Near-Concrete Program Interpretation

Paritosh Shroff[1], Christian Skalka[2], and Scott F. Smith[1]

[1] Department of Computer Science, Johns Hopkins University
{pari, scott}@cs.jhu.edu
[2] Department of Computer Science, University of Vermont
skalka@cs.uvm.edu

Abstract. We develop a *near-concrete* program interpretation, a program analysis that aims to cut very close to program execution while retaining decidability. Both in name and in spirit, the approach is similar to abstract interpretation, but models heaps with possibly recursive structure, is path sensitive, and applies in a fully higher-order setting. The main technical contribution is a *prune-rerun* technique for analyzing higher-order recursive functions. To illustrate the expressiveness and usefulness of the system, we show how it can be used to enforce temporal program safety properties and information flow security, and show how it betters state-of-the-art systems on some examples.

1 Introduction

Two critical dimensions of program analysis are the expressiveness of the analysis, and the conciseness of the specification of the analysis. More expressive analyses return better results, which lead to better program understanding and therefore better optimizations, flexibility, and security for programs. Specification conciseness is also becoming more important as program analyses become more and more sophisticated: as the power of the analyses grow, so too does the difficulty of understanding and proving guarantees about them. Previous approaches have proposed to combine myriad analyses to achieve desired results, but it becomes difficult to reason about the complex interactions in these systems. While empirical testing may clarify some issues, a rigorous formal foundation is invaluable to end-user understanding and verification of advanced program analyses.

Our goal is to develop a foundation for program analysis that advances the state-of-the-art, while being based on uniform design principles that scale to a variety of important extensions. The advantage of this uniformity is the simplification of specifications. Our analysis is aimed at higher-order paradigms, including functional programs such as ML or Haskell, and object-oriented languages such as Java. Many of today's industrial-strength analyses [13, 25, 5] focus on first-order programs. A primary contribution of our work is the incorporation of the advances in these systems into a higher-order context.

We call our analysis *near-concrete* interpretation (*N*CI), since we intend to cut as close to the actual program execution (the *concrete* interpretation) as

possible, while retaining decidability. Both in name and in spirit, the approach is close to abstract interpretation [8, 9]. Compared to the existing abstract interpretation literature, *N*CI applies to fully higher-order programs, models nontrivial heap structures including recursive heap structures, and incorporates path sensitivity [11] into a higher-order context.

Intuitions about our basic approach are given in Section 2. The main technical contribution is our *prune-rerun* technique for analyzing higher-order recursive functions, which is described in detail. Formalization of these ideas, including a presentation and discussion of main results such as soundness, are given in Section 3. Several extensions are then outlined in Section 4, including context sensitivity, a mutable heap, and path sensitivity. Due to space limitations the formal details of these extensions are given in the Appendix. To illustrate the expressiveness and usefulness of the system, in Section 5 we discuss applications including enforcement of temporal program safety properties, and information flow security. We show some examples for which our algorithm is more precise than existing approaches. Concluding remarks and related work is covered in Section 6.

2 Overview

Our goal is to obtain an analysis that is very close to the operational semantics of programs, and hence highly precise, but with guaranteed termination even for divergent programs. Thus, the central technical contribution of our analysis is how it deals with recursion, since non-termination in functional programs is due to unbounded recursion and unbounded call stack depth. For example, given:

$$\begin{array}{l} sum\ n = \mathsf{Let}\ r = \mathsf{If}\ (n == 0)\ \mathsf{Then}\ 0 \\ \qquad\qquad\qquad\qquad \mathsf{Else\ Let}\ y = sum\,(n-1)\ \mathsf{In}\ y + n \\ \qquad\quad \mathsf{In}\ r, \end{array}$$

consider the divergent program $sum\,(-5)$, that generates unbounded call stack depth during computation. The problem is, how to bound the analysis of such examples in a sound manner, while retaining most of its computational structure?

2.1 The Core System (*N*CI)

We call our analysis *near-concrete interpretation* (*N*CI), to evoke its close resemblance to the operational semantics or *concrete interpretation* (CI) of programs. The resemblance is so close, in fact, that *N*CI and CI are isomorphic for non-recursive programs. To address recursion, at the heart of *N*CI lies a novel *prune-rerun* technique which we now illustrate, using the *sum* function defined above.

