Relative Store Fragments for Singleton Abstraction

Leandro Facchinetti[1]*, Zachary Palmer[2], and Scott F. Smith[1]

[1] {leandro,scott}@jhu.edu
Department of Computer Science
The Johns Hopkins University
[2] zachary.palmer@swarthmore.edu
Department of Computer Science
Swarthmore College

Abstract. A *singleton abstraction* occurs in a program analysis when some results of the analysis are known to be exact: an abstract binding corresponds to a single concrete binding. In this paper, we develop a novel approach to constructing singleton abstractions via *relative store fragments*. Each store fragment is a *locally* exact store abstraction in that it contains only those abstract variable bindings necessary to address a particular question at a particular program point; it is *relative* to that program point and the point of view may be shifted. We show how an analysis incorporating relative store fragments achieves flow-, context-, path- and must-alias sensitivity, and can be used as a basis for environment analysis, without any machinery put in place for those specific aims. We build upon recent advances in *demand-driven* higher-order program analysis to achieve this construction as it is fundamentally tied to demand-driven lookup of variable values.

1 Introduction

A *singleton abstraction* [Mig10b], also known as a *must analysis* [Mid12], is a common thread found across advanced program analyses: some results of the analysis are known to be exact, meaning the abstraction set is in fact a singleton in those cases. Singleton abstractions have traditionally been used in must-alias analyses for first-order programs, e.g. [CWZ90]. For higher-order programs they have been used in lightweight closure conversion [SW97] in must-alias analysis [JTWW98], and abstract garbage collection has been used to produce singleton abstractions [MS06b].

This paper develops a novel approach to constructing singleton abstractions using *relative store fragments*. Unlike stores in traditional abstract interpretation, these are *fragments* in that they contain information about a *subset* of the store necessary to address a particular analysis question; there is no global store. Instead of storing binding information in terms of global calling context, these fragments express binding information *relative* to a particular point in the program using call path difference information similar to ΔCFA frame strings [MS06a,GM17].

* Facchinetti is supported by a CAPES Fellowship, process number 13477/13-7.

In this paper we define *Demand-driven Relative Store Fragment analysis*, abbreviated DRSF. DRSF incorporates relative store fragments to achieve flow-, context-, path- and must-alias sensitivity, and can be used as a basis for environment analysis [Shi91,BFL$^+$14,GM17], without any extra machinery put in place for those specific aims. We define for this analysis an algebra of relative store fragment composition and relativization to combine these fragments in the process of answering control- and data-flow questions. We now briefly describe some dimensions of DRSF's expressiveness.

Path Sensitivity: Path-sensitivity is well-known as an important dimension of expressiveness in program analyses [BA98,DLS02,XCE03,DMH15,THF10]. Path-sensitive analyses observe control-flow decisions and use this information to refine their results. DRSF's store fragments are naturally conducive to path sensitivity: the values of variables used to dictate control flow are recorded in store fragments and used to discard impossible control flow combinations.

Must-alias Analysis: The original purpose of singleton analyses was to extract must-alias properties [CWZ90,JTWW98]. The case where a variable must (not just may) alias another means an assignment to one must (not just may) also affect the other. We can additionally use our singleton store abstraction to achieve must-alias properties over a mutable heap.

Context Sensitivity: Another pleasant aspect of the theory is that context-sensitivity [Shi91] also comes "for free": the call path annotations on variables additionally disambiguate contexts. Creating a multiplicity of store fragments is expensive, but the multiple purposes they may be put to allows the effort to be amortized. Along with standard k-CFA style context sensitivity [Shi91], CPA-style context-sensitivity [Age95,Bes09,VS10] also naturally emerges.

Environment Analysis: While we are primarily focused on fundamentals and not on potential clients of the analysis, the environment problem is a classic stress test for higher-order program analyses [Shi91,BFL$^+$14,GM17] and DRSF can be used a basis to address that problem.

The Context: Demand-Driven Higher-Order Analysis We work in the context of DDPA, a demand-driven higher-order analysis [PS16] which applies ideas of first-order demand-driven analyses [Rep94,DGS97,SR05] to higher-order programs. DDPA incorporates filters on data lookup for some path-sensitivity but lacks "alignment": it considers each variable separately.

Results This paper is a proof-of-concept study of DRSF: we give a complete definition of the analysis for a simple functional language, prove some basic properties, and describe an implementation with additional features including a mutable heap. While we establish that the algorithm has an exponential worst-case, in practice on benchmarks it has reasonable performance.

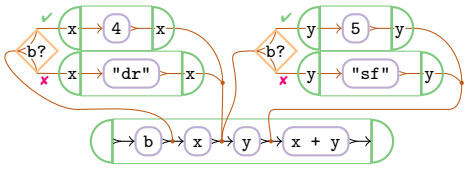
2 Overview

In this section we will present a series of examples that illustrate the concepts behind DRSF. The analysis builds on DDPA [PS16], and basic features of that analysis will be introduced as we go.

2.1 A simple example illustrating path-sensitivity

We start with a very simple program in an ML-like language:

```
1 let b = coin_flip () in
2 let x =
3   if b then 4 else "dr" in
4 let y =
5   if b then 5 else "sf" in
6 x + y
```



The `coin_flip` primitive non-deterministically returns either `true` or `false`. Operator `+` is overloaded to append strings; it is not defined when the operand types mismatch – i.e., there is no implicit coercion *à la* JavaScript. This program never has a type mismatch: the types of `x` and `y` are aligned by `b`. To observe that addition is safe in this program, an analysis must observe this alignment. Using $\hat{\circ}$ for the abstract entity corresponding to a concrete $\circ$, the analysis must show $\hat{\texttt{x}} = \{\widehat{\texttt{4}}\}$ and $\hat{\texttt{y}} = \{\widehat{\texttt{5}}\}$ occur together (and similarly for the strings), and, more importantly, prove that $\hat{\texttt{x}} = \{\widehat{\texttt{4}}\}$ and $\hat{\texttt{y}} = \{\widehat{\texttt{"sf"}}\}$ do not co-occur. DRSF is *path-sensitive*: it preserves this connection. We now outline how this is accomplished.

Both DDPA and DRSF construct a Control-Flow Graph (CFG) of the program; the CFG constructed by DRSF here appears to the right of the code. Given a (perhaps partial) CFG, the analyses answer questions of the type "what are the possible abstract values that reach *this* variable at *this* program point?" The analyses perform a lookup by traversing the CFG backward, in the direction opposite to control-flow, until they find a definition, in the tradition of demand-driven program analyses [Rep94,DGS97,SR05,HT01,PS16]. For example, if asked "what are the possible abstract values for `b` at the end of the program?" the analyses would initiate a lookup at the CFG node representing the end of the program and traverse the graph backward with respect to control-flow to reach the clause `b = coin_flip ()`, producing the result $\hat{b} = \{\widehat{\texttt{true}}, \widehat{\texttt{false}}\}$.

There are two complications in building demand-driven analyses for higher-order languages: the first is preventing unbounded search along cyclic paths in a CFG; the second is building the CFG itself, because higher-order functions determine control-flow, so data-flow and control-flow are intertwined. To address the former, DDPA and DRSF encode the CFG traversal in terms of a *PDA reachability* problem, building on ideas in [JSE+14,EMH10,BEM97]. For CFG construction, the analyses can perform lookups on partial CFGs and let the results inform the incremental construction of the full CFG.

The CFG above illustrates this process on our running example. The black and brown edges represent control flow, with black edges determined directly from program syntax, and brown edges introduced by the analysis. Initially, only the bottom-most spine exists, representing the statements at the top level. The analysis first wires in the upper-left widget for the conditional returning `x`: a lookup of `b` shows it can be `true` and so the `4` node is wired in. Since the lookup of `b` can also be `false` the `"dr"` node is also wired in. Similarly the branches of the `y` conditional are wired in one by one. In general, both DDPA and DRSF construct the CFG in a forward manner but use reverse lookup through the (partial) CFG

to find potential values which can inform CFG construction. By induction, all the information necessary to resolve control-flow at a program point is already available in the partial CFG when the program point is considered.

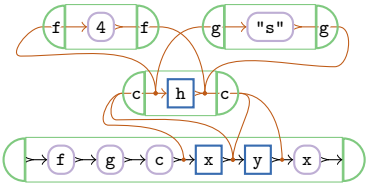
With the complete CFG it is possible to look up the potential values of final expression `x+y`. This, in turn, requires the lookups of variables `x` and `y`. The key difference between DDPA and DRSF is in how the analyses answer to these lookups. DDPA produces only sets of abstract values: $\hat{x}$'s result is $\{\widehat{4}, \widehat{\texttt{"dr"}}\}$ and $\hat{y}$'s result is $\{\widehat{5}, \widehat{\texttt{"sf"}}\}$; any path-sensitivity has been lost by the form of the output. DRSF, in addition to abstract values, produces *abstract store sets* associated with them: $\hat{x}$'s store set is $\{\langle\widehat{4}, \{\hat{b} \mapsto \widehat{\texttt{true}}, \hat{x} \mapsto \widehat{4}\}\rangle, \langle\widehat{\texttt{"dr"}}, \{\hat{b} \mapsto \widehat{\texttt{false}}, \hat{x} \mapsto \widehat{\texttt{"dr"}}\}\rangle\}$ and $\hat{y}$'s store set is $\{\langle\widehat{5}, \{\hat{b} \mapsto \widehat{\texttt{true}}, \hat{y} \mapsto \widehat{5}\}\rangle, \langle\widehat{\texttt{"sf"}}, \{\hat{b} \mapsto \widehat{\texttt{false}}, \hat{y} \mapsto \widehat{\texttt{"sf"}}\}\rangle\}$. Observe that each individual store in the store set contain singletons representing a potential slice of the runtime; it is a pair of a value and the mappings for each variable that led to that value. The key advantage of DRSF is that, to compute the addition, a *store merge* is performed on the store sets for $\hat{x}$ and $\hat{y}$. This store merge operation eliminates impossible stores; for example, it discards the combination of $\hat{x} \mapsto \widehat{4}$ and $\hat{y} \mapsto \widehat{\texttt{"sf"}}$ *because the corresponding abstract stores disagree on their values for* $\hat{b}$. The set of stores can be non-deterministic, but each store contains singletons which agree on all mappings. This can be used to solve a whole class of issues concerning correlation between bindings, such as *fake rebinding* [VS10].

