma; \Sigma \vdash \mathbf{set\text{-}singleton}\ \{d\} : \mathbf{Set}\ U} \qquad \frac{\Gamma; \Sigma \vdash e_1 : \mathbf{Set}\ U \quad \Gamma; \Sigma, u : U \vdash e_2 : L}{\Gamma; \Sigma \vdash \mathbf{big\text{-}join}\ e_2\ \mathbf{for}\ u\ \mathbf{in}\ e_1 : L}$$

Map Lattices

$$\frac{\Sigma \vdash d : U \quad \Gamma; \Sigma \vdash e : L}{\Gamma; \Sigma \vdash \mathbf{map\text{-}singleton}\ [d \mapsto e] : \mathbf{Map}\ U\ L} \qquad \frac{\Gamma; \Sigma \vdash e : \mathbf{Map}\ U\ L \quad \Sigma \vdash d : U}{\Gamma; \Sigma \vdash \mathbf{map\text{-}lookup}\ e(d) : L}$$

Match

$$\frac{\Sigma \vdash d : U_1 + U_2 \quad \Gamma; \Sigma, u_1 : U_1 \vdash e_1 : L \quad \Gamma; \Sigma, u_2 : U_2 \vdash e_2 : L}{\Gamma; \Sigma \vdash \mathbf{match}\ d\ \mathbf{with}\ (\mathrm{in}_i\ u_i \mapsto e_i)_{i \in \{1,2\}} : L}$$

Fig. 2. Typing judgements for the $\mathcal{M}$ language.

are partitioned in $\mathcal{M}$, the typing environments in the type system are partitioned: one for the lattice domains Γ and one for the non-lattice domains Σ.

Judgements for lattice computations take the form $\Gamma; \Sigma \vdash e : L$, which means that e has the type L under the provided lattice typing environment Γ and non-lattice typing environment Σ. Judgements for non-lattice computations take the form $\Sigma \vdash d : U$; because non-lattice computations cannot refer to lattice variables, they do not depend on the lattice typing environment Γ. Because we do not place constraints on what non-lattice computations might be, we must assume an evaluation function for non-lattice computations as well as the notion of typing judgements on these terms, with the property that well-typed non-lattice computations yield well-defined evaluations.

The typing rules are entirely what one expects from such a type system: bottom can be given any lattice type, two computations must belong to the same lattice to be joined, pairs form a pair and projections are actually projections, set-singletons provide a set, big-joins occur over sets, map-singletons produce a map, map-lookups use a map, and matches must match on a sum (in the non-lattice domain) and produce the same type in both branches.

All Lattices

$$[\![\textbf{bot}]\!]_{\gamma,\sigma} = \bot \qquad [\![\textbf{join } e_1\ e_2]\!]_{\gamma,\sigma} = [\![e_1]\!]_{\gamma,\sigma} \sqcup [\![e_2]\!]_{\gamma,\sigma} \qquad [\![\textbf{var } x]\!]_{\gamma,\sigma} = \gamma(x)$$

Product Lattices

$$[\![\textbf{pair } e_1\ e_2]\!]_{\gamma,\sigma} = \langle [\![e_1]\!]_{\gamma,\sigma}, [\![e_2]\!]_{\gamma,\sigma} \rangle \quad [\![\textbf{proj}_i\ e]\!]_{\gamma,\sigma} = v_i \text{ where } [\![e]\!]_{\gamma,\sigma} = \langle v_1, v_2 \rangle$$

Powerset Lattices

$$[\![\textbf{set-singleton } \{d\}]\!]_{\gamma,\sigma} = \{[\![d]\!]_\sigma\} \qquad [\![\textbf{big-join } e_2 \textbf{ for } u \textbf{ in } e_1]\!]_{\gamma,\sigma} = \bigsqcup_{elm \in [\![e_1]\!]_{\gamma,\sigma}} [\![e_2]\!]_{\gamma,\sigma[u \mapsto elm]}$$

Map Lattices

$$[\![\textbf{map-singleton } [d \mapsto e]]\!]_{\gamma,\sigma} = [[\![d]\!]_\sigma \mapsto [\![e]\!]_{\gamma,\sigma}] \qquad [\![\textbf{map-lookup } e(d)]\!]_{\gamma,\sigma} = [\![e]\!]_{\gamma,\sigma}([\![d]\!]_\sigma)$$

Match

$$[\![\textbf{match } d \textbf{ with } (\text{in}_i\ u_i \mapsto e_i)_{i \in \{1,2\}}]\!]_{\gamma,\sigma} = [\![e_i]\!]_{\gamma,\sigma[u_i \mapsto elm_i]} \text{ where } [\![d]\!]_\sigma = \text{in}_i\ elm_i$$

Fig. 3. Denotational semantics for the $\mathcal{M}$ language.

2.2 Denotational Semantics

We provide a denotational semantics for the $\mathcal{M}$ language in Figure 3. Evaluation is performed with the $[\![e]\!]_{\gamma,\sigma}$ function, where γ is a lattice value environment mapping lattice variables to lattice values, and σ is a non-lattice value environment mapping non-lattice variables to non-lattice values. As mentioned previously, we assume that for non-lattice computations d there is an evaluation function $[\![d]\!]_\sigma$ which only relies on the non-lattice environment σ, along with the assumed property that well-typed non-lattice computations evaluate:

If $\Sigma \vdash d : U$ and for all u, $\sigma(u)$ is a value of type $\Sigma(u)$, then $[\![d]\!]_\sigma$ returns a value of type U.

The key property for the type system and the semantics is that well typed terms evaluate:

Theorem 2.1 (Well typed-terms evaluate).

If $\Gamma; \Sigma \vdash e : L$, *for all* x, $\gamma(x)$ *is of type* $\Gamma(x)$, *and for all* u, $\sigma(u)$ *is of type* $\Sigma(u)$, *then* $[\![e]\!]_{\gamma,\sigma}$ *produces an element of* L.

Proof. By structural induction over terms e and the assumptions on non-lattice computations. □

The denotational semantics are structurally recursive, straightforward denotational interpretations of the expression forms: **bot** evaluates to the bottom element (whose lattice can be inferred from the typing derivation), **join** $e1$ $e2$ evaluates the two results and joins them, and **var** x looks up the value of x in the environment. Pairs evaluate to a pair, and projection occurs naturally.

The singleton set expression **set-singleton** $\{d\}$ evaluates the non-lattice computation d under the non-lattice environment and packages it into a singleton set. For big-joins **big-join** e_2 **for** u **in** e_1, the set e_1 is evaluated, and for every element elm in it, e_2 is evaluated under an extended environment mapping u to elm. Map singletons use shorthand for defining maps: $[a \mapsto b]$ means the map that takes every element except a to bottom, and takes a to b. Map lookup evaluates the map, then evaluates the key, then applies the map to the key. Finally, matches evaluate the discriminant, and recursively evaluate one of the body terms in an updated environment.

Now that the semantics of the language $\mathcal{M}$ is defined, we can prove that the functions it describes are actually monotone.

Theorem 2.2 ($\mathcal{M}$ produces monotone functions).

If $\Gamma, x : L; \Sigma \vdash e : L'$, *for all* x, $\gamma(x)$ *is of type* $\Gamma(x)$, *and for all* u, $\sigma(u)$ *is of type* $\Sigma(u)$, *and* $v \sqsubseteq v'$ *are elements of* L, *then* $[\![e]\!]_{\gamma[x \mapsto v],\sigma} \sqsubseteq [\![e]\!]_{\gamma[x \mapsto v'],\sigma}$.

Proof. By structural induction over terms e, and the following properties of big-joins:

$$\bigsqcup_{u \in X \cup Y} f(u) = \left(\bigsqcup_{u \in X} f(u)\right) \sqcup \left(\bigsqcup_{u \in Y} f(u)\right) \quad \text{and} \quad \bigsqcup_{u \in X} f(u) \sqcup f'(u) = \left(\bigsqcup_{u \in X} f(u)\right) \sqcup \left(\bigsqcup_{u \in X} f'(u)\right),$$

which hold due to the finiteness of X and Y and the commutativity of $\sqcup$. □

2.3 Fixpoints and Termination

Given a monotone function $f : L \to L$, one can define the *increasing* (also known as an *extensive*) function

$$g(v) = v \sqcup f(v)$$

which can be iterated, providing an increasing sequence (where $g^{i+1}(v) = g(g^i(v))$)

$$v_0 \sqsubseteq g(v_0) \sqsubseteq g^2(v_0) \sqsubseteq \ldots \sqsubseteq g^i(v_0) \sqsubseteq \ldots$$

which will reach a fixpoint $g^n(v_0) = g^{n+1}(v_0)$, given some well-studied finiteness conditions [Møller and Schwartzbach 2023], in particular when the lattices have finite height.

For the $\mathcal{M}$ language, the analysis function f evaluates a well-typed expression e_0

$$x : L; \cdot \vdash e_0 : L$$

where e_0 has a single input (free variable) x of type L. The evaluation occurs under a lattice value environment taking x to its value, $[x \mapsto v]$, and an empty non-lattice value environment $[]$:

$$f(v) = [\![e_0]\!]_{[x \mapsto v],[]}$$

The fixpoint can be calculated with a single outer loop (written in pseudocode), given an initial value v_0:

```
letrec fixpoint(v) =
   let v' = v ⊔ [[e0]]_[x↦v],[] in
   if v = v' then v
   else fixpoint(v') in
fixpoint(v0)
```
(2.3.1)

The i^{th} iteration of the fixpoint loop will be called with the value $g^i(v_0)$ from the increasing sequence above.

3 Efficient Fixpoints and Derivatives

As discussed in the introduction, the expression e is evaluated on increasingly large values for its input x, which may redo significant amounts of computation. Using the derivative of the monotone analysis function, which can be formulated as the evaluation of a new term, avoids excess recomputation. In the following, we describe how to derive incremental version of monotone computations written in $\mathcal{M}$ and use those to construct more efficient fixpoints.

Definition 1 (Derivative constraint). *The derivative of a monotonic function* $f : L \to L$ *is a function* $\Delta f : L \times L \to L$ *that captures how* f *changes when its input changes:*

$$f(v) \sqcup \Delta f(v, \Delta v) = f(v \sqcup \Delta v).$$

At a high level, the derivative captures how the result of f changes between an input x and a larger input $v \sqcup \Delta v$. We simply require that the result of the derivative combined via join with the result of $f(v)$ is the result of $f(v \sqcup \Delta v)$. In this sense, we have a *constraint* rather than a *definition*, which means there are multiple possible derivatives. Writing the constraint like this ensures that the derivative always exists (taking $\Delta f(v, \Delta v) = f(v \sqcup \Delta v)$). Occasionally it is the case that implementations desire that the derivative, or components of it, are disjoint with the original $f(v)$, which can be done with a "difference" operator [Eo and Yi 2002]. Disjointness means that joins can be more efficient, since elements are unique, and for workset algorithms, disjointness is required to avoid processing elements multiple times. In this work, we do not describe difference operators further.