Figure 1(a) defines a Data- and Control-Flow Graph (DCFG) that graphically illustrates the CI of $sum\,(n)$, for arbitrary n. So for example, $sum\,(1000)$ executes the “rec-call cycle” path in the figure a thousand times before the *true*

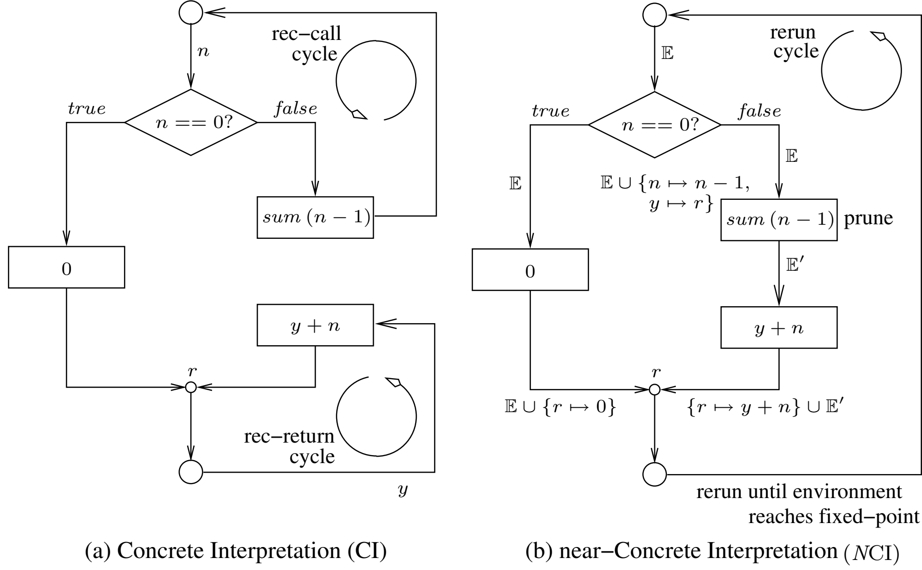

Fig. 1. (a) DCFG of operational semantics of *sum* (n). (b) DCFG of environment ($\mathbb{E}$) based near-concrete interpretation of *sum* (n).

branch is taken, after which the "rec-return cycle" path is followed another thousand times; the invocation *sum* (-5) loops in the rec-call cycle forever. Since it is impossible to predict if a recursive computation will terminate, our solution for *N*CI collapses the recursive computation into a single cycle with a fixed-point property which guarantees the latter is a sound approximation of the former.

Figure 1(b) shows the DCFG of the *N*CI of *sum* (n). This diagram shows the basic outline of how the prune-rerun technique works. There are three major differences between the two DCFGs: 1) The *N*CI environment $\mathbb{E}$ is monotonically increasing over the course of computation, whereas in a CI each stack frame (not shown in the figure) can map variables to different values; 2) The separate rec-call and rec-return cycles of the CI are merged into one single "rerun" cycle in *N*CI, meaning stack-based function execution is simplified to stack-free looping, and 3) recursive invocations *sum* $(n-1)$ are not taken in *N*CI—instead, the argument mapping $n \mapsto n-1$ is added to the environment and, $y \mapsto r$ is immediately added to the environment to demarcate the value return. We now discuss these aspects in detail, to provide clear intuitions about the algorithm.

The NCI Environment The *N*CI environment $\mathbb{E}$ can directly represent recursive structures such as $\{y \mapsto r,\, r \mapsto y+n\}$, and can support multiple bindings of the same variable, such as $\{r \mapsto 0,\, r \mapsto y+n\}$. Formally, the environment is a multi-mapping relation between variables and *near-values*, which are either variables,

conventional values (such as functions, integers, *etc.*) or atomic arithmetic or relational computation expressions of the form $n-1$, $y+n$, $n == 0$, $1+2$ *etc.*.

Possibly infinitely many argument (and return) values of function calls can be expressed via recursive mappings: consider an environment with mappings $\{n \mapsto 5,\ n \mapsto n-1\}$. In a high level sense, this environment is semantically equivalent to the infinite expansion $\{n \mapsto 5,\ n \mapsto 4,\ n \mapsto 3,\ n \mapsto 2,\ n \mapsto 1,\ n \mapsto 0,\ n \mapsto -1, \ldots\}$. However, recursive mappings in the environment are never expanded in *N*CI, but rather interpreted by a decision procedure that is a modular component of the system.