One important aspect of abstract stores is that they are not absolute for the whole program, but *relative* to the program point in which the lookup initiated. They are also *fragments*: only the variables incident on the current lookup need be included. This latter property will be made clear in the subsequent examples.

2.2 A simple example of context-sensitivity

The following example illustrates a use of higher-order functions:

```
1 let f = fun _ -> 4 in
2 let g = fun _ -> "s" in
3 let c = fun h -> h () in
4 let x = c f in
5 let y = c g in
6 x
```



To look up `x` from the end of the program assuming the full CFG has been previously constructed, DRSF traverses the CFG in reverse from the program end. Walking backward, call site `x` in line 4 is the source of the result; the analysis must then proceed in reverse into `c`'s body, whereupon it finds a call to `h`. At this point it must find which functions `h` could be; `h` is a parameter to `c`, so DRSF has to exit its body to look for the arguments that could have been passed in.

Now, the analysis faces a dilemma: there are two wirings to `c`'s body in the CFG. One is at `x`, where `h` is bound to `f`; the other is at `y`, where `h` is bound to `g`. The former wiring matches the execution we are tracing but, in a naive walk back through the CFG, this information has been lost.

DRSF achieves context-sensitivity via *traces* attached to variables to indicate the relative call stack context the variable was found in. When `f` is looked up from the perspective of the top-level program, the trace is empty: []. The result of this lookup is $\hat{\texttt{f}} = \{\langle\widehat{\texttt{fun}_4}, \{\hat{\texttt{f}}@[] \mapsto \widehat{\texttt{fun}_4}\}\rangle\}$: trace [] indicates that the definition of $\hat{\texttt{f}}$ was found in the same calling context as where the question was asked.

During our lookup of `x`, however, we must find a store fragment for $\hat{\texttt{h}}$, the parameter of the $\hat{\texttt{c}}$ function, from *inside* of that function's body. $\hat{\texttt{h}}$ is defined in terms of $\hat{\texttt{f}}$, but that definition occurs in the same calling context as $\hat{\texttt{c}}$'s call site; that is, we must go to where $\hat{\texttt{c}}$ is called to find the value of $\hat{\texttt{f}}$. One caller of $\hat{\texttt{c}}$ is site $\hat{\texttt{x}}$ and we use the trace $[\triangleleft\hat{\texttt{x}}]$ to describe that $\hat{\texttt{x}}$ is the caller of the program point where our lookup started. Thus, one store we can obtain in looking up $\hat{\texttt{h}}$ from within `c` is $\langle\widehat{\texttt{fun}_4}, \{\hat{\texttt{f}}@[\triangleleft\hat{\texttt{x}}] \mapsto \widehat{\texttt{fun}_4}, \hat{\texttt{h}}@[] \mapsto \widehat{\texttt{fun}_4}\}\rangle$. Similarly, if we consider that $\hat{\texttt{c}}$ can be called from site $\hat{\texttt{y}}$, we obtain another potential store: $\langle\widehat{\texttt{fun}_s}, \{\hat{\texttt{g}}@[\triangleleft\hat{\texttt{y}}] \mapsto \widehat{\texttt{fun}_s}, \hat{\texttt{h}}@[] \mapsto \widehat{\texttt{fun}_s}\}\rangle$.

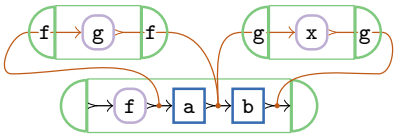
Now recall that, in the path we were originally taking to search for `x`, we had entered `h` via call site `x`. So, to relativize the above stores to the top level of the program, we must append a $\triangleright\hat{\texttt{x}}$ to the trace on all store variables. This gives stores containing $\{\hat{\texttt{f}}@[\triangleleft\hat{\texttt{x}}, \triangleright\hat{\texttt{x}}] \mapsto \widehat{\texttt{fun}_4}, \hat{\texttt{h}}@[\triangleright\hat{\texttt{x}}] \mapsto \widehat{\texttt{fun}_4}\}$ and $\{\hat{\texttt{g}}@[\triangleleft\hat{\texttt{y}}, \triangleright\hat{\texttt{x}}] \mapsto \widehat{\texttt{fun}_s}, \hat{\texttt{h}}@[\triangleright\hat{\texttt{x}}] \mapsto \widehat{\texttt{fun}_s}\}$. Trace $[\triangleleft\hat{\texttt{x}}, \triangleright\hat{\texttt{x}}]$ is a no-op which cancels out to []. On the other hand, $[\triangleleft\hat{\texttt{y}}, \triangleright\hat{\texttt{x}}]$ is a *contradiction* since call and return do not align and it means this latter store can be *eliminated* and only the former is sound. So, the lookup of $\hat{\texttt{x}}$ yields only the result of $\hat{\texttt{x}}$; this demonstrates context-sensitivity.

Our call string logic is related to ΔCFA's delta frame strings [MS06a,GM17]. Note that trace-based context-sensitivity also influences path-sensitivity: DRSF is path-sensitive for sources of non-determinism not related to loss of context sensitivity: user input, `coin_flip`, integer comparisons in which precision was lost, and so on. We will see later that, just as in kCFA, the loss of context-sensitivity takes a toll path-sensitivity as well.

2.3 Precise non-local variable lookup

One of the key features of DDPA inherited by DRSF is precision in non-local variable lookup. Consider the following program.

```
1 let f = fun x ->
2   let g = fun y -> x in g
3 in
4 let a = f 0 in
5 let b = a 5 in
6 b
```



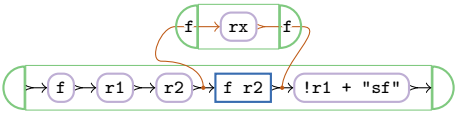
To look up `b` from the end of the program, DRSF first finds `b = a 5` so the function in `a` must be entered in reverse to find the value. `a` itself is a closure over the function `g`, so it then has to look up `g`'s return value, `x`. But if the analysis continues traversing the CFG backward with respect to control-flow looking for `x`, it comes back to the call site `b` and proceeds to skip over `a` and `f`, failing to find `x` at the start of the program. DRSF solves this problem with a *statically-scoped* search for non-locals in the style of *access links* in compiler implementations.

So, at the point where we were looking up `x` above, DRSF first must look up the *definition of the function that was applied*, `g`, and resume search for `x` from there as that is where `x`'s definition lexically occurs. Searching for `g` from within call site `b`, it is the return value from call site `a` which continues the search into `f`'s body. At that point, it finds `g`'s definition, and can resume lookup for `x` from that CFG location, where it finds that `x` is the function parameter. So DRSF returns to the call site `a` and finds the argument `0`, which leads to result $\hat{b} = \{\widehat{0}\}$.

2.4 Must-alias analysis

Each relative store contains singletons and this property naturally leads to must-alias analysis expressiveness. Consider the following program.

```
1 let f = fun rx -> rx := "dr" in
2 let r1 = ref 4 in
3 let r2 = r1 in
4 f r2;
5 !r1 + "sf"
```



In looking up `r1`, we encounter a call to `f`. Since this is a stateful lookup unlike earlier lookups above, we must examine the `f` call for side effects; we discover that it sets its parameter `rx`. If we knew that `rx` *must* be an alias for `r1` we would have found the most recent assignment to `rx`, so this condition is checked. Searching for `rx` shows it *must* be `r2`, which in turn *must* be `r1`. So, there is only one viable store here and in it `"dr"` is the most recent assignment to `r1`. Therefore `!r1` always returns a string.

These small examples illustrate DRSF's expressiveness. Fortunately, the principles of the analysis are general: precision is retained on less trivial examples we run through the implementation in Section 4.1. The next step is to formalize the analysis to make its meaning precise.

3 The Analysis

We formally analyze the grammar of the language in Figure 1. All expressions in this language are in A-normal form [FSDF93] to better align the expressions and the analysis; we also require that every variable declaration is unique. This grammar is a small subset of the implemented language; we leave out deep pattern matching, state, and atomic data and operators. The operational semantics for this simplistic call-by-value language is straightforward and is not given here for space reasons. For this discussion, it bears mentioning that conditional clause bodies are functions; when the condition is matched (or not matched), we call the appropriate function with the matched argument.

$e ::= [c, \ldots]$	*expressions*	$r ::= \{\ell = x, \ldots\}$	*records*
$c ::= x = b$	*clauses*	$f ::= \texttt{fun}\ x\ \texttt{->}\ (\ e\ )$	*functions*
$b ::= v \mid x \mid x\ x \mid x \sim p\ \texttt{?}\ f : f \mid x.\ell$	*clause bodies*	x	*variables*
$v ::= r \mid f$	*values*	ℓ	*labels*
$p ::= \{\ell, \ldots\} \mid \texttt{fun} \mid \texttt{any}$	*patterns*		

Fig. 1. Expression Grammar

As in the previous section, we express the abstraction of a construct $\circ$ as $\hat{\circ}$. In this simple language, these constructs are in fact identical; we use two distinct forms for readability only.

The DRSF analysis incrementally constructs a control flow graph, a process which relies heavily upon a variable-value lookup function. Given a variable and a point in the program, this lookup function generally produces relative store fragments which indicate possible (abstract) values for that variable as well as store mappings for any variables which influenced this decision. We begin our definition of the analysis by defining these relative store fragments and then specify the lookup function and single-step CFG construction operation in turn.