In terms of the lattice language, given the expression e_0 that computes f we'll see in the following Section 3.1 that we can symbolically derive an expression Δe_0 from e_0 that computes Δf. The expression Δe_0 is well-typed with two free variables, x and Δx, both of type L,

$$x : L, \Delta x : L; \cdot \vdash \Delta e_0 : L$$

such that

$$[\![e_0]\!]_{[x \mapsto v],[\,]} \sqcup [\![\Delta e_0]\!]_{[x \mapsto v,\, \Delta x \mapsto \Delta v],[\,]} = [\![e_0]\!]_{[x \mapsto v \sqcup \Delta v],[\,]}.$$

With the derivative expression Δe_0, the earlier fixpoint loop (2.3.1) can be reformulated as

$$\begin{array}{l} \texttt{letrec } \mathit{fixpoint}(v, \Delta v) = \\ \quad \texttt{let } v' = v \sqcup \Delta v \\ \quad \texttt{let } \Delta v' = [\![\Delta e_0]\!]_{[x \mapsto v,\, \Delta x \mapsto \Delta v],[\,]} \texttt{ in} \\ \quad \texttt{if } v = v' \texttt{ then } v \\ \quad \texttt{else } \mathit{fixpoint}(v', \Delta v') \texttt{ in} \\ \mathit{fixpoint}(\bot, [\![e_0]\!]_{[x \mapsto \bot],[\,]}) \end{array} \tag{3.0.1}$$

This fixpoint provides the *same* increasing sequence for v as the original fixpoint method did, which is proven through the following two invariants, where v_i is the sequence for (2.3.1) and $v'_i, \Delta v_i$ is the sequence for (3.0.1).

Theorem 3.1 (Derivative yields same fixpoint).

$$\textit{for all } i, \quad \begin{array}{ll} (1) & v_{i+1} = v'_i \sqcup \Delta v_i \\ (2) & v'_i \sqcup \Delta v_i = v'_i \sqcup f(v'_i) \end{array}$$

Proof. By induction on i. Invariant (2) can be proven in isolation by unfolding the definitions and applying the derivative constraint. Invariant (1) is proven by unfolding the definitions and applying invariant (2) and the derivative constraint. □

Despite producing the same sequence, this fixpoint will be more efficient, assuming that the derivative avoids redoing computation.

One might expect some kind of "disjointness" from the derivative constraint (which is what a "difference" would provide), but in general that is hard to guarantee. For example, in the set-of-states scenario the result of Δf is a set of states one step away from the change Δv. Both the sets v and Δv may contain two distinct states that step to the same state; in this case $f(v)$ and $\Delta f(v, \Delta v)$ would have overlap. It *is* desired to ensure disjointness of v and Δv in the fixpoint (and potentially in other computations inside the monotone function), obtained via difference operators, but we omit this detail. Given a derivative function, it is always possible to compose a difference operator with it, as long as monotonicity is preserved.

In the following, we describe how to derive the derivative term Δe_0 from the source term e_0. Afterwards, we explore an abstract interpretation example to motivate the discussion of specialized derivatives and further optimizations.

All Lattices

$$\Delta_{x,\Delta x}\ \mathbf{bot} = \mathbf{bot}$$
$$\Delta_{x,\Delta x}\ (\mathbf{join}\ e_1\ e_2) = \mathbf{join}\ (\Delta_{x,\Delta x}\ e_1)\ (\Delta_{x,\Delta x}\ e_2)$$
$$\Delta_{x,\Delta x}\ \mathbf{var}\ x = \mathbf{var}\ \Delta x$$
$$\Delta_{x,\Delta x}\ \mathbf{var}\ y = \mathbf{bot}$$

Product Lattices

$$\Delta_{x,\Delta x}\ (\mathbf{pair}\ e_1\ e_2) = \mathbf{pair}\ (\Delta_{x,\Delta x}\ e_1)\ (\Delta_{x,\Delta x}\ e_2)$$
$$\Delta_{x,\Delta x}\ (\mathbf{proj}_i\ e) = \mathbf{proj}_i\ (\Delta_{x,\Delta x}\ e)$$

Powerset Lattices

$$\Delta_{x,\Delta x}\ (\mathbf{set\text{-}singleton}\ \{d\}) = \mathbf{bot}$$

$$\Delta_{x,\Delta x}\ (\mathbf{big\text{-}join}\ e_2\ \mathbf{for}\ u\ \mathbf{in}\ e_1) = \begin{array}{l}\mathbf{join}\ (\mathbf{big\text{-}join}\ e_2\ \mathbf{for}\ u\ \mathbf{in}\ \Delta_{x,\Delta x}\ e_1) \\ \mathbf{join}\ (\mathbf{big\text{-}join}\ \Delta_{x,\Delta x}\ e_2\ \mathbf{for}\ u\ \mathbf{in}\ e_1) \\ \quad (\mathbf{big\text{-}join}\ \Delta_{x,\Delta x}\ e_2\ \mathbf{for}\ u\ \mathbf{in}\ \Delta_{x,\Delta x}\ e_1)\end{array}$$

Map Lattices

$$\Delta_{x,\Delta x}\ (\mathbf{map\text{-}singleton}\ [d \mapsto e]) = \mathbf{map\text{-}singleton}\ [d \mapsto \Delta_{x,\Delta x}\ e]$$
$$\Delta_{x,\Delta x}\ (\mathbf{map\text{-}lookup}\ e(d)) = \mathbf{map\text{-}lookup}\ (\Delta_{x,\Delta x}\ e)(d)$$

Match

$$\Delta_{x,\Delta x}\ (\mathbf{match}\ d\ \mathbf{with}\ (\mathrm{in}_i\ u_i \mapsto e_i)_{i\in\{1,2\}}) = \mathbf{match}\ d\ \mathbf{with}\ (\mathrm{in}_i\ u_i \mapsto \Delta_{x,\Delta x}\ e_i)_{i\in\{1,2\}}$$

Fig. 4. Derivatives for the $\mathcal{M}$ language.

3.1 Deriving Derivatives

Figure 4 defines a metafunction $\Delta_{x,\Delta x}\ e$ that derives the derivative term from a given expression e, with respect to the variable x, which produces a term with a new free variable for the change, Δx. This metafunction is structurally recursive and, for the most part, quite simple. These derivatives satisfy the derivative constraint,

$$[\![e]\!]_{\gamma[x\mapsto v],\sigma} \sqcup [\![\Delta_{x,\Delta x}\ e]\!]_{\gamma[x\mapsto v,\Delta x\mapsto \Delta v],\sigma} = [\![e]\!]_{\gamma[x\mapsto v\sqcup\Delta v],\sigma}, \tag{3.1.1}$$

and are analogous to semi-naïve evaluation in Datalog [Abiteboul et al. 1995] or Datafun [Arntzenius and Krishnaswami 2019].

The derivative of bottom is bottom, and the derivative of a join is simply the join of its derivatives. The derivative of a variable **var** x is simply the variable for the change to the input, **var** Δx, and for any other variable **var** y the derivative is bottom (since **var** y does not change with respect to x).

Just like for joins, the derivative distributes into both pairs and projections. A set singleton **set-singleton** $\{d\}$ is a *constant* with respect to the lattice input, because the non-lattice computation d *cannot reference* the lattice variable. Because of this, the derivative of a set singleton is always bottom.

Set big-joins are the most complicated case of derivative; considering a big-join

$$\mathbf{big\text{-}join}\ e_2\ \mathbf{for}\ u\ \mathbf{in}\ e_1,$$

the result of e_1 may change with respect to x, meaning that e_2 is evaluated more times, but e_2 may also change with respect to x. The derivative consists of three big-joins (each then joined together,

not parenthesized as join is associative): the first exploring the original computation e_2 under the new elements $\Delta_{x,\Delta x}\ e_1$, the second exploring the change $\Delta_{x,\Delta x}\ e_2$ under the original elements e_1, and the third exploring the change $\Delta_{x,\Delta x}\ e_2$ under the new elements $\Delta_{x,\Delta x}\ e_1$. This is proven to be a derivative by inspecting the evaluations

$$\begin{array}{ll} & [\![\textbf{big-join}\ e_2\ \textbf{for}\ u\ \textbf{in}\ e_1]\!]_{\gamma[x\mapsto v],\sigma} \\ \sqcup & [\![\textbf{big-join}\ \Delta_{x,\Delta x}\ e_2\ \textbf{for}\ u\ \textbf{in}\ e_1]\!]_{\gamma[x\mapsto v,\Delta x\mapsto \Delta v],\sigma} \\ \sqcup & [\![\textbf{big-join}\ e_2\ \textbf{for}\ u\ \textbf{in}\ \Delta_{x,\Delta x}\ e_1]\!]_{\gamma[x\mapsto v,\Delta x\mapsto \Delta v],\sigma} \\ \sqcup & [\![\textbf{big-join}\ \Delta_{x,\Delta x}\ e_2\ \textbf{for}\ u\ \textbf{in}\ \Delta_{x,\Delta x}\ e_1]\!]_{\gamma[x\mapsto v,\Delta x\mapsto \Delta v],\sigma} \\ = & [\![\textbf{big-join}\ e_2\ \textbf{for}\ u\ \textbf{in}\ e_1]\!]_{\gamma[x\mapsto v\sqcup\Delta v],\sigma} \end{array}$$

and applying the same two properties of big-joins from the proof of Theorem 2.2 to reorder joins, along with the fact that $[\![e]\!]_{\gamma[x\mapsto v],\sigma} = [\![e]\!]_{\gamma,\sigma}$ if e does not refer to x (for all e, x, γ, and σ).

The final derivatives are for map lattices: the derivative of a map singleton is simply a map singleton mapping to the change, and the derivative of a map lookup is a map lookup on the change of the map.

The correctness theorem for the derivatives is that they satisfy the derivative constraint.

Theorem 3.2 (Derivatives are derivatives).

If $\Gamma[x : L]; \Sigma \vdash e : L'$, v *and* Δv *are elements of* L, *and* Δx *is not free in* e, *then*

$$[\![e]\!]_{\gamma[x\mapsto v],\sigma} \sqcup [\![\Delta_{x,\Delta x}\ e]\!]_{\gamma[x\mapsto v,\Delta x\mapsto \Delta v],\sigma} = [\![e]\!]_{\gamma[x\mapsto v\sqcup\Delta v],\sigma}$$

Proof. By structural induction over terms e. □

In addition to satisfying the derivative constraint, the derivatives described above produce well-typed terms: derivatives themselves are monotone.

Theorem 3.3 (Derivatives are well-typed).

If $\Gamma[x : L]; \Sigma \vdash e : L'$, *then* $\Gamma[x : L, \Delta x : L]; \Sigma \vdash \Delta_{x,\Delta x}\ e : L'$.

Proof. By structural induction over terms e. □

4 Abstract Interpretation Example

Now that we have described how to write monotone functions and their derivatives in the language $\mathcal{M}$, we are now ready to motivate specialized derivatives and small-step fixpoints with a formal example. In the following section, we present an abstract interpretation of a CPS-style λ-calculus. Following, we discuss various inefficiencies and associated optimizations for this example, and describe the small-step workset-like fixpoint method. Throughout this example, we use red to indicate non-lattice domains, visually segregating them from their lattice domain counterparts in black.

The CPS-style language has exactly one type of expression: function call $\underline{f\ arg}$. We use underlines to denote that an element is a piece of syntax in this language. Both $\underline{f}$ and $\underline{arg}$ can be either variables $\underline{x}$ or λs, $\underline{\lambda x.e}$.