Since the environment can contain multiple bindings for the same variable, it is possible for the guard of a conditional branch to be both true and false, in which case both branches are interpreted in parallel and their resultant environments merged. For example, if $\{n \mapsto 0,\ n \mapsto 1\} \subseteq \mathbb{E}$ in Figure 1(b) then both branches would be executed.

It is essential to note that the environment grows monotonically during the analysis: mappings are only added and never removed, *i.e.* the environment *accumulates* near-value bindings. Furthermore, all values placed in the *N*CI environment are subexpressions in the program – for example, Let $x = 1 + 2$ is interpreted by simply adding $x \mapsto 1 + 2$ to the environment – and, all variables in the environment domain are declared statically in the program text. Thus, the number of possible environments for a given program is finite, so that monotonic environment growth is bounded during analysis. This implies that growth cannot continue forever but must eventually reach a fixed-point.

Pruning and re-running to a fixed-point Figure 2 shows the complete *N*CI for *sum* (5). The boxes with dashed edges are environments. A function call stack (not shown in the figure) is used to maintain the sequence of function invocations and hence detect any re-activations of the same function. At most one invocation of any function will appear on the *N*CI stack, so its size is tightly bounded. As mentioned above, the recursive call *sum* $(n-1)$ is interpreted by pruning: simply add the argument value and return variable to the environment. Note that $n-1$ is added as $n \mapsto n-1$ and not by reducing it to 4, also as discussed above. Thus the resulting environment includes $\{n \mapsto 5,\ n \mapsto n-1\}$. Observe that neither y nor r is bound to any definite value in the environment immediately after pruning the recursive call *sum* $(n-1)$ in the initial run (marked as "1" in the figure) – we say they are both *inchoate*, in a state of incomplete formedness. We allow this state of affairs to exist as a partial solution, to be completed by later analysis. Now, since the environment changes during the first run, the environment has not reached a fixed-point so we rerun the function body with this new environment. At this point the environment mappings $\{n \mapsto 5,\ n \mapsto n-1\}$ imply that $n \leq 5$, so sound interpretation of $n == 0$ in this environment yields both *true* and *false*, hence both branches of the conditional must be interpreted in parallel and their resulting environments merged, adding $r \mapsto 0$ to the environment. At the end of the second run a fixed-point has not yet been reached so the function body must be rerun again. In the third run, no new mappings are added so the environment stabilizes at a fixed-point, and the analysis terminates.