3.1 Relative Store Fragments

The grammar for the analysis store constructs is given in Figure 2. We further restrict $\hat{\Psi}$ such that there cannot be two mappings $\hat{\psi}/\hat{\psi}'$ with the same $\hat{\rho}$, provided $\hat{\rho} = \hat{x}@\hat{\Delta}$ and $\hat{\Delta}$ is a full trace. And, we impose an invariant that for any rooted store $\langle \hat{\psi}, \hat{\Psi} \rangle$, $\hat{\psi} \in \hat{\Psi}$.

$$
\begin{array}{llll}
\hat{\rho} ::= \hat{x}@\hat{\Delta} & \textit{relative trace variable} & \hat{\sigma} ::= \langle \hat{\psi}, \hat{\Psi} \rangle & \textit{rooted store} \\
\hat{\Delta} ::= [\hat{\delta}, \ldots] \mid (\hat{\delta}, \ldots] & \textit{full and partial relative traces} & \hat{\Psi} ::= \{\hat{\psi}, \ldots\} & \textit{raw store} \\
\hat{\delta} ::= \lhd \hat{c} \mid \rhd \hat{c} & \textit{relative trace part} & \hat{\psi} ::= \hat{\rho} \mapsto \hat{v} & \textit{store mapping}
\end{array}
$$

Fig. 2. Abstract Store Grammar

A lookup of a variable in the DRSF analysis does not just produce a set of abstract values; it more generally returns a set of *store fragments* $\hat{\sigma}$ paired with locations where they are found (discussed in Section 3.2). Each store fragment $\hat{\sigma}$ is a *rooted store* because it includes a *root mapping* $\hat{\rho} \mapsto \hat{v}$ indicating the abstract value $\hat{v}$ of the looked-up variable (in the Overview we elided the "$\hat{\rho} \mapsto$" on the root mapping for simplicity), as well as a *raw store* $\hat{\Psi}$ of mappings for all other relevant variables in context. Mappings $\hat{\psi}$ do not map raw variables $\hat{x}$ to values: they map *trace-relative* variables $\hat{\rho} = \hat{x}@\hat{\Delta}$ to values, with the traces $\hat{\Delta}$ defining the context where the variable was defined relative to the current context.

Relative traces come in two forms. The first form is a *full trace* $[\ldots]$ which is precise: it describes exactly from where the result of lookup is obtained. The second form $(\ldots]$ is a *partial trace* which represents a lossy suffix of the former, trimmed to allow the analysis to terminate. We formally define a trimming operation on relative traces here which retains only the rightmost k elements:

Definition 1. *We define* $(\hat{\Delta})\lceil^k$ *such that*

$$
[\hat{\delta}_n, \ldots, \hat{\delta}_1]\lceil^k = \begin{cases} [\hat{\delta}_n, \ldots, \hat{\delta}_1] & \textit{when } n \leq k \\ (\hat{\delta}_k, \ldots, \hat{\delta}_1] & \textit{otherwise} \end{cases} \qquad (\hat{\delta}_n, \ldots, \hat{\delta}_1]\lceil^k = (\hat{\delta}_{\min(k,n)}, \ldots, \hat{\delta}_1]
$$

We are using standard notation $[a_1, \ldots, a_n]$ for regular lists and $\hat{\Delta} \,||\, \hat{\Delta}'$ for list append. Relative traces are list-like, so it is convenient to use similar notation to our lists when considering list suffixes; for instance, we may write $\hat{\Delta} = \hat{\Delta}' \,||[\hat{\delta}]$ to indicate that $\hat{\delta}$ is the last item of the trace $\hat{\Delta}$. Because traces may be partial, however, we only use this notation when the trace is the left operand and the right operand is a simple list of trace parts.

We also define here two convenience routines for creating rooted stores and extracting the underlying value from them:

Definition 2. *1. We define $\langle\!|\hat{x}, \hat{v}|\!\rangle$, the "store creation" operation, as $\langle \hat{x}@[] \mapsto \hat{v}, \{\hat{x}@[] \mapsto \hat{v}\}\rangle$.*
2. We define $(\!|\hat{\sigma}|\!)$, the "store read" operation, as $(\!|\langle \hat{\rho} \mapsto \hat{v}, \hat{\Psi}\rangle|\!) = \hat{v}$.

Trace Suffixing. Raw stores $\hat{\Psi}$ map *relative trace variables* to values. As indicated in Figure 2, each relative trace variable is a pairing between a variable and a *relative trace* which describes the location of that variable relative to the current program stack. If a value was originally constructed outside of the current function, for instance, the trace $[\triangleleft\hat{c}]$ indicates that the call site $\hat{c}$ was pushed onto the stack since the value was bound. If lookup found a value originally bound inside of a function that has since returned, the trace may contain $[\triangleright\hat{c}]$.

As every mapping in a store is relative to the current position of variable lookup, we must uniformly modify these traces as lookup moves between function calls. We begin by defining an operation which suffixes an existing relative trace with a new stack operation. This may cancel redundant values: $[\triangleleft\hat{c}, \triangleright\hat{c}]$ enters and exits a function from the same call site and so is a no-op. Trace suffixing may also fail on impossible stack transformations: $[\triangleleft\hat{c}, \triangleright\hat{c}']$ for $\hat{c} \neq \hat{c}'$ returns to a point not the calling point, an impossibility.

We now define a trace suffixing operation to formalize the above. Traces longer than a given k are trimmed to partial traces of the form $(\ldots]$ by truncating the front of the list. This prevents traces from growing arbitrarily long and allows stores to keep finite context, as described above. We represent suffix failure in impossible cases by leaving the operation undefined.

Definition 3. *For each $k \geq 0$, trace suffixing $\ltimes_k$ is defined as follows:*

$$
\begin{array}{rlrl}
[] \ltimes_k \hat{\delta} & = [\hat{\delta}]\lceil^k & () \ltimes_k \hat{\delta} & = (\hat{\delta}]\lceil^k \\
\hat{\Delta}\,||[\triangleleft\hat{c}] \ltimes_k \triangleright\hat{c} & = \hat{\Delta} & \hat{\Delta}\,||[\triangleleft\hat{c}_1] \ltimes_k \triangleright\hat{c}_2 & \textit{is undefined if } \hat{c}_1 \neq \hat{c}_2 \\
\hat{\Delta}\,||[\triangleleft\hat{c}_1] \ltimes_k \triangleleft\hat{c}_2 & = (\hat{\Delta}\,||[\triangleleft\hat{c}_1, \triangleleft\hat{c}_2])\lceil^k & \hat{\Delta}\,||[\triangleright\hat{c}] \ltimes_k \hat{\delta} & = (\hat{\Delta}\,||[\triangleright\hat{c}, \hat{\delta}])\lceil^k
\end{array}
$$

We extend the trace suffixing operator to operate on pairs of traces. We allow $n = 0$ and/or $m = 0$ below.

$$\hat{\Delta} \ltimes_k [\hat{\delta}_1, \ldots, \hat{\delta}_n] = \hat{\Delta} \ltimes_k \hat{\delta}_1 \ltimes_k \ldots \ltimes_k \hat{\delta}_n$$
$$\hat{\Delta} \ltimes_k (\hat{\delta}_1, \ldots, \hat{\delta}_n] = (\hat{\delta}_1, \ldots, \hat{\delta}_n]\lceil^k$$

Observe that traces are *pop/push-bitonic* [Mig07] in that they have the form $[\triangleright\hat{c}_1, \ldots \triangleright\hat{c}_n, \triangleleft\hat{c}_1, \ldots \triangleleft\hat{c}_m]$ for $n, m \geq 0$.

Trace suffixing homomorphically extends to stores and other entities.

Definition 4. *We extend trace suffixing to variables, mappings, raw stores, and rooted stores as follows:*

$$
\begin{array}{rlrl}
(\hat{x}@\hat{\Delta}) \ltimes_k \hat{\delta} & = \hat{x}@(\hat{\Delta} \ltimes_k \hat{\delta}) & (\hat{x}@\hat{\Delta}) \ltimes_k \hat{\Delta}' & = \hat{x}@(\hat{\Delta} \ltimes_k \hat{\Delta}') \\
(\hat{\rho} \mapsto \hat{v}) \ltimes_k \hat{\delta} & = (\hat{\rho} \ltimes_k \hat{\delta}) \mapsto \hat{v} & (\hat{\rho} \mapsto \hat{v}) \ltimes_k \hat{\Delta} & = (\hat{\rho} \ltimes_k \hat{\Delta}) \mapsto \hat{v} \\
\hat{\Psi} \ltimes_k \hat{\delta} & = \left\{ \hat{\psi} \ltimes_k \hat{\delta} \middle| \hat{\psi} \in \hat{\Psi} \right\} & \hat{\Psi} \ltimes_k \hat{\Delta} & = \left\{ \hat{\psi} \ltimes_k \hat{\Delta} \middle| \hat{\psi} \in \hat{\Psi} \right\} \\
\langle \hat{\psi}, \hat{\Psi} \rangle \ltimes_k \hat{\delta} & = \langle \hat{\psi} \ltimes_k \hat{\delta}, \hat{\Psi} \ltimes_k \hat{\delta} \rangle & \langle \hat{\psi}, \hat{\Psi} \rangle \ltimes_k \hat{\Delta} & = \langle \hat{\psi} \ltimes_k \hat{\Delta}, \hat{\Psi} \ltimes_k \hat{\Delta} \rangle
\end{array}
$$

Note that in each case above if any of the $\ltimes_k$ on the right are undefined, the suffixing operation on the left is taken to be undefined.

To illustrate how the "undefinedness" of trace suffixing is fully propagated, $\hat{x}@[\triangleleft\hat{c}_1] \ltimes_k \triangleright\hat{c}_2$ and $\{\hat{x}@[\triangleleft\hat{c}_1] \mapsto \hat{v}, \ldots\} \ltimes_k \triangleright\hat{c}_2$ are both undefined if $\hat{c}_1 \neq \hat{c}_2$. Hereafter we will take k to be fixed and abbreviate $\ltimes_k$ as $\ltimes$.

Store Merge. Some lookup operations in DRSF require a *subordinate lookup*. For example, before proceeding from a call site into a function body, we must perform a lookup to make sure that function is called at this site. A key feature of DRSF is how stores from lookups can be *merged*: formally, $\hat{\Psi}_1 \oplus \hat{\Psi}_2$. Any store that represents an inconsistent run-time state can be eliminated in $\oplus$. There are two conditions which cause store merge to fail. First, a *variable* could be inconsistent: one store maps `x` to an integer while the other maps `x` to a record. Second, the *call stack* implied by the store could be inconsistent, for example one store contains $\mathtt{r}@[\triangleright\mathtt{x}, \triangleleft\mathtt{y}]$ and the other contains $\mathtt{r}@[\triangleright\mathtt{x}, \triangleleft\mathtt{z}]$: these constraints cannot simultaneously be satisfied and merge fails.

We now define store merge. The first condition – merging inconsistent variables – is implicitly addressed by the manner in which we propagate the undefined cases of the definitions above. To handle the second condition, we must define an auxiliary trace relation $\hat{\Delta}_1 \doteq \hat{\Delta}_2$ and apply it to stores.