Figure 5 contains an example program that we will use to describe how the CPS analysis works, how various fixpoints can be inefficient, and how derivatives and our additional optimizations can improve them. For the sake of readability, we provide both a human-parsable version (on the left) and the corresponding CPS form of this program (on the right) where the `lets` have been desugared into applications and the continuation parameters have been added (`k0` being the top-level continuation). The CPS version is formatted so that whenever an application would take multiple lines, the function and the arguments appear at the same indentation level. There is no theoretical reason that analyses on ANF-style languages will not work (our evaluation uses a

```
let f = λ x y .
   let x' = (λ a . a) x in
   let y' = (λ b . b) y in
   y' x' in
let y0' = λ y0 . y0
let x1' = f (λ x1 . x1) y0' in
let x2' = f (λ x2 . x2) y0' in
x2'
```

```
(λ f .
  (λ y0' .
    f
    (λ x1 kx1 . kx1 x1)
    y0'
    (λ x1' .
      f
      (λ x2 kx2 . kx2 x2)
      y0'
      (λ x2' . k0 x2')))
  (λ y0 ky0 . ky0 y0))
```

```
(λ x y k .
  (λ a ka . ka a)
  x
  (λ x' .
    (λ b kb . kb b)
    y
    (λ y' .
      y' x' k)))
```

Fig. 5. Example for the abstract interpretation. The left is ANF-style, and the right is the equivalent CPS-style.

hybrid ANF-CPS language), but adding extra expression forms to the analyzed language would simply introduce extra expressions in the monotone computations, without providing much insight: the CPS language contains all the necessary core ideas. Additionally, the example will use multi-argument λs whereas the formalism will only use single-argument; the use of single-argument functions is to avoid extra expressions that don't contribute core ideas.

First, we will explain what and how this example program computes, then describe the CPS-style analysis and its monotone function, and then describe how the fixpoints operate for this example program. To begin, this program binds to `f` a function of two parameters that it will later call twice, with (syntactically) distinct first arguments (`λ x1 . x1`) and (`λ x2 . x2`), but the same second argument `y0'`, all of which are semantically equivalent to the identity function.

The internals of `f` calls an identity function on its first parameter `x` obtaining `x'`, and applies a distinct identity function to its second `y` obtaining `y'`, and finishes by returning the result of `y'` applied to `x'`. Because `x` and `y` are always semantically the identity function, this example doesn't *compute* anything particularly productive, but the *way* that it does this is yields insights into the various fixpoint methods.

The goal of the program analysis is to determine the final result of the program, which is the result of the first call to the function `f`. We describe the abstract semantics of the CPS language in the style of Abstracting Abstract Machines [Van Horn and Might 2010]. Program states for this language consist of (1) a program location e, (2) an "environment" env mapping variables in scope to "addresses" a (memory locations), and (3) a "store" $store$ mapping addresses to closures (a λ paired with an environment). This definition can serve for both the incomputable "concrete" semantics by choosing an infinite address set as well as the computable abstract interpretation by choosing a finite set. When finitized, the addresses must be reused during the analysis (which finitely bounds the number of possible states). In the standard 0CFA [Shivers 1991], which we will use for the example program, there is one address per variable. Due to this reuse, stores map to *sets* of closures: when reusing an address, the old closure values must be retained for soundness.

With these ideas, one can formulate a set-of-states style semantics to find the reachable program states and the eventual final result of the program. For some state $\langle f\ arg, env, store \rangle$, the successors states are found by evaluating the function f and argument arg and then branching into the λs that f may refer to (this process will be explained in-depth later).

The set-of-states abstract interpretation is often too expensive due to stores being large and hard to compare (making it expensive to test if a state has been visited), and so often the technique of

$$
\begin{array}{rcll}
\underline{e}, \underline{f\ arg} & \in & \texttt{Expr} & \\
\underline{x}, \underline{y} & \in & \texttt{Var} & \\
\underline{f}, \underline{arg}, \underline{\lambda x.e} & \in & \texttt{Var} + (\texttt{Var} \times \texttt{Expr}) & \\
state & \in & \texttt{State} & ::= \textbf{Pair}\ (\textbf{Set}\ \texttt{Config})\ \texttt{Store} \\
configs & \in & \textbf{Set}\ \texttt{Config} & \\
config & \in & \texttt{Config} & ::= \texttt{Expr} \times \texttt{Env} \\
env & \in & \texttt{Env} & ::= \texttt{Var} \rightarrow \texttt{Addr} \\
cl & \in & \texttt{Closure} & ::= \texttt{Var} \times \texttt{Expr} \times \texttt{Env} \\
a & \in & \texttt{Addr} & ::= \text{a finite set} \\
store & \in & \texttt{Store} & ::= \textbf{Map}\ \texttt{Addr}\ (\textbf{Set}\ \texttt{Closure})
\end{array}
$$

Fig. 6. Semantic domains of the abstract interpretation example.

$$
\begin{array}{l}
\mathcal{E}(\underline{arg}, env, store) = \ \textbf{match}\ \underline{arg}\ \textbf{with} \\
\qquad\qquad\qquad\qquad \text{in}_1\ \underline{x} \mapsto \textbf{map-lookup}\ store(env(\underline{x})) \\
\qquad\qquad\qquad\qquad \text{in}_2\ \underline{\lambda x.e} \mapsto \textbf{set-singleton}\ \{\langle \underline{\lambda x.e}, env\rangle\}
\end{array}
$$

$$
\begin{array}{rl}
\mathcal{S}(state) & = \textbf{big-join}\ inner\ \textbf{for}\ config\ \textbf{in}\ configs \\
\text{where}\ configs, store & = \textbf{proj}_1\ \textbf{var}\ state, \textbf{proj}_2\ \textbf{var}\ state \\
\langle \underline{f\ arg}, env\rangle & = config \\
val_f, val_{arg} & = \mathcal{E}(\underline{f}, env, store), \mathcal{E}(\underline{arg}, env, store) \\
inner & = \textbf{big-join}\ (\textbf{pair}\ (\textbf{set-singleton}\ \{config'\})\ store')\ \textbf{for}\ cl\ \textbf{in}\ val_f \\
\langle \underline{\lambda x.e}, env_{cl}\rangle & = cl \\
a & = \ldots \text{abstract address for } \underline{x} \ldots \\
store' & = \textbf{map-singleton}\ [a \mapsto val_{arg}] \\
config' & = \langle \underline{e}, env_{cl}[\underline{x} \mapsto a]\rangle
\end{array}
$$

Fig. 7. Abstract interpretation monotone analysis function example.

"store-widening" is used for the computed analysis, which globalizes the store, making it shared among all *configurations* (states without stores). Performing this change changes the type of the *lattice* of the analysis: as the result is now a pair (of a set of configurations and a store), the store changes from a non-lattice value, as it was state-internal, to a lattice value.

Figure 6 contains the semantic domains of the store-widened CPS abstract interpretation. The lattice of the analysis is `State`, which again consists of a pair of a set of configurations $configs \in$ **Set** `Config` and a store $store \in$ `Store`. Each configuration $config = \langle \underline{f\ arg}, env\rangle$ contains an expression $\underline{e}$ and an environment env, and each closure contains a λ (written as $\underline{\lambda x.e} \in \texttt{Var} \times \texttt{Expr}$) and an environment. The monotone "successor" computation $\mathcal{S}$ takes a `State` and returns a `State`, and can be found in Figure 7. We use "where" notation to indicate substitution, as the lattice language is very verbose.

The successor $\mathcal{S}$ takes the pair of the set of configurations *configs* and the store *store* and transitions each of the configurations under the store. Each configuration is handled with the store via the *inner* term. A configuration *config* transitions by first evaluating the function $\underline{f}$ and argument $\underline{arg}$ with the evaluation function $\mathcal{E}$ (defined separately for conciseness and could be inlined into $\mathcal{S}$), which either looks up a variable in the environment and store, or evaluates a λ

to a closure. Then, as there may be multiple closures for $\underline{f}$, *inner* consists of another big-join that explores each possible function call.

For each possible closure *cl*, the closure is destructed into the λ and environment, and the parameter $\underline{x}$ is bound to the argument value by updating the closure's environment $env_{cl}[\underline{x} \mapsto a]$ and producing the update to the store *store*′. Because the overall lattice is a pair, the result of each transition is also a pair consisting of "newly-found" configurations **set-singleton** {*config*′} and updates to the store *store*′. The two nested big-joins ensure that every possible configuration, with every possible function, is explored.

Store-widened abstract-interpretations are not typically written in this fashion, instead often applying a widening operator to the set-of-states approach as in Gilray et al. [2016], which does not yield particularly efficient analyses. Glaze et al. [2013] is an exception, which also introduces the idea of "store-deltas": the successor only needs to produce the *updates* to the store, written as *store*′ in the figure. Store updates from the analysis are all joined together through the big-joins, and are eventually joined with the global store in the fixpoint loop.

For the example program, there will be one configuration for every function call. The fixpoint discovers these roughly in evaluation order, with typically one new configuration (and some new store entries) discovered per iteration of the monotone function: the fixpoint begins with the state only containing the outermost application (the binding for `f`). Recall that each iteration, the naïve fixpoint recomputes every piece of work that occurred previously.

First it binds `f`, then `y0`, and then begins to evaluate `f` applied to the first set of arguments. As it does so, it evaluates the outer and inner applications of `(λ a ka . ka a) x` …, then binds `x'`, and does the same for `(λ b kb . kb b) y` … and `y'`. It then performs `y' x' k`, returning the result `x1'`.

Next, the fixpoint begins to call `f` again using the other set of arguments. This iteration doesn't produce any new configurations, as the body of `f` has already been explored, but it does produce updates to the store, particularly to the binding for `x`: `x` now maps to both `(λ x1 kx1 . kx1 x1)` and `(λ x2 kx2 . kx2 x2)`. In the next iteration, the configuration that provides `x` as an argument to `(λ a ka . ka a)` now passes along both of these values to `a`, and then on the next iteration to `x'`.

When the next fixpoint iteration examines the configuration for `y' x' k`, the updated values for `x'` will now get passed along to `y0`, and then finally returned to both `x1'` and `x2'` (as both are possible return points from the function `f`). Finally, the fixpoint iteration attempts to call the top-level continuation `k0` on the final result of the program `x2'`, and no new configurations or store entries are discovered (as the top-level continuation `k0` does not map to any closure).

There are several clear points of inefficiency in this fixpoint computation, such as the recomputation of every configuration under the store every single iteration, which the derivative makes progress towards improving, as well as some less obvious sources of inefficiency that will be made clear after we describe the derivative.

In the description of the fixpoint calculation earlier, we describe only what is *new* or *changed* each iteration. Ideally, the final fixpoint procedure we obtain for the analysis will perform the same sequence of computations with *no* extra work, and this is indeed what is achieved.