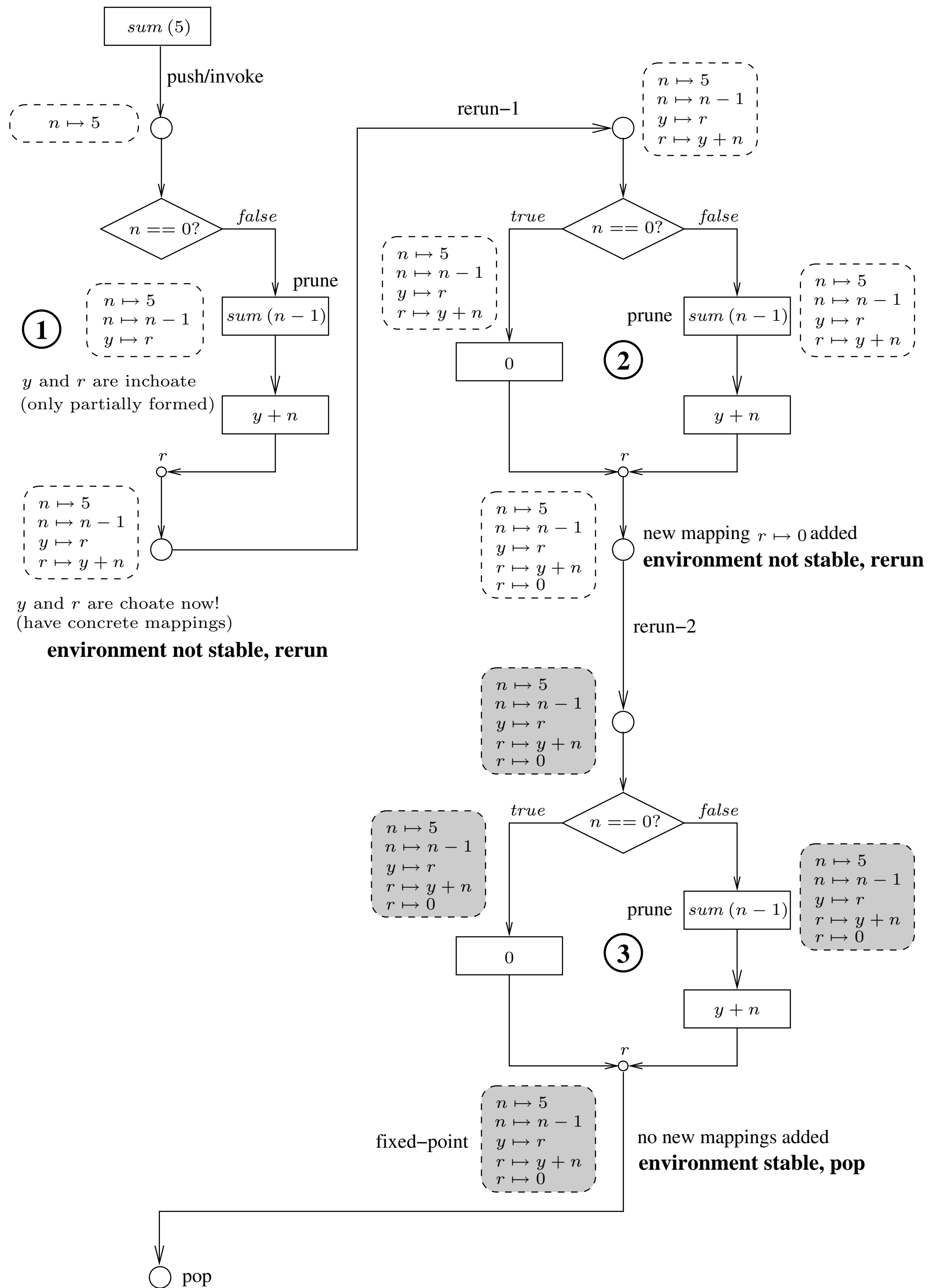

Fig. 2. Near-concrete interpretation of *sum* (5). Rerun-2 represents the fixed-point cycle as no new mappings are added throughout the rerun, so further reruns will not change the environment.

x, y, z, f			*variable*
i	$::=$	$\ldots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \ldots$	*integer*
b	$::=$	$true \mid false$	*boolean*
aop	$::=$	$+ \mid - \mid * \mid /$	*binary arithmetic operators*
rop	$::=$	$> \mid < \mid == \mid !=$	*binary relational operators*
ψ	$::=$	$i \mid b \mid \lambda_f x.\, e \mid \vartheta \; aop \; \vartheta \mid \vartheta \; rop \; \vartheta$	*near-concrete value*
ϑ	$::=$	$x \mid \psi$	*near-value*
κ	$::=$	$e \mid \vartheta \; \vartheta \mid \mathsf{If}\ \vartheta\ \mathsf{Then}\ e\ \mathsf{Else}\ e$	*atomic computation*
e	$::=$	$\vartheta \mid \mathsf{Let}\ x = \kappa\ \mathsf{In}\ e$	*A-normalized expression*
R	$::=$	$\bullet \mid \mathsf{Let}\ x = R\ \mathsf{In}\ e$	*reduction context*

Fig. 3. Core Language Syntax Grammar

The soundness of *N*CI is justified by this third, fixed-point, cycle (marked as "3" in the figure): rerun-2 faithfully simulates the entire concrete interpretation of *sum* (5), in that every node in its CI can be faithfully simulated by the corresponding node in rerun-2 of its *N*CI and some value in the stable environment. What is perhaps surprising about the algorithm is how computing ahead with inchoate results is not unsound: the computation steps are fundamentally being re-ordered, but since the environment will be "filled in" by further iterations over the function body, the final stable environment resolves all bindings and faithfully simulates concrete recursions. Inchoate variables bear some resemblance to futures [21]: computation can proceed ahead, even though some result values are missing.

Termination and Complexity As observed above, environments grow monotonically during the analysis, towards an upper bound. This means that the prune-rerun cycle for any given function will reach a fixed-point and terminate. Furthermore, the prune-rerun technique never recursively invokes a function—it is run to the end instead of pushing a recursive activation on the call stack.

The runtime complexity of *N*CI is exponential in the size of the program modulo the complexity of the decision procedure. We have a preliminary implementation that runs quickly on many examples; the worst-case examples involve higher-order programs that permute many functions in many orders. It is an open question if any realistic programs will exhibit this behavior in practice. For first-order programs that do not pass functions, we show the algorithm is polynomial.