Definition 5. *1. We define the* trace consistency relation $\hat{\Delta}_1 \doteq \hat{\Delta}_2$ *to hold iff there exist some $\hat{\Delta}'_0$, $\hat{\Delta}'_1$, and $\hat{\Delta}'_2$ such that the following hold:*
- *No terms of the form $\triangleleft\hat{c}$ appear in $\hat{\Delta}'_0$, $\hat{\Delta}'_1$, or $\hat{\Delta}'_2$;*
- *$\hat{\Delta}_1 \ltimes \hat{\Delta}'_0 = \hat{\Delta}'_1$, and $\hat{\Delta}_2 \ltimes \hat{\Delta}'_0 = \hat{\Delta}'_2$.*

2. Two stores $\hat{\Psi}_1$ and $\hat{\Psi}_2$ are trace consistent *(written $\hat{\Psi}_1 \doteq \hat{\Psi}_2$) iff $\forall(\hat{x}_1@\hat{\Delta}_1) \mapsto \hat{v}_1 \in \hat{\Psi}_1, (\hat{x}_2@\hat{\Delta}_2) \mapsto \hat{v}_2 \in \hat{\Psi}_2$. $\hat{\Delta}_1 \doteq \hat{\Delta}_2$.*

3. The merge $\hat{\Psi}_1 \oplus \hat{\Psi}_2$ *of two unrooted stores $\hat{\Psi}_1$ and $\hat{\Psi}_2$ is the union of their mappings, $\hat{\Psi}_1 \cup \hat{\Psi}_2$, provided the following conditions hold:*
- *For all $\hat{\rho} \mapsto \hat{v}_1 \in \hat{\Psi}_1$ and $\hat{\rho} \mapsto \hat{v}_2 \in \hat{\Psi}_2$, if $\hat{\rho}$ of form $\hat{x}@[\ldots]$ then $\hat{v}_1 = \hat{v}_2$*
- *$\hat{\Psi}_1 \doteq \hat{\Psi}_2$*

If these conditions do not hold, then $\hat{\Psi}_1 \oplus \hat{\Psi}_2$ is undefined.

Two forms of merge for rooted stores are used in DRSF and we make explicit definitions for readability. *Parallel store merge* $\hat{\sigma}_1 \leftleftarrows \hat{\sigma}_2$ merges two stores computed from the same reference point and the root of the merged store is set to the root of $\hat{\sigma}_1$. *Serial store merge* $\hat{\sigma}_1 \leftarrow\!\!\!\rho\, \hat{\sigma}_2$ merges stores whilst offsetting the reference point of the first store by the reference point of the second. This adjustment is used to support the non-local lookups described in Section 2.3.

Definition 6. *1. The* parallel store merge *operation $\leftleftarrows$ is defined as follows:*
$\langle\hat{\psi}_1, \hat{\Psi}_1\rangle \leftleftarrows \langle\hat{\psi}_2, \hat{\Psi}_2\rangle = \langle\hat{\psi}_1, \hat{\Psi}_1 \oplus \hat{\Psi}_2\rangle$

2. The serial store merge *operation, $\leftarrow\!\!\!\rho$, is defined as follows:*
$\hat{\sigma} \leftarrow\!\!\!\rho \langle\hat{x}@\hat{\Delta} \mapsto \hat{v}, \hat{\Psi}\rangle = (\hat{\sigma} \ltimes \hat{\Delta}) \leftleftarrows \langle\hat{x}@\hat{\Delta} \mapsto \hat{v}, \hat{\Psi}\rangle$

These operations are undefined when any of the component operations are undefined.

Singleton Abstractions. At this point we have all the tools necessary to construct and utilize singleton abstractions in the analysis. Each mapping in a relative store fragment includes a relative trace $\hat{\Delta}$ which is either partial or full. Full traces describe in the analysis a set of memory locations at run-time that share a property, for example, closures over the same lambda or records with the same fields corresponding to the same variables. The conditions for unrooted store merge $\oplus$ (and, by extension, the store merge operations $\Subset$ and $\hookleftarrow$) are then used to identify and discard cases in which stores have immediately dissonant values for full-trace bindings.

Note that immediate dissonance is only evident for values in which the concrete-to-abstract mapping is one-to-one, as is the case for our functions and records. When extending DRSF to other kinds of values (for example, numbers and strings) the analysis must conservatively assume dissonance whenever the abstract values fall out of the range for which the concrete-to-abstract mapping is one-to-one. For those values within the one-to-one mapping, subsequent lookups of e.g. function non-locals or record fields can be used to check for dissonance deeply within a structure. By induction, when these lookups reach the leaves of these data structures, they prove that *a chain of full-trace bindings represent singleton abstractions.* This allows the analysis we define below to be a basis for expressing singleton abstractions, though using DRSF to develop e.g. an environment analysis is beyond the scope of this paper.

3.2 Lookup over Control Flow Graphs

Before defining variable lookup for the analysis, we must formally define the control flow graphs described in Section 2. The grammar appears in Figure 3. Recall that our simplified language is A-normalized and each variable declaration is unique. Each $\hat{a}$ is a program clause (a program point in Overview terminology) and each $\hat{a} \ll \hat{a}$ is a CFG edge (a brown or black arrow in the Overview CFGs) collected into set $\hat{G}$, the full CFG. Variable lookup produces a set of *positioned* stores $\hat{\Phi}$, which are relative store fragments together with a reference point for them. This reference point is used to support the statically scoped search described in Section 2.3.

$\hat{a}$	::=	$\hat{c} \mid \hat{x} \overset{\mathbb{Q}\hat{c}}{=} \hat{x} \mid \hat{x} \overset{\mathbb{D}\hat{c}}{=} \hat{x} \mid \textsc{Start} \mid \textsc{End}$	*abstract annotated clauses*
$\hat{g}$	::=	$\hat{a} \ll \hat{a}$	*control flow edges*
$\hat{G}$	::=	$\{\hat{g}, \ldots\}$	*control flow graphs*
$\hat{\phi}$	::=	$\langle \hat{a}, \hat{\sigma} \rangle$	*positioned store*
$\hat{\Phi}$	::=	$\{\hat{\phi}, \ldots\}$	*positioned store set*

Fig. 3. DRSF Analysis Grammar

For notational purposes, we overload each operation on stores $\hat{\sigma}$ to positioned stores $\hat{\phi}$; for instance, we let $\langle \hat{a}, \hat{\sigma} \rangle \ltimes \hat{\delta} = \langle \hat{a}, \hat{\sigma} \ltimes \hat{\delta} \rangle$ and $(\!|\langle \hat{a}, \hat{\sigma} \rangle|\!) = (\!|\hat{\sigma}|\!)$. We overload each such operation to sets of $\hat{\phi}$; for example, $\hat{\Phi} \ltimes \hat{\delta} = \{\hat{\phi} \ltimes \hat{\delta} \mid \hat{\phi} \in \hat{\Phi}\}$.

Let $\textsc{Match}(\hat{v}, \hat{p})$ be the natural shallow pattern match relation between a value and a pattern. Let $\mathrm{RV}(e)$ be the last variable defined in an expression: $\mathrm{RV}([\ldots, \hat{x} = \hat{b}]) = \hat{x}$.

We are finally in a position to define variable lookup $\hat{G}(\hat{a}, \hat{x})$, a function returning the set of positioned stores that give a value to $\hat{x}$ from the perspective of program point $\hat{a}$.

Definition 7. *For CFG $\hat{G}$, let $\hat{G}(\hat{a}_0, \hat{x})$ be the function returning the least set of positioned stores $\hat{\Phi}$ for some $\hat{a}_1 \overset{\hat{G}}{\ll} \hat{a}_0$ satisfying the following clauses:*