An inefficient fixpoint procedure derived from set-of-states. Before we move on to the derivative-based approach to the above fixpoint, we want to briefly rewind to the set-of-states approach to compare with the fixpoint procedure above. As described towards the beginning of this section, the store-widened analysis presented above is derived from a set-of-states approach. A natural question to ask is whether we can derive an efficient fixpoint procedure from the efficient fixpoint for a set-of-states analysis.

$$
\begin{array}{rl}
 & \textbf{join}\ (\textbf{big-join}\ \Delta \mathit{inner}\ \textbf{for}\ \mathit{config}\ \textbf{in}\ \mathit{configs}) \\
\Delta\mathcal{S}(\mathit{state}, \Delta\mathit{state}) = & \textbf{join}\ (\textbf{big-join}\ \mathit{inner}\ \textbf{for}\ \mathit{config}\ \textbf{in}\ \Delta\mathit{configs}) \\
 & \quad (\textbf{big-join}\ \Delta \mathit{inner}\ \textbf{for}\ \mathit{config}\ \textbf{in}\ \Delta\mathit{configs}) \\
\text{where } \mathit{configs}, \mathit{store} = & \textbf{proj}_1\ \textbf{var}\ \mathit{state}, \textbf{proj}_2\ \textbf{var}\ \mathit{state} \\
\Delta\mathit{configs}, \Delta\mathit{store} = & \textbf{proj}_1\ \textbf{var}\ \Delta\mathit{state}, \textbf{proj}_2\ \textbf{var}\ \Delta\mathit{state} \\
\langle \underline{f\ arg}, \mathit{env}\rangle = & \mathit{config} \\
\mathit{val}_f, \mathit{val}_{arg} = & \mathcal{E}(\underline{f}, \mathit{env}, \mathit{store}), \mathcal{E}(\underline{arg}, \mathit{env}, \mathit{store}) \\
\Delta\mathit{val}_f, \Delta\mathit{val}_{arg} = & \mathcal{E}(\underline{f}, \mathit{env}, \Delta\mathit{store}), \mathcal{E}(\underline{arg}, \mathit{env}, \Delta\mathit{store}) \\
\mathit{inner} = & \textbf{big-join}\ (\textbf{pair}\ (\textbf{set-singleton}\ \{\mathit{config}'\})\ \mathit{store}')\ \textbf{for}\ \mathit{cl}\ \textbf{in}\ \mathit{val}_f \\
 & \quad \textbf{join}\ (\textbf{big-join}\ (\textbf{pair}\ \textbf{bot}\ \Delta\mathit{store}')\ \textbf{for}\ \mathit{cl}\ \textbf{in}\ \mathit{val}_f) \\
\Delta\mathit{inner} = & \textbf{join}\ (\textbf{big-join}\ (\textbf{pair}\ (\textbf{set-singleton}\ \{\mathit{config}'\})\ \mathit{store}')\ \textbf{for}\ \mathit{cl}\ \textbf{in}\ \Delta\mathit{val}_f) \\
 & \quad (\textbf{big-join}\ (\textbf{pair}\ \textbf{bot}\ \Delta\mathit{store}')\ \textbf{for}\ \mathit{cl}\ \textbf{in}\ \Delta\mathit{val}_f) \\
\langle \underline{\lambda x.e}, \mathit{env}_{cl}\rangle = & \mathit{cl} \\
a = & \ldots \text{abstract address for } \underline{x} \ldots \\
\mathit{store}' = & \textbf{map-singleton}\ [a \mapsto \mathit{val}_{arg}] \\
\Delta\mathit{store}' = & \textbf{map-singleton}\ [a \mapsto \Delta\mathit{val}_{arg}] \\
\mathit{config}' = & \langle \underline{e}, \mathit{env}_{cl}[\underline{x} \mapsto a]\rangle
\end{array}
$$

Fig. 8. Derivative of the abstract interpretation example.

The efficient fixpoint for the set-of-states essentially performs a graph traversal such as breadth-first search implemented as a work-set procedure. We believe it *is* natural to attempt a modification of this strategy, as previous work on optimizing higher-order program analysis such as Glaze et al. [2013] takes this approach: in the store-widened approach, we can still maintain a work-set of configurations, where whenever a configuration is processed, its successor configurations are put into the work-set (if they haven't already been explored under the current store).

For this kind of fixpoint, when propagating a change to the inputs of a function in the analysis, such as the second call to `f` in the example program, the entire body will need to be recomputed in order to revist the configuration that "returns" from the function. This means that the application `(λ b kb . kb b) y` ... will be re-evaluated under the exact same inputs as the previous time `f` was called, which doesn't occur with the (idealized) fixpoint calculation based on the monotone analysis function. In our evaluation (Section 6), we compare this modified set-of-states fixpoint to the final derivative-based fixpoint.

By explicitly writing out the monotone analysis function for the store widened analysis (which is often not done for abstract interpretation papers such as in Gilray et al. [2016]), we can derive more efficient fixpoints in a principled manner using derivatives and the optimizations described in the rest of this paper.

4.1 Derivative of the Analysis

The derivative $\Delta\mathcal{S}$ of the analysis function $\mathcal{S}$ appears in Figure 8, and was derived via the method in Figure 4. The derivative is a function of both the pair of the current set of configurations *configs* and store *store* as well as the pair of the change to the current set of configurations Δ*configs* and the change to the store Δ*store*. Because the derivative of a big-join split into three big-joins, the outermost form of the derivative consists of three big-joins, which process changes to the set of configurations and changes to the *inner* form. The first of these explores how every old configuration changes with respect to the store updates, the second explores each new configuration

under the old store, and the third explores each new configuration under the store updates. It is incremental in the sense that the old configurations are never explored under the old store.

The *inner* form is the same as previous, and Δ*inner* is the change for *inner* based on the store changes Δ*store*. As *inner* is a big-join, Δ*inner* consists of three big-joins, which explore (1) how the store updates affect previously known called closures (2) newly-found called closures via the store update whose argument comes from the old store, and (3) newly-found called closures whose argument comes from the store updates.

Consider how the fixpoint computation for the example changes. In the example program, the "changes" that occur in the first part of the fixpoint consist of a new configuration along with store updates, which are really only used by those new configurations (because they are newly-bound variables). In the second part of the fixpoint, an *updated* value for the variable `x` is found after the second call to `f`, and the rest of the fixpoint is spent propagating this change through various bindings. Even in the derivative, in each iteration the entire set of old configurations are still explored, just under the much smaller *change* to the store, even if most of those configurations do not use the update. In order to achieve an efficient fixpoint procedure, we'll need to optimize the derivative.

We require two primary optimizations. First, we need to *specialize* the derivative. Consider the case where the change consists of a new configuration but no change to the store; the store update is vacuous, but the old configurations will still be processed. We want a specialization of the derivative that processes a particular kind of change (when the store update is empty) more efficiently. Although this behavior doesn't occur for the particular example program, it can often occur in a more expressive analysis language that contains side effects, where a "set" operation will still cause a new configuration to be explored (the program location after the "set" operation), but won't necessarily induce a change to the store. For example, if numeric values are represented by a singular abstraction "`NUMBER`," the contents of an array might be updated from `NUMBER` to `NUMBER` during the "set" operation, which does not yield any change to the store.

Specialization of the derivative, explored in the following section, also paves the way for the most impactful optimization, which is tracking dependencies. Later in Section 5, we will describe a notion of *dependency* that allows the fixpoint to track which computations rely on which *kinds* of updates. For the fixpoint above, processing an update to the store should *only* involve processing configurations that use that update.

4.2 Specialized Derivatives

In the following, we study specializing the derivative to particular kinds of changes, using the abstract interpretation above as motivation. In particular, we study *small* changes: instead of trying to process an entire collection of new configurations and store updates each iteration, we can instead process just a single new configuration or a single store update. Processing small changes allows one to derive a *work-set* -style fixpoint procedure, which, in addition to the benefits of being able to specialize the derivative for a particular kind of change, allows fine control over the order in which changes are processed.

In the example abstract interpretation, the lattice is a pair of a powerset lattice on configurations and a map for the store. In general, for a pair, small changes are either small changes to the first component or small changes to the second. Small changes for a powerset lattice means a change of exactly one element, and small changes for a map lattice consist of updating one address with a small element.

We can define an operator δ on lattice types that provides small elements:

$$\delta(\mathbf{Pair}\ L_1\ L_2) \cong \delta(L_1) + \delta(L_2) \qquad \delta(\mathbf{Set}\ U) \cong U \qquad \delta(\mathbf{Map}\ U\ L) \cong U \times \delta(L)$$

$$
\begin{array}{lll}
\delta_{configs}\mathcal{S}(state, \{\Delta config\}) & = \textbf{big-join } inner \textbf{ for } config \textbf{ in set-singleton } \{\Delta config\} \\
\text{where } store & = \textbf{proj}_2 \textbf{ var } state \\
\quad \langle \underline{f\ arg}, env\rangle & = config \\
\quad val_f, val_{arg} & = \mathcal{E}(\underline{f}, env, store), \mathcal{E}(\underline{arg}, env, store) \\
\quad inner & = \textbf{big-join } (\textbf{pair } (\textbf{set-singleton } \{config'\})\ store') \textbf{ for } cl \textbf{ in } val_f \\
\quad \langle \underline{\lambda x.e}, env_{cl}\rangle & = cl \\
\quad a & = \ldots \text{abstract address for } \underline{x} \ldots \\
\quad store' & = \textbf{map-singleton } [a \mapsto val_{arg}] \\
\quad config' & = \langle \underline{e}, env_{cl}[\underline{x} \mapsto a]\rangle
\end{array}
$$

$$
\begin{array}{lll}
\delta_{store}\mathcal{S}(state, \Delta store) & = \textbf{big-join } \Delta inner \textbf{ for } config \textbf{ in } configs \\
\text{where } configs, store & = \textbf{proj}_1 \textbf{ var } state, \textbf{proj}_2 \textbf{ var } state \\
\quad \langle \underline{f\ arg}, env\rangle & = config \\
\quad val_f, val_{arg} & = \mathcal{E}(\underline{f}, env, store), \mathcal{E}(\underline{arg}, env, store) \\
\quad \Delta val_f, \Delta val_{arg} & = \mathcal{E}(\underline{f}, env, \Delta store), \mathcal{E}(\underline{arg}, env, \Delta store) \\
& \quad \textbf{join } (\textbf{big-join } (\textbf{pair bot } \Delta store') \textbf{ for } cl \textbf{ in } val_f) \\
\quad \Delta inner & = \textbf{join } (\textbf{big-join } (\textbf{pair } (\textbf{set-singleton } \{config'\})\ store') \textbf{ for } cl \textbf{ in } \Delta val_f) \\
& \quad\quad (\textbf{big-join } (\textbf{pair bot } \Delta store') \textbf{ for } cl \textbf{ in } \Delta val_f) \\
\quad \langle \underline{\lambda x.e}, env_{cl}\rangle & = cl \\
\quad a & = \ldots \text{abstract address for } \underline{x} \ldots \\
\quad store' & = \textbf{map-singleton } [a \mapsto val_{arg}] \\
\quad \Delta store' & = \textbf{map-singleton } [a \mapsto \Delta val_{arg}] \\
\quad config' & = \langle \underline{e}, env_{cl}[\underline{x} \mapsto a]\rangle
\end{array}
$$

Fig. 9. Specialized derivatives of the abstract interpretation example.

These elements are the *join-irreducible* elements of the lattice: v is join-irreducible if it cannot be "split" into non-trivial v_1 and v_2, such that $v = v_1 \sqcup v_2$. In other words, v cannot be decomposed further.

Applying this to the abstract interpretation example, we have that the small changes consist of either one new configuration $\Delta state = \langle \{\Delta config\}, \bot\rangle$, or an update to one store address $\Delta state = \langle \bot, [\Delta a \mapsto \{\Delta cl\}]\rangle$.