3 Formalization

3.1 Language Model and Concrete Interpretation

Our core programming language model (Figure 3) is a pure higher-order functional language with arithmetic, booleans, conditional branching and let. $\lambda_f x.\, e$

arith-op
$$\frac{\vartheta_1 \dashrightarrow^{n_1} i_1 \quad \vartheta_2 \dashrightarrow^{n_2} i_2 \quad i_1\ aop\ i_2 = i_3}{(\vartheta_1\ aop\ \vartheta_2) \dashrightarrow i_3}$$

rel-op
$$\frac{\vartheta_1 \dashrightarrow^{n_1} i_1 \quad \vartheta_2 \dashrightarrow^{n_2} i_2 \quad i_1\ rop\ i_2 = b}{(\vartheta_1\ rop\ \vartheta_2) \dashrightarrow b}$$

reflex-val
$$\vartheta \dashrightarrow^{0} \vartheta$$

trans-val
$$\frac{\vartheta \dashrightarrow^{n-1} \vartheta' \quad \vartheta' \dashrightarrow \vartheta''}{\vartheta \dashrightarrow^{n} \vartheta''}$$

app
$$(\lambda_f x.\, e)\ \vartheta \rightarrowtail e\,[(\lambda_f x.\, e)/f]\,[\vartheta/x]$$

let
$$(\mathsf{Let}\ x = \vartheta\ \mathsf{In}\ e) \rightarrowtail e\,[\vartheta/x]$$

context
$$\frac{e \rightarrowtail e'}{R[e] \rightarrowtail R[e']}$$

if-true
$$\mathsf{If}\ \mathit{true}\ \mathsf{Then}\ e_1\ \mathsf{Else}\ e_2 \rightarrowtail e_1$$

if-false
$$\mathsf{If}\ \mathit{false}\ \mathsf{Then}\ e_1\ \mathsf{Else}\ e_2 \rightarrowtail e_2$$

reflex
$$e \rightarrowtail^{0} e$$

if
$$\frac{(\vartheta\ rop\ \vartheta') \dashrightarrow b}{\mathsf{If}\ (\vartheta\ rop\ \vartheta')\ \mathsf{Then}\ e_1\ \mathsf{Else}\ e_2 \rightarrowtail \mathsf{If}\ b\ \mathsf{Then}\ e_1\ \mathsf{Else}\ e_2}$$

trans
$$\frac{e \rightarrowtail^{n-1} e' \quad e' \rightarrowtail e''}{e \rightarrowtail^{n} e''}$$

Fig. 4. Concrete Interpretation (CI) Rules

is a function with f the name of the function for recursive calls; we will drop the subscript f when it is not used. Recursion can also be encoded via *e.g.* the Y-combinator. The grammar places expressions in an A-normal form, so each program point has an associated program variable. This grammar thus reflects an "internal" language which the original source program is translated into; since the differences between the original source and this A-normal form are minor we have left out the original source language. Program variables need not be unique, but to achieve a precise analysis it does help greatly to have unique variables and we will assume that is the case in informal discussions. Figure 4 gives a small step operational semantics or concrete interpretation (CI) rules for our core language, where $e[e'/x]$ denotes the capture avoiding substitution of all free occurrences of x in e with e'. The semantics is standard except that arithmetic ($\vartheta\ aop\ \vartheta$) and relational ($\vartheta\ rop\ \vartheta$) operations are evaluated lazily, hence the prefix "near-" in near-values. So for example the reduction of $(1 + 2)$ to 3 is postponed until it is essential to do so for the computation to proceed; in the meantime it is propagated as a near-concrete value $(1+2)$. This minor change is designed to align the concrete interpretation more closely with the near-concrete interpretation, making correctness proofs easier.

3.2 Near-Concrete Program Interpretation

Figure 5 gives the language syntax grammar extended with notation needed for the near-concrete program interpretation, *N*CI. $\mathbb{E}$ is the environment, a multi-mapping relation from variables to near-values. $\mathcal{S}$ is the function call stack. Each stack frame $[\mathcal{F}]_{\mathbb{E}}$ contains the function $\mathcal{F}$ and the (checkpointed) environment $\mathbb{E}$ at its point of invocation. $\varnothing$ is the empty stack. $\langle e \rangle$ is used to delimit function

$\mathbb{E}$	$=$	$\{x \mapsto \vartheta \mid x$ is a *variable* and ϑ is a *near-value*$\}$	*environment*
$\mathcal{F}$	$::=$	$\lambda_f x.\, e$	*function*
$\mathcal{S}$	$::=$	$\varnothing \mid \mathcal{S} : [\mathcal{F}]_{\mathbb{E}}$	*stack*
e	$::=$	$\ldots \mid \langle e \rangle$	*A-normalized expression (extended)*
R	$::=$	$\ldots \mid \langle R \rangle$	*reduction context (extended)*

Fig. 5. *N*CI: Language Syntax Grammar (Extended)

body e so that the end of its interpretation can be detected and appropriate action taken. Figure 6 gives the near-concrete interpretation rules.