1. ***Variable search***
   (a) VALUE DISCOVERY
   If $\hat{a}_1 = (\hat{x} \texttt{=} \hat{v})$ *then* $\langle \hat{a}_1, \langle\!|\hat{x}, \hat{v}|\!\rangle \rangle \in \hat{\Phi}$.
   (b) VALUE ALIAS
   If $\hat{a}_1 = (\hat{x} \texttt{=} \hat{x}')$ *then* $\hat{G}(\hat{a}_1, \hat{x}') \subseteq \hat{\Phi}$.
   (c) CLAUSE SKIP
   If $\hat{a}_1 = (\hat{x}' \texttt{=} b)$ *and* $\hat{x}' \neq \hat{x}$, *then* $\hat{G}(\hat{a}_1, \hat{x}) \subseteq \hat{\Phi}$.
2. ***Function wiring***
   (a) FUNCTION ENTER: PARAMETER VARIABLE
   If $\hat{a}_1 = (\hat{x} \overset{\hat{c}\complement}{\texttt{=}} \hat{x}')$ *and* $\hat{c} = (\hat{x}_1'' \texttt{=} \hat{x}_2''\ \hat{x}')$, $\langle \hat{a}', \hat{\sigma} \rangle \in \hat{G}(\hat{a}_1, \hat{x}_2'')$, *then*
   $(\hat{G}(\hat{a}_1, \hat{x}') \leftleftarrows \hat{\sigma}) \ltimes \complement\hat{c} \subseteq \hat{\Phi}$.
   (b) FUNCTION EXIT: RETURN VARIABLE
   If $\hat{a}_1 = (\hat{x} \overset{\mathbb{D}\hat{c}}{\texttt{=}} \hat{x}')$, $\hat{c} = (\hat{x} \texttt{=} \hat{x}_2''\ \hat{x}_3'')$, $\langle \hat{a}_1', \hat{\sigma}_1 \rangle \in \hat{G}(\hat{c}, \hat{x}_2'')$, $\langle \hat{a}_2', \hat{\sigma}_2 \rangle \in \hat{G}(\hat{c}, \hat{x}_3'')$,
   $(\texttt{fun}\ \hat{x}_4''\ \texttt{->}\ (\ \hat{e}\ )) = \langle\!|\hat{\sigma}|\!\rangle$, *and* $\mathrm{RV}(\hat{e}) = \hat{x}'$, *then*
   $((\hat{G}(\hat{a}_1, \hat{x}') \ltimes \mathbb{D}\hat{c}) \leftleftarrows \hat{\sigma}_1 \leftleftarrows \hat{\sigma}_2) \subseteq \hat{\Phi}$.
   (c) FUNCTION ENTER: NON-LOCAL VARIABLE
   If $\hat{a}_1 = (\hat{x}'' \overset{\complement\hat{c}}{\texttt{=}} \hat{x}')$, $\hat{c} = (\hat{x}_1'' \texttt{=} \hat{x}_2''\ \hat{x}')$, $\hat{x}'' \neq \hat{x}$, *and* $\langle \hat{a}_1', \hat{\sigma}_1 \rangle \in \hat{G}(\hat{a}_1, \hat{x}_2'')$, $\langle \hat{a}_2', \hat{\sigma}_2 \rangle \in \hat{G}(\hat{a}_1, \hat{x}')$, *then*
   $((\hat{G}(\hat{a}_1', \hat{x}) \leftharpoonup \hat{\sigma}_1) \leftleftarrows \hat{\sigma}_2 \ltimes \complement\hat{c}) \subseteq \hat{\Phi}$.
3. ***Conditional wiring***
   (a) CONDITIONAL ENTER: PARAMETER POSITIVE
   If $\hat{a}_1 = (\hat{x}' \overset{\complement\hat{c}}{\texttt{=}} \hat{x}_1)$, $\hat{c} = (\hat{x}_2 \texttt{=} \hat{x}_1 \sim \hat{p}\ \texttt{?}\ \hat{f}_1 : \hat{f}_2)$, $\hat{f}_1 = \texttt{fun}\,\hat{x}'\ \texttt{->}\ (\ \hat{e}\ )$, *and* $\hat{x} \in \{\hat{x}', \hat{x}_1\}$, *then*
   $\{\hat{\phi} | \hat{\phi} \in \hat{G}(\hat{a}_1, \hat{x}_1) \wedge \textsc{Match}(\langle\!|\hat{\phi}|\!\rangle, \hat{p})\} \subseteq \hat{\Phi}$.
   (b) CONDITIONAL ENTER: PARAMETER NEGATIVE
   If $\hat{a}_1 = (\hat{x}' \overset{\complement\hat{c}}{\texttt{=}} \hat{x}_1)$, $\hat{c} = (\hat{x}_2 \texttt{=} \hat{x}_1 \sim \hat{p}\ \texttt{?}\ \hat{f}_1 : \hat{f}_2)$, $\hat{f}_2 = \texttt{fun}\,\hat{x}'\ \texttt{->}\ (\ \hat{e}\ )$, *and* $\hat{x} \in \{\hat{x}', \hat{x}_1\}$, *then*
   $\{\hat{\phi} | \hat{\phi} \in \hat{G}(\hat{a}_1, \hat{x}_1) \wedge \neg\textsc{Match}(\langle\!|\hat{\phi}|\!\rangle, \hat{p})\} \subseteq \hat{\Phi}$.
   (c) CONDITIONAL ENTER: NON-PARAMETER POSITIVE
   If $\hat{a}_1 = (\hat{x}' \overset{\complement\hat{c}}{\texttt{=}} \hat{x}_1)$, $\hat{c} = (\hat{x}_2 \texttt{=} \hat{x}_1 \sim \hat{p}\ \texttt{?}\ \hat{f}_1 : \hat{f}_2)$, $\hat{f}_1 = \texttt{fun}\,\hat{x}'\ \texttt{->}\ (\ \hat{e}\ )$, $\hat{x} \notin \{\hat{x}', \hat{x}_1\}$, $\langle \hat{a}', \hat{\sigma} \rangle \in \hat{G}(\hat{c}, \hat{x}_1)$, *and* $\textsc{Match}(\langle\!|\hat{\sigma}|\!\rangle, \hat{p})$, *then*
   $(\hat{G}(\hat{a}_1, \hat{x}) \leftleftarrows \hat{\sigma}) \subseteq \hat{\Phi}$.
   (d) CONDITIONAL ENTER: NON-PARAMETER NEGATIVE
   If $\hat{a}_1 = (\hat{x}' \overset{\complement\hat{c}}{\texttt{=}} \hat{x}_1)$, $\hat{c} = (\hat{x}_2 \texttt{=} \hat{x}_1 \sim \hat{p}\ \texttt{?}\ \hat{f}_1 : \hat{f}_2)$, $\hat{f}_2 = \texttt{fun}\,\hat{x}'\ \texttt{->}\ (\ \hat{e}\ )$, $\hat{x} \notin \{\hat{x}', \hat{x}_1\}$, $\langle \hat{a}', \hat{\sigma} \rangle \in \hat{G}(\hat{c}, \hat{x}_1)$, *and* $\neg\textsc{Match}(\langle\!|\hat{\sigma}|\!\rangle, \hat{p})$, *then*
   $(\hat{G}(\hat{a}_1, \hat{x}) \leftleftarrows \hat{\sigma}) \subseteq \hat{\Phi}$.

(e) CONDITIONAL EXIT: RETURN POSITIVE

If $\hat{a}_1 = (\hat{x} \overset{\mathbb{D}\hat{c}}{=} \hat{x}')$, $\hat{c} = (\hat{x} = \hat{x}_1 \sim \hat{p}\,?\,\hat{f}_1 : \hat{f}_2)$, $\hat{f}_1 = \texttt{fun}\,\hat{x}''\,\texttt{->}\,(\,\hat{e}\,)$, $\mathrm{RV}(\hat{e}) = \hat{x}'$, $\langle \hat{a}', \hat{\sigma} \rangle \in \hat{G}(\hat{c}, \hat{x}_1)$, *and* $\textsc{Match}(\langle\!|\hat{\sigma}|\!\rangle, \hat{p})$, *then*
$(\hat{G}(\hat{a}_1, \hat{x}') \leftleftarrows \hat{\sigma}) \subseteq \hat{\Phi}$.

(f) CONDITIONAL EXIT: RETURN NEGATIVE

If $\hat{a}_1 = (\hat{x} \overset{\mathbb{D}\hat{c}}{=} \hat{x}')$, $\hat{c} = (\hat{x} = \hat{x}_1 \sim \hat{p}\,?\,\hat{f}_1 : \hat{f}_2)$, $\hat{f}_2 = \texttt{fun}\,\hat{x}''\,\texttt{->}\,(\,\hat{e}\,)$, $\mathrm{RV}(\hat{e}) = \hat{x}'$, $\langle \hat{a}', \hat{\sigma} \rangle \in \hat{G}(\hat{c}, \hat{x}_1)$, *and* $\neg\textsc{Match}(\langle\!|\hat{\sigma}|\!\rangle, \hat{p})$, *then*
$(\hat{G}(\hat{a}_1, \hat{x}') \leftleftarrows \hat{\sigma}) \subseteq \hat{\Phi}$.

4. ***Record projection***

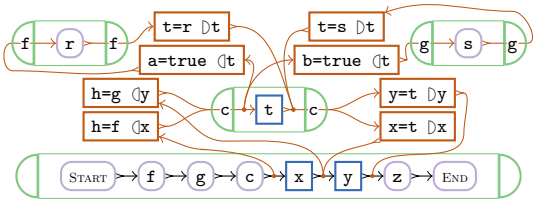
(a) RECORD PROJECTION

If $\hat{a}_1 = (\hat{x} = \hat{x}'\,.\,\ell)$, $\langle \hat{a}', \hat{\sigma} \rangle \in \hat{G}(\hat{a}_1, \hat{x}')$, $\langle\!|\hat{\sigma}|\!\rangle = \hat{r}$, *and* $(\ell = \hat{x}'') \in \hat{r}$, *then*
$(\hat{G}(\hat{a}', \hat{x}'') \leftarrow\!\!\rho\, \hat{\sigma}) \subseteq \hat{\Phi}$.

We will write $\hat{v} \in \hat{G}(\hat{c}, \hat{x})$ as an abbreviation for $\hat{v} = \langle\!|\hat{\phi}|\!\rangle$ for some $\hat{\phi} \in \hat{G}(\hat{c}, \hat{x})$.

Understanding lookup. There is a lot going on in Definition 7. To better understand the clauses, we will trace the lookup example of Section 2.2. Below is that example translated to the formal A-normalized grammar we use in this section, retaining integer and string data from the original example for clarity. (Note that there is only one of each, so singleton properties hold.) The diagram to the right is the final CFG produced by the wiring rules described in Section 3.3 below; here we illustrate variable lookup on this final CFG.

```
1 f = fun a -> ( r = 4 );
2 g = fun b -> ( s = "s" );
3 c = fun h -> ( t = h true );
4 x = c f;
5 y = c g;
6 z = x;
```



There are a few differences between this CFG and the informal presentation of Section 2.2 – this CFG precisely matches the dependency graph notation of Figure 3. Each black or brown arrow-tipped edge in the CFG corresponds to a dependency $\hat{a} \ll \hat{a} \in \hat{G}$, for $\hat{G}$ being the whole CFG. The green half circles are only block delimiters for readability and do not correspond to nodes – the brown wiring edges go through them. The rectangular brown nodes are the wiring nodes $\hat{x} \overset{\mathbb{C}\hat{c}}{=} \hat{x}'$ and $\hat{x} \overset{\mathbb{D}\hat{c}}{=} \hat{x}'$ of the $\hat{a}$ grammar, and the whole CFG starts at node START and ends with END. The program is in ANF and all program points have names (except for the dummy argument in the function call at `t`); the result of function `f`, for instance, is named `r`.

Recall from the overview that we desire to look up the final value, $\hat{\texttt{z}}$, from the end of the program: $\hat{G}(\textsc{End}, \hat{\texttt{z}})$. (We coventionally begin the search not at the indicated node, e.g. END here, but at any predecessors to this node). Looking back from END, we see `z` defined as `x`; rule 1b of Definition 7 tells us to first continue by looking up $\hat{\texttt{x}}$ from $\hat{\texttt{z}}$: $\hat{G}(\hat{\texttt{z}}, \hat{\texttt{x}})$. (We will use the variable – $\hat{\texttt{z}}$ here –

alone as shorthand for its clause). We proceed lookup with rule 1c, which allows us to skip the clause `y = c g` as it does not affect $\hat{\mathtt{x}}$. Our lookup is then $\hat{G}(\hat{\mathtt{y}}, \hat{\mathtt{x}})$.