If we fix a change Δv for the derivative Δe, we can specialize Δe based on that Δv. Specialization primarily takes the form of partial evaluation and substitution. For example, if we know that $\Delta v = \langle \bot, \cdot\rangle$ then we can rewrite

$$\textbf{proj}_1\ (\textbf{var } \Delta x) \rightsquigarrow \textbf{bot}$$

After, we can perform simplifying rewrites to propagate the **bot** upwards. For example, all the following can be rewritten to just **bot**.

$$
\begin{array}{cc}
\textbf{join bot } e & \textbf{join } e \textbf{ bot} \\
\textbf{pair bot bot} & \textbf{proj}_i \textbf{ bot} \\
\textbf{big-join } e \textbf{ for } u \textbf{ in bot} & \textbf{big-join bot for } u \textbf{ in } e \\
\textbf{map-singleton } [d \mapsto \textbf{bot}] & \textbf{map-lookup bot}(d)
\end{array}
$$

Figure 9 contains specialized derivatives for the abstract interpretation example: the first of these $\delta_{configs}\mathcal{S}(state, \{\Delta config\})$ is specialized on configuration updates and the second of these $\delta_{store}\mathcal{S}(state, \Delta store)$ is specialized on store updates. We use the δ symbol to indicate a specialization.

In the following discussions, we recommend referring to Figure 8 to compare how the derivative is transformed into its specialized form(s) in Figure 9.

First, let us focus on the one that processes a new configuration $\delta_{configs}\mathcal{S}(state, \{\Delta config\})$. To specialize, we begin by noting that $\Delta store$ is empty, and so any lookups that reference it are empty too, in particular Δval_f and Δval_{arg} are both empty, if both are variables. Additionally, the new store update $\Delta store' =$ **map-singleton** $[a \mapsto \Delta val_{arg}]$ is also empty because the map takes everything to $\bot$. Propagating these observations into the $\Delta inner$ term yields that the entire term evaluates to $\bot$ as well: the first big-join always produces a pair of $\bot$ (which is $\bot$ for the product lattice), and joining over $\bot$ always produces $\bot$. The second and third big-joins both operate on the empty set, and so evaluate trivially to $\bot$. In the case that both are not variables, then the computation performed by the $\Delta inner$ term still doesn't perform any *new* work, because it doesn't reference the only change $\Delta config$.

Since $\Delta inner$ is in fact vacuous, two of the three big-joins at the top level of the derivative are vacuous as well! Then, if desired, the final top-level big-join could be reduced to just *inner* (substituting $\Delta config$ for *config*). We do not do this for the purpose of later discussion in Section 5. The result of these transformations (in Figure 9) is that the specialized derivative expression simply computes the *inner* term for the new configuration.

This small specialization has the potential to significantly improve performance: calling the original $\Delta\mathcal{S}$ with an empty $\Delta store$ still explores $\Delta inner$ for each old configuration — which is an entirely meaningless loop!. An implementation which does not make this specialization will re-explore every configuration, every time the derivative is evaluated. If the program analyzed by the abstract interpreter tended to explore new configurations rather than propagate updates to the store, this specialization would be very impactful.

The second specialization $\delta_{store}\mathcal{S}(state, \Delta store)$ is for a particular store update $\Delta store = [\Delta a \mapsto \{\Delta cl\}]$. In this case $\Delta configs$ is empty, and so two of the three outer big-joins can be eliminated immediately. We are left with the case where each old configuration is processed under the change to the store **big-join** $\Delta inner$ **for** *config* **in** *configs*. Unlike in the configuration specialization, the three big-joins in $\Delta inner$ must stay, because the change to the store may affect just the argument, just the function, or both.

The following observation is the subject of the most impactful optimization, developed in Section 5: for a particular store update $[\Delta a \mapsto \{\Delta cl\}]$, only a few configurations in *configs* actually *depend* on that store update. For example, if the addresses $env(f)$ and $env(arg)$ (assuming they are variables) are not the same as Δa, then both Δval_f and Δval_{arg} will evaluate to $\bot$, causing the entire $\Delta inner$ term to evaluate to $\bot$ (as was the case for the new configuration derivative specialization).

Essentially, for a particular $\Delta store$ the fixpoint calculation can track which configurations rely on that change, and only process those configurations accordingly in the specialized derivative. We note that in this case, such an optimization can also handle large changes: a large change to the store consists of small changes, and a configuration depends on the larger change if it contains a smaller change that the configuration depends on. In this case, determining dependencies still boils down to understanding which configurations depend on each small change. One key difficultly with this method is that the dependencies change over time: as more configurations are discovered, there are more dependencies that need to be tracked. In this sense we not only need to compute the derivative, but also *maintain* it as the fixpoint evolves.

In the following, we will first develop small-step fixpoint methods for processing small changes using specialized derivatives. Subsequently, Section 5 explores the dependency optimization in depth.

4.3 Small-step Fixpoint Method

A fixpoint that processes only small changes is a special case of a fixpoint that simply does not process all known changes at once, instead at each iteration choosing a portion of the known changes to process. The key word is *choose* — the actual implementation is tunable and different changes can be purposely processed before others.

Just like the derivative-based fixpoint algorithm (3.0.1), this fixpoint method tracks a change element, now called ws (for *workset*), and an accumulated v. Unlike the derivative-based fixpoint algorithm, the entirety of ws is not processed every step. Instead, a portion Δv of ws is extracted, leaving ws', and the derivative is evaluated on v and Δv. The result of the derivative is then joined back into the remaining ws' for processing in future iterations.

$$
\begin{array}{l}
\texttt{letrec } \mathit{fixpoint}(v, ws) = \\
\quad \texttt{let } \Delta v, ws' = \text{ choose and remove a portion from } ws \text{ with } ws' \sqcup \Delta v = ws \\
\quad \texttt{let } v' = v \sqcup \Delta v \\
\quad \texttt{let } ws'' = ws' \sqcup [\![\Delta_{x,\Delta x}\ e_0]\!]_{[x \mapsto v,\, \Delta x \mapsto \Delta v],[\,]} \texttt{ in} \\
\quad \texttt{if } v = v' \texttt{ then } v \\
\quad \texttt{else } \mathit{fixpoint}(v', ws'') \texttt{ in} \\
\mathit{fixpoint}(\bot, [\![e_0]\!]_{[x \mapsto \bot],[\,]})
\end{array}
\tag{4.3.1}
$$

The key invariant of the small-step fixpoint is that at every iteration, the evaluation of the original top-level term is contained within the current accumulated value v and the workset ws.

Theorem 4.1 (Small-step fixpoint). *The following is an invariant of the small-step fixpoint:*

$$[\![e_0]\!]_{[x \mapsto v],[\,]} \sqsubseteq v \sqcup ws.$$

Proof. By induction on the fixpoint sequence and applying the derivative constraint. □

Once it is the case that the workset is empty (or contained in v), then we have $[\![e]\!]_{[x \mapsto v],[\,]} \sqsubseteq v$, indicating the v is a fixpoint of the increasing function $v \sqcup [\![e]\!]_{[x \mapsto v],[\,]}$. Note that in order to terminate, each iteration must process a portion of the workset that is not yet included in v. Additionally, note that this fixpoint iteration does not necessarily provide the same increasing sequence as (2.3.1) and (3.0.1) due to the inherent choice of the change to propagate.

The "small-step" (in the sense that it does not process all changes at once) fixpoint method provides considerable flexibility and customization as to how the fixpoint is calculated. In particular, one choice is to focus on the "small changes" within ws for which we can specialize the derivative; in this case the various specializations would take the form of distinct implementations of $[\![\Delta_{x,\Delta x}\ e_0]\!]_{[x \mapsto v,\, \Delta x \mapsto \Delta v],[\,]}$.

A key benefit of the small-step fixpoint is also control over the *order* that changes are processed in. For example, in the abstract interpretation example it is more efficient to process all store updates before configuration updates: processing a new configuration $\Delta\mathit{config}$ before a store update $\Delta\mathit{store}$ means that the configuration is first explored under the old store, and then later under the store update, which involves processing $\Delta\mathit{config}$ twice (which will end up computing more joins overall). Instead, propagating the store update first means that the configuration is processed only once, under the store combined with the update.

Similarly, the order that store updates are applied in can matter as well: for a function call $\mathit{f}\ \mathit{arg}$, processing a change $\Delta\mathit{store}$ that affects the function f before a change $\Delta\mathit{store}'$ that affects the argument arg is inefficient, since just as in the configuration example above, the new closure for the function needs to be explored twice.

In general, the order that changes should be processed in is domain and implementation specific, but writing the fixpoint method in this workset-based small-step fashion yields an easily tunable algorithm.

5 Dependency Optimization

We explore the most impactful optimization, tracking dependencies, in the following section. To recap, recall that when computing the fixpoint for the example program (Figure 5), the first portion of the fixpoint involved exploring many new configurations, each of which produced store updates for newly-discovered bindings. These store bindings didn't impact any previous configurations, in the sense that re-exploring those configurations with those updates would lead to no new results. Additionally, the second part of the fixpoint involved propagating only store changes through already discovered configurations. In both of these cases, we want to track which configurations *depend* on which store changes so that when propagating a store change, we only visit the productive configurations.

Specifically, consider propagating some change to the store $\Delta store$ through each of the configurations. Each of these configurations represents a function call $f\ arg$ under an environment, and each function call references at most two addresses (or for multi-argument functions, some constant number), which is a very limited portion of the store. If $\Delta store$ does not update either of these addresses, then this function call will not see any change, since the $\Delta inner$ term in Figure 9 will evaluate to bottom.

The optimization is to track the dependencies between store updates and the configurations which use those updates. Then, when evaluating the derivative on a particular store update, only the tracked configurations need to be iterated over in the outer big-join. This changes the outer big-join over configurations to be a *sparse* big-join, which enables scalability.

In this case, because the set of configurations can change as the fixpoint is computed, the dependency information also changes (monotonically) over time: when processing a new configuration, we need to record which addresses that configuration relies on, which in turn describes which changes $\Delta store$ will affect this configuration.

In the following section, we outline how this kind of dependency information can be computed and automatically maintained for computations written in the $\mathcal{M}$ language.

5.1 What are Dependencies?

Fix a top-level term $x : L; \cdot \vdash e_0 : L$, which has one free variable x. The overall goal is to avoid doing useless computation in the derivative $\Delta_{x,\Delta x}\ e_0$, when processing a change Δv. Specialization can eliminate some computation (as discussed previously), but the large remaining problem is that when evaluating some big-join, **big-join** e_2 **for** u **in** e_1, some of the many evaluations of e_2, which depend on the actual choices for u, may be bottom. How can we determine which elements of the result of e_1 are actually "useful," and avoid computing e_2 on the remaining useless elements? To answer this question, we make some observations about the environments γ and σ the computations contained in the derivative evaluate under.

The first observation is that when evaluating the top-level term e_0, if we evaluate some subterm e under environments γ and σ, then when evaluating the top-level derivative term $\Delta_{x,\Delta x}\ e_0$ with some change Δv, we will evaluate the *derivative of the subterm*, $\Delta_{x,\Delta x}\ e$ under $\gamma[\Delta x \mapsto \Delta v]$ and the *same* non-lattice environment σ.

Theorem 5.1 (Observation 1). *When evaluating e_0 under $\gamma[x \mapsto v]$ and $[]$, if some subterm e is evaluated under $\gamma[x \mapsto v]$ and σ, then when evaluating the derivative $\Delta_{x,\Delta x}\ e_0$ under $\gamma[x \mapsto v, \Delta x \mapsto \Delta v]$ and $[]$, the subterm $\Delta_{x,\Delta x}\ e$ will be evaluated under $\gamma[x \mapsto v, \Delta x \mapsto \Delta v]$ and σ.*

Proof. By structural induction on terms e_0. The crucial case are big-joins, as these extend the non-lattice environment.

Recall that the derivative of a big-join yields three big-joins,

$$\Delta_{x,\Delta x}\ (\textbf{big-join}\ e_2\ \textbf{for}\ u\ \textbf{in}\ e_1) = \begin{array}{l}\textbf{join}\ (\textbf{big-join}\ e_2\ \textbf{for}\ u\ \textbf{in}\ \Delta_{x,\Delta x}\ e_1)\\ \textbf{join}\ (\textbf{big-join}\ \Delta_{x,\Delta x}\ e_2\ \textbf{for}\ u\ \textbf{in}\ e_1)\\ \quad(\textbf{big-join}\ \Delta_{x,\Delta x}\ e_2\ \textbf{for}\ u\ \textbf{in}\ \Delta_{x,\Delta x}\ e_1).\end{array}$$

The crucial insight is that the middle big-join, **big-join** $\Delta_{x,\Delta x}\ e_2$ **for** u **in** e_1, will evaluate the body term $\Delta_{x,\Delta x}\ e_2$ in exactly the non-lattice environments $\sigma[u \mapsto elm]$ that e_2 is evaluated in within the non-derivative term, since they both range over the same e_1 (that cannot refer to Δx). □

These same "second big-joins" are the computations we have a hope of optimizing in the derivative: the first and third big-joins operate over possibly never-before-seen elements (the result of $\Delta_{x,\Delta x}\ e_1$), but the second operate over elements and environments that have been processed before, *supposing* that the fixpoint computation has evaluated the top-level term e_0.

If we had some way to extract which changes Δv will make the evaluation of $\Delta_{x,\Delta x}\ e$ non-bottom under $\gamma[\Delta x \mapsto \Delta v]$ and σ, *only* by inspecting e under γ and σ, then when evaluating e_0 we could extract and track those changes.

Then, when computing the derivative for some Δv, we only bother evaluating the subterm $\Delta_{x,\Delta x}\ e$ under $\gamma[\Delta x \mapsto \Delta v]$ and σ if we tracked that it will be non-bottom.

Formally, the term e under γ and σ has a *dependency* on Δv if the derivative does not evaluate to bottom:

$$[\![\Delta_{x,\Delta x}\ e]\!]_{\gamma[\Delta x \mapsto \Delta v],\sigma} \neq \bot.$$

We assume the existence of a function,

$$depend(e, \gamma, \sigma) \to \mathcal{P}(L)$$

which computes the dependencies, which are a set of lattice elements, of e under γ and σ. Trivially, such a function exists (by enumerating all changes Δv, and checking evaluations). In practice, the implementer of the monotone function can make observations about the subterm e to understand what changes it relies on.

In general, it may be difficult to exactly characterize the dependent changes, so this function is allowed to *over-approximate* dependencies — as each dependency simply indicates that $\Delta_{x,\Delta x}\ e$ will be computed in the derivative, extra computations are always allowed (because they will always evaluate to bottom). Formally, $depend(e, \gamma, \sigma)$ must satisfy:

$$\forall \Delta v \notin depend(e, \gamma, \sigma), [\![\Delta_{x,\Delta x}\ e]\!]_{\gamma[\Delta x \mapsto \Delta v],\sigma} = \bot$$

Let us examine the example abstract interpretation derivative from Figure 8. From the discussion at the beginning of this section, we know that the *inner* term under some configuration *config* will rely on (at most) two addresses, and so the dependencies are with any change to the state space $\Delta state$ that updates one of those addresses. In the example, this occurs because non-bottomness of the *inner* term depends on the non-bottomness of two map-lookups (for the function's address and the argument's address in the change), which each directly reference the input $\Delta state$. We leave further discussion of the general problem of finding which changes Δv have a dependency with e under environments γ and σ to future work.

To recap, if we evaluate any subterm e under γ and σ while evaluating the top-level term e_0, then we will evaluate the subterm $\Delta_{x,\Delta x}\ e$ under $\gamma[\Delta x \mapsto \Delta v]$ and σ when evaluating the top-level derivative term, $\Delta_{x,\Delta x}\ e_0$. Supposing we have such a $depend(e, \gamma, \sigma)$ function, we can determine and track whether we need to actually run that computation in the derivative, or not (as the absence of a dependency indicates a bottom valuation).

Before that, we need to address the remaining issue: we never actually evaluate the top-level term e_0. In the derivative-based fixpoint methods, we *only* evaluate the derivative $\Delta_{x,\Delta x}\ e_0$. It turns out that just as the derivative captures the change between *evaluations* of a term on a larger input, computations the derivative performs are actually the *changes in computation*. Applying this idea to the fixpoint loop, all of the computations in $[\![x]\!]_{[x \mapsto v],[]}$ are actually represented by the accumulated computations in the evaluated derivatives.

Formally, we need to relate each of the evaluations performed in e_0 under the *increased* environments, $[\![e_0]\!]_{[x \mapsto v \sqcup \Delta v],[]}$, to the evaluations performed in e_0 under the *original* environments, $[\![e_0]\!]_{[x \mapsto v],[]}$, and the evaluations $[\![\Delta_{x,\Delta x}\ e_0]\!]_{[x \mapsto v, \Delta x \mapsto \Delta v],[]}$ performed when propagating Δv.

It is the case that for any subterm e, evaluating

$$[\![e]\!]_{[x \mapsto v \sqcup \Delta v],\sigma}$$

in $[\![e_0]\!]_{[x \mapsto v \sqcup \Delta v],[]}$ means first that we will evaluate

$$[\![\Delta_{x,\Delta x}\ e]\!]_{[x \mapsto v,\, \Delta x \mapsto \Delta v],\sigma}$$

in the derivative, and second that either

$$[\![e]\!]_{[x \mapsto v],\sigma}$$

is evaluated in $[\![e_0]\!]_{[x \mapsto v],[]}$ or

$$[\![e]\!]_{[x \mapsto v,\, \Delta x \mapsto \Delta v],\sigma}$$

is evaluated in the derivative (noting that e does not refer to Δx). The disjunction occurs because the value $v \sqcup \Delta v$ may explore *new* environments in e_0 compared to just v: the old environments will be explored by e_0, and the new environments will be explored by the derivative.

Theorem 5.2 (Observation 2). *While evaluating e_0 under $\gamma[x \mapsto v \sqcup \Delta v]$ and σ, if some subterm e is evaluated under $\gamma[x \mapsto v \sqcup \Delta v]$ and σ', then while evaluating $\Delta_{x,\Delta x}\ e_0$ under $\gamma[x \mapsto v, \Delta x \mapsto \Delta v]$ and σ, the subterm $\Delta_{x,\Delta x}\ e$ will be evaluated under $\gamma[x \mapsto v,\ \Delta x \mapsto \Delta v]$ and σ'.*
Additionally, while evaluating e_0 under $\gamma[x \mapsto v]$ and σ, either the subterm e will be evaluated under $\gamma[x \mapsto v]$ and σ', or while evaluating $\Delta_{x,\Delta x}\ e_0$ under $\gamma[x \mapsto v, \Delta x \mapsto \Delta v]$ and σ, the subterm e will be evaluated under $\gamma[x \mapsto v,\ \Delta x \mapsto \Delta v]$ and σ'.

Proof. By induction on terms e_0.

As before, the main point of interest are big-joins: when evaluating **big-join** e_2 **for** u **in** e_1 under the extended environment, we know from the derivative constraint that

$$[\![e_1]\!]_{[x \mapsto v],\sigma} \sqcup [\![\Delta_{x,\Delta x}\ e_1]\!]_{[x \mapsto v,\, \Delta x \mapsto \Delta v],\sigma} = [\![e_1]\!]_{[x \mapsto v \sqcup \Delta v],\sigma}$$

and the same for e_2, meaning we can break the evaluation up as

$$\begin{aligned}[\![\textbf{big-join}\ e_2\ \textbf{for}\ u\ \textbf{in}\ e_1]\!]_{[x \mapsto v \sqcup \Delta v],\sigma} = {} & \left([\![\textbf{big-join}\ e_2\ \textbf{for}\ u\ \textbf{in}\ e_1]\!]_{[x \mapsto v],\sigma}\right) \\ & \sqcup \left([\![\textbf{big-join}\ e_2\ \textbf{for}\ u\ \textbf{in}\ \Delta_{x,\Delta x}\ e_1]\!]_{[x \mapsto v,\, \Delta x \mapsto \Delta v],\sigma}\right) \\ & \sqcup \left([\![\textbf{big-join}\ \Delta_{x,\Delta x}\ e_2\ \textbf{for}\ u\ \textbf{in}\ e_1]\!]_{[x \mapsto v,\, \Delta x \mapsto \Delta v],\sigma}\right) \\ & \sqcup \left([\![\textbf{big-join}\ \Delta_{x,\Delta x}\ e_2\ \textbf{for}\ u\ \textbf{in}\ \Delta_{x,\Delta x}\ e_1]\!]_{[x \mapsto v,\, \Delta x \mapsto \Delta v],\sigma}\right)\end{aligned}$$

where the first term comes from the non-derivative, and the latter three come from the derivative. Of the four, the first covers the first case of the "either" the second covers the second case, the last two cover the first portion of the theorem before the either. □

Let us examine the relation of evaluations under the increased environments to evaluations under the original environments and evaluations in the derivative, from the point of view of *dependencies*. We want the ability to determine the *dependencies* in the increased environments via

the evaluations under the original environments and evaluations in the derivative. This turns out to be straight-forward; inspecting the derivative constraint (Equation 3.1.1), for the evaluations of e,

$$[\![e]\!]_{[x \mapsto v],\sigma} \sqcup [\![\Delta_{x,\Delta x}\ e]\!]_{[x \mapsto v,\, \Delta x \mapsto \Delta v],\sigma} = [\![e]\!]_{[x \mapsto v \sqcup \Delta v],\sigma},$$

we can push the derivative into each of these components (which involves the *second* derivative),

$$\begin{aligned}[\![\Delta_{x,\Delta x'}\ e]\!]_{[x \mapsto v,\, \Delta x' \mapsto \Delta v'],\sigma} \sqcup [\![\Delta_{x,\Delta x'}\ (\Delta_{x,\Delta x}\ e)]\!]_{[x \mapsto v,\, \Delta x \mapsto \Delta v,\, \Delta x' \mapsto \Delta v'],\sigma} \\ = [\![\Delta_{x,\Delta x'}\ e]\!]_{[x \mapsto v \sqcup \Delta v,\, \Delta x' \mapsto \Delta v'],\sigma}\end{aligned}$$

and the equality still holds because the derivative constraint holds for all terms and environments! This equation is exactly what we need for dependencies: if the right-hand side is non-bottom, then one of the two left-hand sides will also be non-bottom. So, knowing the dependencies $\Delta v'$ of e under $[x \mapsto v]$ and σ as well as the dependencies of $\Delta_{x,\Delta x}\ e$ under $[x \mapsto v,\ \Delta x \mapsto \Delta v]$ and σ means that we know the dependencies in the increased environment e under $[x \mapsto v \sqcup \Delta v]$ and σ.

The fact that the second derivative makes an appearance is interesting, but very natural, as we are tracking how dependencies $\Delta v'$ change as the Δv is propagated through the computation. In general, dependency information changes over time and we must maintain it; for example, take Δv to be a new configuration and $\Delta v'$ to be a store update from the abstract interpretation example. When propagating the new configuration in the derivative, we explore a new environment σ. We then look at the dependencies for the *inner* subterm under that new environment, and see that there is a dependency on the store change Δv. The dependency in the derivative can be directly interpreted as a dependency in the main term.