From $\hat{\mathtt{y}}$, rule 2b applies, which first entails looking up both function $\hat{\mathtt{c}}$ and argument $\hat{\mathtt{f}}$ from point $\hat{\mathtt{x}}$; these two lookups are both straightforward under the variable search rules 1a and 1c. Each returns a store which then gets parallel-merged to $\langle \hat{\mathtt{c}}@[] \mapsto \widehat{\mathtt{fun_h}}, \{\hat{\mathtt{c}}@[] \mapsto \widehat{\mathtt{fun_h}}, \hat{\mathtt{f}}@[] \mapsto \widehat{\mathtt{fun_4}}\}\rangle$. Continuing, it remains to look up $\hat{\mathtt{t}}$ from the wiring node, suffix the result with the trace $[\rhd\hat{\mathtt{x}}]$, and merge that result with the aforementioned store. Note that this merge requires all three to be in sync, a key to giving us path- and context-sensitivity.

The fact that we align the argument lookup in particular corresponds with what CPA [Age95,Bes09,VS10] achieves. CPA separately analyzes function calls with differing types of arguments. DRSF does so as well by creating for each argument type a distinct relative store fragment (the result of the lookup of $\hat{G}(\hat{c}, \hat{x}_3'')$ in rule 2b). This alignment requires only one operation and no special machinery in DRSF, as relative store fragments are already used to model many other kinds of inter-binding relationships as well.

We now tackle the lookup of $\hat{\mathtt{t}}$. We begin by exploring call site `t = h true`: again, rule 2b applies. Here, there are two candidate wiring nodes: $\mathtt{t} \overset{\rhd\mathtt{t}}{=} \mathtt{r}$ and $\mathtt{t} \overset{\rhd\mathtt{t}}{=} \mathtt{s}$; both are possible, as either $\widehat{\mathtt{fun_4}}$ or $\widehat{\mathtt{fun_s}}$ may reach this call site. By rule 1a, looking back through node $\mathtt{t} \overset{\rhd\mathtt{t}}{=} \mathtt{r}$ yields $\widehat{\mathtt{int}}$ while node $\mathtt{t} \overset{\rhd\mathtt{t}}{=} \mathtt{s}$ yields $\widehat{\mathtt{string}}$. Clearly only the $\widehat{\mathtt{int}}$ is possible at run-time. What does DRSF do here?

Lookup in DRSF can in general produce multiple (singleton) stores representing multiple abstract values. The lookup $\hat{G}(\mathtt{t} \overset{\rhd\mathtt{t}}{=} \mathtt{r}, \hat{\mathtt{r}})$ produces the store $\langle \hat{\mathtt{r}}@[] \mapsto \widehat{\mathtt{int}}, \{\hat{\mathtt{r}}@[] \mapsto \widehat{\mathtt{int}}\}\rangle$, which is then suffixed with $\rhd\hat{\mathtt{t}}$. The resulting store $\langle \hat{\mathtt{r}}@[\rhd\hat{\mathtt{t}}] \mapsto \widehat{\mathtt{int}}, \{\hat{\mathtt{r}}@[\rhd\hat{\mathtt{t}}] \mapsto \widehat{\mathtt{int}}\}\rangle$ is then merged with the store used to verify that $\widehat{\mathtt{fun_4}}$ could be called at this site, yielding $\langle \hat{\mathtt{r}}@[\rhd\hat{\mathtt{t}}] \mapsto \widehat{\mathtt{int}}, \{\hat{\mathtt{r}}@[\rhd\hat{\mathtt{t}}] \mapsto \widehat{\mathtt{int}}, \hat{\mathtt{f}}@[\lhd\hat{\mathtt{x}}] \mapsto \widehat{\mathtt{fun_4}}\}\rangle$. At this point, the lookup of $\hat{\mathtt{t}}$ is complete and trace $[\rhd\hat{\mathtt{x}}]$ can finally be suffixed to this result, giving $\langle \hat{\mathtt{r}}@[\rhd\hat{\mathtt{t}}, \rhd\hat{\mathtt{x}}] \mapsto \widehat{\mathtt{int}}, \{\hat{\mathtt{r}}@[\rhd\hat{\mathtt{t}}, \rhd\hat{\mathtt{x}}] \mapsto \widehat{\mathtt{int}}, \hat{\mathtt{f}}@[] \mapsto \widehat{\mathtt{fun_4}}, \hat{\mathtt{c}}@[] \mapsto \widehat{\mathtt{fun_h}}\}\rangle$ for $\hat{\mathtt{x}}$.

The lookup $\hat{G}(\mathtt{t} \overset{\rhd\mathtt{t}}{=} \mathtt{s}, \hat{\mathtt{s}})$, demonstrates how store merging prunes impossible paths. This lookup produces the store $\langle \hat{\mathtt{s}}@[] \mapsto \widehat{\mathtt{string}}, \{\hat{\mathtt{s}}@[] \mapsto \widehat{\mathtt{string}}\}\rangle$, which we then suffix and merge as we did above to yield $\langle \hat{\mathtt{s}}@[\rhd\hat{\mathtt{t}}] \mapsto \widehat{\mathtt{string}}, \{\hat{\mathtt{s}}@[\rhd\hat{\mathtt{t}}] \mapsto \widehat{\mathtt{string}}, \hat{\mathtt{g}}@[\lhd\hat{\mathtt{y}}] \mapsto \widehat{\mathtt{fun_s}}\}\rangle$. Next, this store should be suffixed with $\rhd\hat{\mathtt{x}}$ to adjust the perspective to be back out of the $\hat{\mathtt{x}}$ call site, but this is *undefined*: the suffixing $\hat{\mathtt{g}}@[\lhd\hat{\mathtt{y}}] \ltimes \rhd\hat{\mathtt{x}}$, is undefined because reading in the forward-running direction it indicates a call at site $\hat{\mathtt{y}}$ but which returns to site $\hat{\mathtt{x}}$, an impossibility at run-time. So, this store is eliminated at the merge, and lookup of $\hat{G}(\textsc{End}, \hat{\mathtt{z}})$ yields only the $\widehat{\mathtt{int}}$ store described above.

Conditionals and records are variations on how functions are handled: the CFG is traversed in reverse, and stores are merged and relativized along the way as necessary. The parallel merges in conditional lookup give path-sensitivity on the conditional value: only aligned stores can merge.

3.3 Abstract Evaluation

We now present the single-step abstract evaluation relation which incrementally adds edges to build a full CFG. The above lookup function is the key subroutine.

Active nodes. To preserve standard evaluation order, we define the notion of an Active node: only nodes with all previous nodes already executed can fire. This serves a purpose similar to an evaluation context in operational semantics.

Definition 8. $\textsc{Active}(\hat{a}', \hat{G})$ *iff path* $\{\textsc{Start} \ll \hat{a}_1, \ldots, \hat{a}_i \ll \hat{a}_{i+1}, \ldots, \hat{a}_n \ll \hat{a}'\} \subseteq \hat{G}$ *such that no* $\hat{a}_i$ *is of one of the forms* $\hat{x} = \hat{x}'\ \hat{x}''$ *or* $\hat{x} = \hat{x}' \sim \hat{p}\ ?\ \hat{f} : \hat{f}'$.

Wiring. Function application requires the concrete function body to be "wired" into the call site:

Definition 9. *Let* $\textsc{Wire}(\hat{c}, \texttt{fun}\ \hat{x}_0\ \texttt{->}\ \hat{e}, \hat{x}_1, \hat{x}_2) =$
$\{\hat{c}' \ll (\hat{x}_0 \overset{\hat{c}}{=} \hat{x}_1) \mid \hat{c}' \overset{\hat{G}}{\ll} \hat{c}\} \cup \{(\hat{x}_0 \overset{\hat{c}}{=} \hat{x}_1) \ll \hat{e} \ll (\hat{x}_2 \overset{\hat{c}}{=} \mathrm{RV}(\hat{e}))\} \cup \{(\hat{x}_2 \overset{\hat{c}}{=} \mathrm{RV}(\hat{e})) \ll \hat{c}' \mid \hat{c} \overset{\hat{G}}{\ll} \hat{c}'\}$.

$\hat{c}'$ here is the call site, and we are wiring in function body $\hat{e}$ at this site.

$$\textsc{Application}\quad \frac{\hat{c} = (\hat{x}_1 = \hat{x}_2\ \hat{x}_3) \qquad \textsc{Active}(\hat{c}, \hat{G}) \qquad \hat{f} \in \hat{G}(\hat{c}, \hat{x}_2) \qquad \hat{G}(\hat{c}, \hat{x}_3) \neq \emptyset}{\hat{G} \longrightarrow^1 \hat{G} \cup \textsc{Wire}(\hat{c}, \hat{f}, \hat{x}_3, \hat{x}_1)}$$

$$\textsc{Record Conditional True}\quad \frac{\hat{c} = (\hat{x}_1 = \hat{x}_2 \sim \hat{p}\ ?\ \hat{f}_1 : \hat{f}_2) \qquad \textsc{Active}(\hat{c}, \hat{G}) \qquad \exists \hat{\phi} \in \hat{G}(\hat{c}, \hat{x}_2).\ \textsc{Match}(\langle\!|\hat{\phi}|\!\rangle, \hat{p})}{\hat{G} \longrightarrow^1 \hat{G} \cup \textsc{Wire}(\hat{c}, \hat{f}_1, \hat{x}_2, \hat{x}_1)}$$

$$\textsc{Record Conditional False}\quad \frac{\hat{c} = (\hat{x}_1 = \hat{x}_2 \sim \hat{r}\ ?\ \hat{f}_1 : \hat{f}_2) \qquad \textsc{Active}(\hat{c}, \hat{G}) \qquad \exists \hat{\phi} \in \hat{G}(\hat{c}, \hat{x}_2).\ \neg\textsc{Match}(\langle\!|\hat{\phi}|\!\rangle, \hat{p})}{\hat{G} \longrightarrow^1 \hat{G} \cup \textsc{Wire}(\hat{c}, \hat{f}_2, \hat{x}_2, \hat{x}_1)}$$

Fig. 4. Abstract Evaluation Rules

Next, we define the abstract small-step relation $\longrightarrow^1$ on graphs; see Figure 4. The evaluation rules end up being straightforward after the above preliminaries. For application, if $\hat{c}$ is an Active call site, lookup of the function variable $\hat{x}_2$ returns function body $\hat{f}$ and *some* value $\hat{v}$ can be looked up at the argument position, we may wire in $\hat{f}$'s body to this call site. Note that $\hat{v}$ is only observed here to constrain evaluation order to be call-by-value. The case clause rules are similar. We define the small step relation $\longrightarrow^1$ to hold if a proof exists in the system in Figure 4. We write $\hat{G}_0 \longrightarrow^* \hat{G}_n$ to denote $\hat{G}_0 \longrightarrow^1 \hat{G}_1 \longrightarrow^1 \ldots \longrightarrow^1 \hat{G}_n$.