To summarize, while computing the small-step fixpoint loop (4.3.1), the collection of evaluated derivatives represents the accumulated *computations* that would be computed by $[\![e_0]\!]_{[x \mapsto v],[\,]}$, which provide the dependencies for future changes.

5.2 Implementing Dependencies

Earlier we alluded to being able to track dependencies and then use them when evaluating the derivative. Specifically for dependencies, we can introduce a new monotonic form to $\mathcal{M}$,

$$e ::= \ldots \quad | \quad \textbf{big-join}\ e_2\ \textbf{for}\ u\ \textbf{in}\ set(u_1, \ldots, u_k)$$

where $set(u_1, \ldots, u_k)$ are a collection of already computed sets, indexed by the non-lattice variables in scope: for each choice of $u_1, \ldots, u_k$ there is a (possibly empty) set. These sets have this argument because of nested big-joins: in

$$\textbf{big-join}\ (\textbf{big-join}\ e_3\ \textbf{for}\ u_2\ \textbf{in}\ e_2)\ \textbf{for}\ u_1\ \textbf{in}\ e_1,$$

the elements of e_2 that cause e_3 to have a non-bottom valuation depend on the choice of u_1. In the derivative specializations, we replace the subterms from Observation 1 (5.1) with this new form,

$$\textbf{big-join}\ \Delta_{x,\Delta x}\ e_2\ \textbf{for}\ u\ \textbf{in}\ e_1 \rightsquigarrow \textbf{big-join}\ \Delta_{x,\Delta x}\ e_2\ \textbf{for}\ u\ \textbf{in}\ set(u_1, \ldots, u_k)$$

where $set(u_1, \ldots, u_k)$ depends on the actual change Δv of the specialization. To maintain sparsity, only non-empty sets are tracked in an implementation: when evaluating $set(u_1, \ldots, u_k)$ that does not have a set associated with the choices of $\sigma(u_1), \ldots, \sigma(u_k)$, the empty set is produced.

An implementation maintains a map $\mathfrak{m}(\Delta v)$ from changes Δv to specialization of the derivative $\Delta_{x,\Delta x}\ e$ on that change. When evaluating the derivative $\mathfrak{m}(\Delta v)$, we may discover new dependencies between some $\Delta v'$ and a subterm e under $[x \mapsto v,\ \Delta x \mapsto \Delta v]$ and σ. First, we relate this evaluation on a subterm of $\Delta_{x,\Delta x}\ e$ under $[x \mapsto v,\ \Delta x \mapsto \Delta v]$ as an evaluation on subterm of e_0 under $[x \mapsto v \sqcup \Delta v]$, which Observation 2 (5.2) describes how to do. This is like interpreting $\Delta inner$ as the change to *inner* in the abstract interpretation example. Then, we interpret that dependency in e_0 under $[x \mapsto v \sqcup \Delta v]$ as a subcomputation that needs to occur in $\Delta_{x,\Delta x}\ e$ under

$[x \mapsto v, \Delta x \mapsto \Delta v']$, which Observation 1 (5.1) describes how to do. Once we interpret it this way, we can modify specialized derivatives $\mathfrak{m}(v')$ accordingly, updating the internal sets $set(u_1, \ldots, u_k)$ depending on the non-lattice environment σ of the dependency.

Dependencies in practice. An efficient implementation does not actually want to maintain a large map $\mathfrak{m}(\Delta v)$ on all possible changes; there are far too many possible changes for that. Here we see another power of focusing on *small changes*: instead of tracking dependencies and specializations for all kinds of changes, restricting to small changes means we have to maintain fewer entries in the map. This means that the dependency function only needs to produce small changes as well, $depend(e, \gamma, \sigma) \subseteq \mathcal{P}(\delta(L))$.

In the abstract interpretation example, an efficient implementation maintains a map from addresses to configurations rather than a mapping from store updates $[a \mapsto \{cl\}]$, as all store updates that update the same address will depend on the same set of configurations; the actual closure value does not matter. Additionally, it does not even need to track dependencies for new configurations, since they have no dependencies. Due to these two optimizations, the map $\mathfrak{m}(\Delta v)$ can be implemented efficiently.

Dependencies and large changes. It is not necessarily the case that in general (for arbitrary lattices and monotone computations) a dependency between $\Delta v \sqcup \Delta v'$ and e under γ and σ means that there is a dependency between either Δv and e under γ and σ or $\Delta v'$ and e under γ and σ. However, this does occur for the abstract interpretation example (and common extensions of it to more expressive languages): the dependencies of a collection of store changes is the union of the dependencies of each individual change. This means that in this case, even a big-step style fixpoint can benefit from dependencies by computing and maintaining them in the exact same manner, *i.e.*only tracking them for small changes, as in a small-step fixpoint.

5.3 The Second Derivative

Interestingly, using the second derivative provides a way to propagate multiple "small changes" at once, without worrying about dependencies, which we believe can yield new ways to improve the small-step fixpoints in future work.

Theorem 5.3 (Empty second derivative). *If the second derivative is bottom,*

$$[\![\Delta_{x,\Delta x'}\ (\Delta_{x,\Delta x}\ e_0)]\!]_{[x \mapsto v,\, \Delta x \mapsto \Delta v,\, \Delta x' \mapsto \Delta v'],[]} = \bot,$$

then Δv and $\Delta v'$ can be processed in parallel *in the small-step fixpoint (4.3.1), as*

$$[\![e]\!]_{[x \mapsto v],[]} \sqcup [\![\Delta_{x,\Delta x}\ e]\!]_{[x \mapsto v,\, \Delta x \mapsto \Delta v],[]} \sqcup [\![\Delta_{x,\Delta x'}\ e]\!]_{[x \mapsto v,\, \Delta x' \mapsto \Delta v'],[]} = [\![e]\!]_{[x \mapsto v \sqcup \Delta v' \sqcup \Delta v],[]}.$$

Proof. By repeated applications of the derivative constraint. □

6 Implementation and Evaluation

We implemented the abstract state-space construction in the 3CPS Standard ML compiler [Quiring et al. 2022], which compiles a substantial subset of the Standard ML language.[2] The abstract interpretation is essentially an extension of the store-widened λ-calculus from Section 4. The compiler first computes the abstract state-space, and then uses it in several secondary passes to extract facts used for optimizations.

We compared two methods that generate the state-space. The first uses a derivative-based method with the dependency optimization, as explored in Sections 4 and 5. The second uses the adapted

[2]The 3CPS implementation supports almost all of the core language, including references and exceptions, as well as structures and nested structures. The main missing features are record types, signatures, and functors.

Table 1. Benchmark results

Program	# Lines	Derivative with dependencies (ms)	Adapted set-of-states fixpoint (ms)	Speedup
nucleic	3631	69	3332	48x
boyer	888	36	2049	57x
k-cfa	843	16	1651	103x
ratio-regions	707	13	2256	174x
mc-ray	924	3	137	46x
knuth-bendix	607	25	2590	104x
raytracer	456	6	329	55x
smith-normal-form	449	2	145	73x
tsp	317	3	55	18x
cps-convert	277	3	133	44x
interpreter	252	8	246	31x
parser-comb	247	8	313	39x
life	150	4	55	14x
derivative	165	4	160	40x
quicksort	50	2	13	7x

set-of-states fixpoint as described in Section 4 and Glaze et al. [2013]. Essentially, this method performs a graph traversal on the graph of *configurations*. Unlike the set-of-*states* approach in the non-widened setting configurations may be visited multiple times, as store may have been updated since the last time the configuration was visited. If, while processing a configuration, the store hasn't been updated from the previous time the configuration has been processed, then the fixpoint stops exploring that branch, just as a graph-traversal doesn't re-traverse already-explored vertices.

Both of these implementations are hand-optimized: both methods employ several domain-specific tricks in the abstract interpretation to improve performance — but these tricks are common between the two. Precisely, we tested using different kinds of data structures (ordered maps, hash maps, primitive arrays, hash consing, *etc.*) for each parts of the analysis (the store, environments, configurations, addresses, *etc.*). For the store, we use a primitive array, which requires addresses having a unique integer index. For environments, ordered maps were most effective. The analysis is a variant of 0CFA, with the exception that as the analysis is focussed on control-flow, each kind of non-control data (integers, strings, booleans, *etc.*) each have a single unique address; variables that map to such kinds of values must use that address, instead of the normal 0CFA addresses. The effect of this optimization is that the store becomes much more sparse, which affects programs that are heavy in *e.g.*floating point and string processing operations. Algebraic data structures such as lists are not abstracted in this way.

We measured the speed-up between these two methods; Table 1 contains the evaluation results, which were taken as the average over ten runs on a machine with a 12$^{\text{th}}$ Gen Intel i7 with 32GB RAM running Ubuntu 22.

On average the derivatives method provided a 56x speed-up, with the average speed-up increasing for larger programs, up to 174x. As described in Section 4, the reason why the derivative-based fixpoint is so much more effective is because of code that looks like the following program fragment.

```
let x = f arg1 in
let y = g arg2 in
x + y
```
(6.0.1)

If the value for `arg1` is updated, then `f` must be recomputed, which may update the value for `x`. The derivative based approach can then jump straight to `x + y` in the next iteration, whereas the

set-of-configurations approach must re-explore `g arg2` — potentially a very expensive journey! — before arriving at `x + y`.

Note that the optimizations described in this work, particularly the dependency optimization, are what make the derivative-based approach efficient. Without this optimization, the derivative-based approach, the naïve derivative would, upon receiving a store update for `x`, explore every single known configurations under that store update, even if those configurations don't use that update. Using the dependencies, the analysis knows upon seeing an update to `x`, that it should immediately jump to where `x` is *used*. Without the dependency tracking, our implementation effectively does not terminate: processing the entire set of configurations each step is much too inefficient. There are several additional factors contributing to this speed-up.

The workset order matters. The order elements are processed in from the workset matters. The analysis implementation has two worksets: one for configurations, and one for the store maintained by the analysis; and processing the various store updates before the new configurations is indeed the most performant order, the reasoning behind which is given in Section 4.3.

Processing larger store updates. The analysis processes all store updates $\Delta store$ that share the same address at the same time, which are not the smallest possible changes. This avoids extra joins and iterations over the same configurations.

7 Limitations and Extensions

This work intentionally presents a "minimal" language to demonstrate why taking a small-step and derivative specialization approach to fixpoints is beneficial. The language $\mathcal{M}$ contains enough lattices to describe the higher-order flow analysis example, which motivates our work.

Extending this language and the theory to account for other monotone expressions, such as meets, is easily accomplished. However, handling certain kinds of monotone expressions may be more complex than others. For example, Arntzenius [2017] introduces a way to take derivatives of fixpoints themselves, which opens the door to having nested and language-internal fixpoints, and Arntzenius and Krishnaswami [2019] introduces higher-order monotone forms. Unfortunately, there are some potential issues with nested fixpoints (see Arntzenius and Krishnaswami [2019] for discussion), and it is likely that these forms complicate dependency analysis.

Introducing a form of lattice-variable `let` binding into $\mathcal{M}$, such as

$$\textbf{let } y = e_1 \textbf{ in } e_2$$

in principle does not cause potential issues, as the value of the variable bound by such a `let` can just be substituted. However, it can greatly complicate the presentation of the math, since the derivative would need to propagate multiple new variables through the term (carrying around an entire context), along the lines of

$$\Delta_{x,\Delta x}\ (\textbf{let } y = e_1 \textbf{ in } e_2) = \textbf{let } y = e_1 \textbf{ in let } \Delta y = \Delta_{x,\Delta x;y,\Delta y}\ e_1 \textbf{ in } \Delta_{x,\Delta x;y,\Delta y}\ e_2.$$

The term e_1 needs to be kept around because the derivative of e_2 may still refer to it (for example, due to big-joins). And in general, if some subterm e was evaluated in the environment γ, then the derivative will be evaluated in an environment that has one new "Δ" variable for every let-bound value in scope at e.