Now we can finally put it all together to analyze a program. We define a typical program abstraction function $\textsc{Embed}(e)$ to denote the lifting of an expression e into a CFG $\hat{G}$:

Definition 10. $\textsc{Embed}([c_1, \ldots, c_n])$ *is the graph* $\hat{G}_0 = \{\textsc{Start} \ll \hat{c}_1, \ldots, \hat{c}_i \ll \hat{c}_{i+1}, \ldots, \hat{c}_n \ll \textsc{End}\}$, *where each* $\hat{c}_i = c_i$.

This initial graph is simply the linear sequence of clauses in the "main program". The *DRSF Analysis* for a program e is then graph $\hat{G}$ where $\textsc{Embed}(e) \longrightarrow^* \hat{G}$ and $\hat{G}$ can only further step to itself.

3.4 DRSF Soundness

Soundness of DRSF is proven in two steps: we first demonstrate equivalence between a standard small step operational semantics and a *graph-based* operational semantics [PS16]; we then show that DRSF conservatively approximates the latter. We begin by overloading $\longrightarrow^1$ to a canonical small step evaluation relation on expressions e (straightforward and elided for reasons of space). Next, we define a graph-based operational semantics using ωDRSF: DRSF where trace concatenation $\ltimes$ never loses information (and so never produces partial traces). We use unhatted analogues of the grammars in Figures 2 and 3 for ωDRSF entities. We also overload the operators on traces and stores to work similarly on the unhatted grammar, where trace concatenation has no maximum trace length.

We align these two operational semantics by defining a bisimulation $e \cong G$ and showing that it holds throughout evaluation. Key to this bisimulation is an alignment between the freshenings of bound variables in e and the traces in the mappings of concrete store fragments in G. We state the equivalence of these operational semantics as follows:

Lemma 1 (Equivalence of Operational Semantics). *For any $e \cong G$:*

if $e \longrightarrow^1 e'$ then $G \longrightarrow^* G'$ such that $e' \cong G'$; and
if $G \longrightarrow^1 G'$ then $e \longrightarrow^* e'$ such that $e' \cong G'$.

The discrepancy in the number of steps arises in how the operational semantics process variable aliasing. The small step operational semantics processes aliases eagerly while ωDRSF need not step for variable aliases (due to demand-driven lookup).

Given the equivalence of these operational semantics, we now show kDRSF simulates ωDRSF. We do this via a simulation relation $\preccurlyeq$ defined from each non-hatted term to its hatted analogue. The only non-trivial case here is that of traces, in which a full trace is simulated by the partial trace of any of its suffixes (e.g. $[\mathbb{D}x_1, \mathbb{d}x_2] \preccurlyeq (\mathbb{d}\hat{x}_2]$).

Lemma 2 (Simulation). *If $G \preccurlyeq \hat{G}$ and $G \longrightarrow^1 G'$, then $\hat{G} \longrightarrow^* \hat{G}'$ such that $G' \preccurlyeq \hat{G}'$.*

Lemma 2 is proven by establishing a Galois connection between ωDRSF and kDRSF following standard techniques [CC77,NNH99,VHM10,Mig10a]. Soundness of the overall analysis is an immediate corollary of the above two lemmas.

3.5 DRSF Complexity

DRSF as presented above has exponential worst-case complexity, but this can be mitigated. We now outline a proof of this property to understand its singular cause. Fortunately, our initial experiments suggest that this worst case is not common in practice; see Section 4.

DRSF's complexity is shown by bounding the size of $\hat{G}$ produced by the abstract evaluation rules of Figure 4. We begin by counting the forms of the grammar in Figure 2. Restricting that grammar to the program being analyzed

and to traces of length at most k, we observe there are a polynomial number of forms of $\hat{\delta}$, $\hat{\Delta}$, $\hat{\rho}$, and $\hat{\psi}$.

The exponential growth of DRSF lies in the definition of $\hat{\Psi}$, which admits any subset of the $\hat{\psi}$ forms above. This leads us to the following key Lemma:

Lemma 3 (Exponential Raw Store Growth). *Restrict the grammar of Figure 2 to contain only clauses, values, and variables appearing within $\hat{e}$ and only traces of length at most k. Let $|L_{\hat{\Psi}}|$ be the number of values of form $\hat{\Psi}$ and let $|L_{\hat{\sigma}}|$ be the number of values of form $\hat{\sigma}$. Then $|L_{\hat{\Psi}}|$ and $|L_{\hat{\sigma}}|$ are $O(2^{n^{k+2}})$.*

The rest of the proof demonstrates that DRSF is polynomial in $|L_{\hat{\sigma}}|$. Definition 7's lookup operation is reduced to a reachability problem on a push-down automaton [PS16] (which is polynomial in the size of the automaton [BEM97]) and the abstract evaluation relation $\hat{G} \longrightarrow^1 \hat{G}'$ is shown to be confluent, bounding the number of lookups to a polynomial of $|L_{\hat{\sigma}}|$. This leads us to:

Theorem 1 (DRSF Analysis Complexity). *The DRSF analysis of a program e is computable in time polynomial in $|L_{\hat{\sigma}}|$.*

3.6 Weakening DRSF

While the exponential case of DRSF appears to be uncommon, it may ultimately be necessary to apply weakenings to guarantee polynomial behavior. A full exploration of such weakenings is beyond the scope of this paper, but we outline one such weakening here.

Consider adding a "time-to-live" index to each mapping. Mappings with TTL are written $\hat{\rho} \overset{\ell}{\mapsto} \hat{v}$, where $\ell \in \mathbb{N}$. Each singleton store creates its single mapping with ℓ equal to some fixed value d. Finally, the raw store merge operation $\oplus$ decreases the value of each mapping's ℓ, and discards mappings with $\ell = 0$. When merging two mappings with the same $\hat{\rho}$ and $\hat{v}$, the maximum ℓ is retained.

With this TTL index, it is possible to prove no raw store can be created with a number of mappings greater than 2^d. Since d is a fixed constant, this bounds the number of mappings per raw store to be fixed. With only polynomially many stores, it follows from a lemma similar to Lemma 3 that $|L_{\hat{\sigma}}|$ is also polynomial, and by Theorem 1, the analysis with TTL is polynomial.

4 Implementation

An implementation of DRSF is available on GitHub[3] which additionally includes binary operators, deep pattern matching, and state as well as integers and strings for basic values. The abstraction function maps all integers and strings to the same abstract integer and abstract string, so the singleton abstractions are lost for them. The implementation proceeds as described in Section 3.5: lookups are performed by reduction to a reachability problem on a push-down automaton just as in DDPA [PS16]. We improve on previous reachability algorithms [BEM97,JSE+14,EMH10] by generalizing over patterns appearing in the automaton which arise due to the form of DRSF's lookup definition. We also rely upon the confluence of abstract evaluation to reuse work performed during lookup, even across lookup invocations.

[3] `https://github.com/JHU-PL-Lab/odefa/tree/sas2017-drsf`

4.1 Evaluation

To evaluate the performance of DRSF, we benchmarked it against an implementation of DDPA [PS16,PF16]. The analysis client in these experiments answers the question "what are all possible values for all variables at the top level of the program?" We used this client to compare both running times and precision.

The test cases were taken from other higher-order program analysis evaluations [GLA+16,JLMVH13]. These Scheme programs were automatically translated to our core language. Many of these benchmarks are moderate sized programs which represent real-world uses.[4] Others are micro-benchmarks designed to stress particular aspects of the analysis.[5]

We conducted three experiments to measure the performance of the analyses.[6] For a monomorphic baseline we selected $k = 0$. For the second and third experiments, we selected $k = 2$ and $k = 4$ somewhat arbitrarily: on several of the benchmarks, $k = 4$ gave full precision, and $k = 2$ is a midpoint. The results are shown in Figure 5.

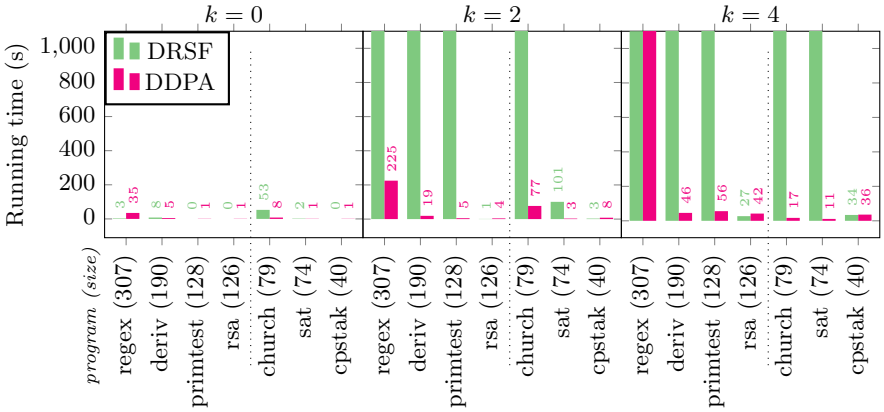


Fig. 5. The benchmark results. Numerical labels on bars are running times. Unlabeled bars timed out after 30 minutes. Numbers in parentheses are program point counts. The dotted lines separate the real-world programs from the micro-benchmarks.

DDPA is faster in all test cases, with a few exceptions, in which the difference between DDPA and DRSF is negligible. This is an expected result when comparing a polynomial analysis (DDPA) to an exponential one (DRSF). Both `regex` and `deriv` involve recursive functions operating on symbols and lists of symbols, which we encode as records in DRSF. We conjecture that DRSF's performance degenerates in these cases because they contain recursive functions, which is a

[4] `regex` is a regular expression engine using derivatives; `rsa` performs the RSA encryption algorithm; `primtest` is the Fermat primality testing algorithms; and `deriv` performs symbolic derivation of an equation.

[5] `sat` is a SAT solver, which stresses path- and context-sensitivity; `cpstak` is the TAK micro-benchmark in continuation-passing style, which stresses nested function calls and non-local lookups; and `church` tests the distributive property of multiplication over addition on Church numerals which includes polymorphic recursive functions and a massive number of function calls, stressing the function-wiring part of the analysis.