Introducing "conditional" forms such a set-membership test or set-filtering add more opportunities for dependencies. In the current language $\mathcal{M}$, the source of dependencies are map-lookups **map-lookup** $e(d)$, because the bottom-valuedness of this term depends on the actual value of d (and not just the bottom-valuedness of e). Other forms allow for new ways for terms to depend on

non-lattice values, and so determine which changes Δv a computation depends on may become more difficult.

Additional lattices. Introducing other lattices, such as intervals, should be possible as *changes* to such values can be propagated through a monotone computation; the derivative for joins and for variables holds no matter what the lattice is. However: widening operators are commonly seen with such (infinite) lattices, and are often non-monotone, which may cause deeper correctness issues with the derivative approach described in this paper. The infinite lattices themselves aren't necessarily an issue, as key theorems in this paper (Theorems 3.1 and 4.1) are invariants on the increasing sequences of the fixpoints (even if those increasing sequences are infinite) and others (Theorems 5.1 and 5.2) are about computations internal to a single iteration; we believe monotone functions over these lattices do not pose an issue. Issues with widening operators can likely be solved (after all, efficient and sound analyses are constructed with them) but may require effort specific to the widening operator and lattice. Additionally, the dependency analysis described in this paper is still amenable to such lattices, as the program pattern in 6.0.1 has nothing to do with the lattice type of program values at all.

8 Related Work

Eo [2004] introduces fixpoint methods and derivatives that are very similar to some of those described in this work, though do not specifically handle powerset or map lattices. Their work defines monotone forms for lattice meets, lattice **let**s, and conditionals, which can be easily introduced into the monotone language $\mathcal{M}$ this work describes, along with their derivatives. They define a similar workset-based fixpoint method that handles small changes, but they do not discuss how this can be used to specialize the derivative computations, process both small and large changes in the same fixpoint, or track dependencies. The focus of this work is not to introduce novel monotone forms, but instead to build upon this style of derivative, and investigate how these kinds of optimizations can further improve fixpoint methods, particularly for higher-order program analyses.

In terms of incremental computation, the derivatives in this work are closely related to the semi-naïve evaluation [Bancilhon 1986] method used for implementing bottom-up Datalog predicates. Like the method in this paper, this algorithm pushes changes through monotonic functions. Traditional Datalog only computes over sets, but extensions such as Flix [Madsen et al. 2016] add user-defined lattices and monotone functions, along with an appropriate extension to the semi-naïve evaluation method. To verify properties such as monotonicity, Flix queries an SMT-based solver, whereas our the $\mathcal{M}$ language in this paper relies on syntactic partitioning to enforce approach to monotonicity. Datafun [Arntzenius and Krishnaswami 2019] is another extension that adds higher-order functions to a monotone language and a type system that distinguishes between monotone and non-monotone computations, as this paper does. Datafun focuses on expressivity, and is a complete generalization of $\mathcal{M}$, and handles a number of monotone forms not that $\mathcal{M}$ does not. This work does not seek to compete with Datafun's expressivity, but rather focuses on how the semi-naïve evaluation step can be made more efficient. With respect to this work, the increased expressivity of higher-order functions likely complicate any discussion of dependencies, as in the $\mathcal{M}$ language, environments (lattice and non-lattice) are very simple and so dependencies are easier to describe. However, just as compilers perform inlining and specialization, perhaps Datafun could be compiled to a simpler language such as $\mathcal{M}$ where more complex optimizations could be applied.

Fecht and Seidl [1998] focus on a similar kind of dependency optimization. They define "computation trees" which are essentially a data-structure representing the computations the fixpoint performs. These computations refer to "free variables" which are the "dependencies". The notion of dependency introduced in Section 5 is more precise: the "computations" depend not just on

the variable, but also on the particular value that variable refers to. Relying on free variables is an over-approximation to the method we describe. In particular, it corresponds to taking the function $\mathit{depend}(e, \gamma, \sigma)$ to return all possible values for the free variable(s) of e. For example, the primary examples this work describes involving dependencies are map-lookups: referencing the CPS analysis, **map-lookup** *store*(*a*) depends on the variables *store* and *a*, but only when it is the case that *store* is defined at *a*. If *store* is updated, the fixpoint doesn't necessarily need to revisit this computation, for a particular value of *a*, because it might not have updated the particular value. Fecht and Seidl [1998] would recompute **map-lookup** *store*(*a*) whenever *store* changes.

Demanded abstract interpretation [Stein et al. 2021], treats static analysis as incremental relative to *program changes*: small changes in the source program do not always lead to large analysis changes. For example, Stein et al. [2021] unify program edits, abstract evaluation, and more using a dynamically updating state graph, providing an incremental approach to all of these. Other works such as Horwitz et al. [1995] and Szabó et al. [2016] use incremental control-flow graph -based approaches to static analysis. Such techniques are amenable to specialization, as program edits can be represented as a series of small changes.

Another technique for performant static analyses is the abstract compilation paradigm [Boucher and Feeley 1996] which constructs an analysis by viewing it as a curried function that takes in a static component (the program to be analyzed) and an initial environment. It then essentially performs partial evaluation on the input program to achieve performance. In such a scenario, all dependencies may be able to be statically known (given the program input). For example, one could write the classic bit-vector dataflow problems [Reps et al. 1995] using a big-join over successors of nodes, and dependency tracking would "dynamically" figure out the relations. Once the program is fixed, however, those big-joins can be replaced by a series of joins, since the successors of each node are known. This "flattening" of the big-joins essentially performs the same dependency tracking (and each join can be optimized independently) before computing the fixpoint.

Specifically for abstract machines, Glaze et al. [2013] describe a number of optimizations, including the store-delta technique in the store-widened semantics as well as abstract compilation to systematically derive a performant implementation of analyzers derived via the abstracting abstract machines approach. For further discussion of performance, widening, and precision of flow analyses see Midtgaard [2012].

9 Conclusion

The focus of this work is optimizing incremental approaches to computing higher-order flow analyses. To do this, we defined a small language for monotone computations that is sufficient for describing optimizations for these analyses. Optimizations to the incremental forms of these computations occur via specializing the derivative on specific kinds of changes, which are used in conjunction with a fixpoint loop that is amenable to specialized derivatives and is easily tunable. In particular, we note that small changes are good targets as they provide the most opportunities for specialization. Specialization primarily takes the form of partial evaluation and rewrite rules before the fixpoint runs, as well as dynamic dependency tracking between computations and various changes, the combination of which yield an efficient implementation of the derivative.

Acknowledgements. We thank the anonymous reviewers for their helpful and insightful feedback, as well as John Reppy, Olin Shivers, Skye Soss, and Byron Zhong for their work on the 3CPS compiler and their feedback on this work. This work is supported by NSF Award #1846350.

Artifact Availability. Our analysis implementations and benchmarks are available via Zenodo [Quiring and Van Horn 2024]. The instructions for reproducing our experiments and for understanding our implementation is available within the contained `README.md`.

References

Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.*

Michael Arntzenius. 2017. Static differentiation of monotone fixed points. (2017). http://www.rntz.net/files/fixderiv.pdf

Michael Arntzenius and Neel Krishnaswami. 2019. Seminaïve Evaluation for a Higher-Order Functional Language. *Proc. ACM Program. Lang.* 4, POPL, Article 22 (dec 2019), 28 pages. https://doi.org/10.1145/3371090

Francois Bancilhon. 1986. *Naive Evaluation of Recursively Defined Relations.* Springer-Verlag, Berlin, Heidelberg, 165–178.

Dominique Boucher and Marc Feeley. 1996. Abstract Compilation: A New Implementation Paradigm for Static Analysis. In *Proceedings of the 6th International Conference on Compiler Construction (CC '96).* Springer-Verlag, Berlin, Heidelberg, 192–207.

Hyunjun Eo. 2004. A Differential Fixpoint Iteration Method for Static Analysis Specifications. https://api.semanticscholar.org/CorpusID:16795054

Hyunjun Eo and Kwangkeun Yi. 2002. An Improved Differential Fixpoint Iteration Method for Program Analysis. 285–301.

Christian Fecht and Helmut Seidl. 1998. Propagating Differences: An Efficient New Fixpoint Algorithm for Distributive Constraint Systems. *Nordic Journal of Computing* 5, 90–104.

Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. 2016. Pushdown Control-Flow Analysis for Free. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16).* Association for Computing Machinery, New York, NY, USA, 691–704. https://doi.org/10.1145/2837614.2837631

Dionna Glaze, Nicholas Labich, Matthew Might, and David Van Horn. 2013. Optimizing Abstract Abstract Machines. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13).* Association for Computing Machinery, New York, NY, USA, 443–454. https://doi.org/10.1145/2500365.2500604

Susan Horwitz, Thomas Reps, and Mooly Sagiv. 1995. Demand Interprocedural Dataflow Analysis. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering* (Washington, D.C., USA) *(SIGSOFT '95).* Association for Computing Machinery, New York, NY, USA, 104–115. https://doi.org/10.1145/222124.222146

Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. From Datalog to Flix: A Declarative Language for Fixed Points on Lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16).* Association for Computing Machinery, New York, NY, USA, 194–208. https://doi.org/10.1145/2908080.2908096

Jan Midtgaard. 2012. Control-Flow Analysis of Functional Programs. *ACM Comput. Surv.* 44, 3, Article 10 (jun 2012), 33 pages. https://doi.org/10.1145/2187671.2187672

Anders Møller and Michael I. Schwartzbach. 2023. Static Program Analysis. https://cs.au.dk/~amoeller/spa/spa.pdf

Benjamin Quiring, John Reppy, and Olin Shivers. 2022. Analyzing Binding Extent in 3CPS. *Proc. ACM Program. Lang.* 6, ICFP, Article 114 (aug 2022), 29 pages. https://doi.org/10.1145/3547645

Benjamin Quiring and David Van Horn. 2024. Experiments for "Deriving with Derivatives: Optimizing Incremental Fixpoints for Higher-Order Flow Analysis". https://doi.org/10.5281/zenodo.11906121

Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95).* Association for Computing Machinery, New York, NY, USA, 49–61. https://doi.org/10.1145/199448.199462

Olin Shivers. 1991. *Control-Flow Analysis of Higher-Order Languages, or Taming Lambda.* Ph. D. Dissertation. School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania. Technical Report CMU-CS-91-145.

Benno Stein, Bor-Yuh Evan Chang, and Manu Sridharan. 2021. Demanded Abstract Interpretation. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021).* Association for Computing Machinery, New York, NY, USA, 282–295. https://doi.org/10.1145/3453483.3454044

Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: A DSL for the Definition of Incremental Program Analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE '16).* Association for Computing Machinery, New York, NY, USA, 320–331. https://doi.org/10.1145/2970276.2970298

David Van Horn and Matthew Might. 2010. Abstracting Abstract Machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '10).* Association for Computing Machinery, New York, NY, USA, 51–62. https://doi.org/10.1145/1863543.1863553

Received 2024-02-28; accepted 2024-06-18