[6] Intel(R) Xeon(R) CPU E31220 @ 3.10GHz, 8GB of RAM, Debian GNU/Linux 8.8.

know weak spot in both analyses. The `regex` test case triggers this bad behavior more intensively because it includes a set of mutually recursive functions, which causes even DDPA to time out.

Besides the test cases we ran for performance measurement, we also tested the implementation with micro-benchmarks which serve primarily to exercise context-, path- and flow-sensitivity, as well as the capacity to precisely approximate the heap state with must-alias analysis. For most test cases, DRSF loses no precision other than value abstraction (e.g. $\texttt{5} \mapsto \widehat{\texttt{int}}$) given a sufficient k. Our implementation confirms the expressiveness improvements predicted in theory; for example, DDPA loses precision in the program from Section 2.1, and DRSF approximates it exactly, due to path-sensitivity. Further empirical expressiveness evaluations are left for future work due to the significant loss of precision in recursive functions, mentioned above. This impedes the development of more sophisticated clients because recursive programs, in particular those with polymorphic recursion, exhaust the abstract heap, regardless of our choice for k, causing the analyses to lose call-return alignment, and context- and path-sensitivity. This problem regarding recursion is orthogonal to the advancements in DRSF and we plan to address it in future work using a refined model of finitization for the abstract traces, for example, one based on regular expressions [GLA+16].

The abstract stores produced by DRSF may suffice to solve the generalized environment problem [Mig10b,Shi91,BFL+14,GM17]. Unfortunately, the most relevant clients for this feature are difficult to test in isolation. For example, Super-β inlining exercises the resolution of circular dependencies that occur in recursive programs. As discussed above, DRSF loses precision in those cases, so we plan to evaluate these clients after addressing this weakness.

Finally, we attempted to compare DRSF to state-of-the-art forward analyses including P4F [GLA+16] and OAAM [JLMVH13]. Unfortunately, these analyses are too different from DRSF to draw any significant conclusion. OAAM's reference implementation does not support $k > 0$ resulting in lower precision but faster running times. DRSF runs faster than the P4F reference implementation, but the latter was designed for correctness and not performance so this is arguably not a fair comparison.

5 Related Work

Anodization [Mig10b] aims for singleton abstraction by separating bindings so that there is only one concrete binding per abstract binding, making a structure similar to our relative store fragments. Anodization does not create local, relativized stores like we do here and is not a demand-driven analysis. It also has not been implemented or analyzed for complexity. Other higher-order analyses that incorporate a form of singleton abstraction include [JTWW98,SW97]; the former is carefully engineered to achieve a quartic time bound but is focused on must-alias analysis and lacks path- or general context-sensitivity.

ΔCFA's delta frame strings [MS06a,GM17] are very similar in structure to our relative traces, but due to differences in forward- and reverse-analyses they are incorporated into the analysis quite differently. In DRSF they are placed on

variables on the store to singularize them; in ΔCFA they are part of the state. Moreover, ΔCFA defines an abstraction for frame strings which loses more information than DRSF; for example, it discards the order of invocations of different procedures. A recent ΔCFA extension [GM17] shows how the environment problem can be solved, but there is no implementation or evaluation.

There are many different approaches to building path-sensitivity into program analyses. Early work on path-sensitivity directly built the logic into an analysis [BA98]; subsequently, path-sensitivity was was implemented using SMT model-checkers over boolean-valued programs [DLS02,XCE03]. Some higher-order type systems contain analogues of path-sensitive precision [THF10]. Recent work shows how path sensitivity can be viewed as one of several orthogonal dimensions of analysis expressiveness along with context- and flow-sensitivity [SDM$^+$13,DMH15]. In the first-order abstract interpretation literature, trace partitioning techniques give an added expressiveness that is related to DRSF traces [HT98,Bou92] – like DRSF these techniques add path information to the analysis state and path-sensitivity becomes an emergent property.

DRSF's store fragments are related to the heap fragments of separation logic [Rey02]: there are many partial stores for different program contexts which are merged. This analogy with separation logic is only high level since we are comparing first-order program logics to higher-order program analyses, but it points to potential applicability of store fragments to higher-order program logics.

We build on the DDPA higher-order demand-driven analysis [PS16]. Other demand-driven higher-order analyses include [RF01,FRD00,SDAB16], but these are flow-insensitive; they as well as DDPA lack a singleton abstraction. In [PS16] call-return alignment can fully replace the need for an explicit context-sensitivity mechanism, but it requires an explicit call stack. This comes for free in DRSF since it is embedded in the store variable traces.

The exponential problem we run into here is a fundamental issue in path-sensitive analyses. DRSF attemps to infer when path sensitivity is needed, erring on the side of precision rather than performance. SMT solvers are useful in [DLS02,XCE03] in part for their ability to handle a large state-space search.

6 Conclusions

In this paper we develop a general notion of singleton abstraction, *relative store fragments*, that cuts across many of the classic expressiveness dimensions of higher-order program analysis: the resulting analysis, DRSF, exhibits flow-, context-, path- and must-alias sensitivity solely through an accurate and compositional singleton abstraction. We give a formal definition of DRSF, sketch its soundness and complexity, and report on initial results from an implementation.

Future work. While DRSF is usually avoiding its exponential worst-case complexity in practice, this issue will need to be addressed in a more realistic implementation. Also, like DDPA and other analyses such as kCFA, DRSF finitely unrolls recursive cycles and this usually adds no expressiveness as the analysis contour depth k gets larger, but can adversely affect running times; we plan to study how to avoid k-unrolling of recursive cycles when it is not helpful.

Acknowledgments. The authors thank the anonymous reviewers for helpful suggestions which improved the final version of the paper.

References

[Age95] Ole Agesen. The cartesian product algorithm. In *ECOOP*, volume 952 of *Lecture Notes in Computer Science*, 1995.

[BA98] Rastisalv Bodík and Sadun Anik. Path-sensitive value-flow analysis. In *POPL*, 1998.

[BEM97] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR'97*, 1997.

[Bes09] Frédéric Besson. CPA beats ∞-CFA. In *Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs*, 2009.

[BFL+14] Lars Bergstrom, Matthew Fluet, Matthew Le, John Reppy, and Nora Sandler. Practical and effective higher-order optimizations. In *ICFP*, 2014.

[Bou92] François Bourdoncle. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming*, 1992.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[CWZ90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *PLDI*, 1990.

[DGS97] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Trans. Program. Lang. Syst.*, 19(6):992–1030, November 1997.

[DLS02] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: path-sensitive program verification in polynomial time. In *PLDI*, 2002.

[DMH15] David Darais, Matthew Might, and David Van Horn. Galois transformers and modular abstract interpreters. In *OOPSLA*, 2015.

[EMH10] Christopher Earl, Matthew Might, and David Van Horn. Pushdown control-flow analysis of higher-order programs. In *Workshop on Scheme and Functional Programming*, 2010.

[FRD00] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In *PLDI*, 2000.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI*, 1993.

[GLA+16] Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. Pushdown control-flow analysis for free. In *POPL*, 2016.

[GM17] Kimball Germane and Matthew Might. A posteriori environment analysis with pushdown Delta CFA. In *POPL*, 2017.

[HT98] Maria Handjieva and Stanislav Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *Static Analysis Symposium (SAS)*, 1998.

[HT01] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *PLDI*, 2001.

[JLMVH13] J. Ian Johnson, Nicholas Labich, Matthew Might, and David Van Horn. Optimizing Abstract Abstract Machines. In *ICFP*, 2013.

[JSE+14] J. Ian Johnson, Ilya Sergey, Christopher Earl, Matthew Might, and David Van Horn. Pushdown flow analysis with abstract garbage collection. *JFP*, 2014.

[JTWW98] Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew K. Wright. Single and loving it: Must-alias analysis for higher-order languages. In *POPL*, 1998.

[Mid12] Jan Midtgaard. Control-flow analysis of functional programs. *ACM Comput. Surv.*, 2012.

[Mig07] Matthew Might. *Environment Analysis of Higher-Order Languages*. PhD thesis, Georgia Institute of Technology, June 2007.

[Mig10a] Matthew Might. Abstract interpreters for free. In *Proceedings of the 17th International Conference on Static Analysis*, 2010.

[Mig10b] Matthew Might. Shape analysis in the absence of pointers and structure. In *VMCAI*. Springer-Verlag, 2010.

[MS06a] Matthew Might and Olin Shivers. Environment analysis via ΔCFA. In *POPL*, 2006.

[MS06b] Matthew Might and Olin Shivers. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *ICFP*, Portland, Oregon, 2006.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[PF16] Zachary Palmer and Leandro Facchinetti. DDPA implementation. https://github.com/JHU-PL-Lab/odefa/tree/sas2017-ddpa, 2016.

[PS16] Zachary Palmer and Scott Smith. Higher-order demand-driven program analysis. In *ECOOP*, 2016.

[Rep94] Thomas Reps. Demand interprocedural program analysis using logic databases. In *Application of Logic Databases*, 1994.

[Rey02] John Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[RF01] Jakob Rehof and Manuel Fähndrich. Type-base flow analysis: from polymorphic subtyping to CFL-reachability. In *POPL*, New York, NY, USA, 2001.

[SDAB16] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow- and context-sensitive pointer analysis for Java. In *ECOOP*, 2016.

[SDM+13] Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. Monadic abstract interpreters. In *PLDI*, 2013.

[Shi91] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, 1991. TR CMU-CS-91-145.

[SR05] Diptikalyan Saha and C. R. Ramakrishnan. Incremental and demand-driven points-to analysis using logic programming. In *PPDP*, 2005.

[SW97] Paul A. Steckler and Mitchell Wand. Lightweight closure conversion. *ACM Trans. Program. Lang. Syst.*, 19, 1997.

[THF10] Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *ICFP*, 2010.

[VHM10] David Van Horn and Matthew Might. Abstracting abstract machines. In *ICFP*, 2010.

[VS10] Dimitrios Vardoulakis and Olin Shivers. CFA2: A context-free approach to control-flow analysis. In *European Symposium on Programming*, 2010.

[XCE03] Yichen Xie, Andy Chou, and Dawson Engler. Archer: Using symbolic, path-sensitive analysis to detect memory access errors. In *ESEC/FSE*, 2003.