We first define some notation used in the rules. $\overline{a_k}$ is short for $a_1, a_2, \ldots, a_k$ and $\{x \mapsto \overline{\vartheta_k}\}$ is short for $\{x \mapsto \vartheta_1, \ldots, x \mapsto \vartheta_k\}$. The instack relation $\mathcal{F} \in \mathcal{S}$ holds iff $\mathcal{S} = \mathcal{S}' : [\mathcal{F}']_{\mathbb{E}}$ and either $\mathcal{F} = \mathcal{F}'$ or $\mathcal{F} \in \mathcal{S}'$. The result of an expression is defined as the following relation: $\lfloor \vartheta \rfloor = \vartheta$ and $\lfloor \mathsf{Let}\ x = \kappa\ \mathsf{In}\ e \rfloor = \lfloor e \rfloor$. $\mathbb{E}^+$ is the transitive closure of $\mathbb{E}$: it is the least superset of $\mathbb{E}$ such that if $\{x \mapsto y, y \mapsto \vartheta\} \subseteq \mathbb{E}^+$ then $x \mapsto \vartheta \in \mathbb{E}^+$. The existential relation $\mathbb{E} \vDash x \rightsquigarrow \psi$ holds iff $x \mapsto \psi \in \mathbb{E}^+$, while the corresponding exhaustive relation is defined as $\mathbb{E} \vDash x \Rrightarrow \{\overline{\psi_k}\}$ iff $\forall \psi.\, \mathbb{E} \vDash x \rightsquigarrow \psi \iff \psi \in \{\overline{\psi_k}\}$. Also, $\mathbb{E} \vDash \psi \Rrightarrow \{\psi\}$ makes the relation total on near-values. We say "ϑ is inchoate in $\mathbb{E}$" iff $\mathbb{E} \vDash \vartheta \Rrightarrow \{\}$. We say "$\vartheta$ is integral in $\mathbb{E}$", iff $\mathbb{E} \vDash \vartheta \Rrightarrow \{\overline{\psi_k}\}$, $k \geq 1$ and for all $1 \leq i \leq k$, either ψ_i is an integer, or $\psi_i = \vartheta_i\ aop_i\ \vartheta'_i$, for some ϑ_i, aop_i and ϑ'_i.

We now give a brief overview of key rules. Recall the overview (Section 2.1) for intuitions on the bigger picture of the algorithm. The arith-op (cond-op) rule ensures that ϑ and ϑ' are either integral or inchoate in $\mathbb{E}$, and adds the arithmetic (relational) operation, as is without execution, to the environment. The app-push–nonrec-call only applies when the function $\mathcal{F}$ is not already on the call stack *i.e.* not a re-activation. The app-prune–rec-call rule applies only if the application is a re-activation of $\mathcal{F}$, in which case it is pruned by adding the argument to $\mathbb{E}$ and returning the result ϑ_r of $\mathcal{F}$'s body. If ϑ_r not a near-concrete value, it will be inchoate in the first pass since the function has yet to accumulate any results. Note that recursive function invocations encoded via fixed-point combinators (like the Y-combinator) or self-passing will also be detected by app-prune–rec-call since they also lead to re-activation; so will mutual recursion. If the function being applied is itself not yet concrete, app-inchoate pushes the *N*CI past such a call-site without doing anything. A redex $\langle \vartheta \rangle$ indicates that a function invocation has completed its interpretation, and it either needs to rerun (app-rerun–not-fixed-point) if the environment fixed-point has not yet been reached, or popped (app-pop–fixed-point) if there is a fixed-point. If there are multiple functions flowing into a call site, app-merge interprets each of their applications in parallel and merges their resultant environments and return values.

The if-merge rule uses a decision procedure to compute the possible truth values of the condition ϑ given $\mathbb{E}$. If ϑ can result in both *true* and *false*, both of the branches are interpreted in parallel and their results are merged. We define no concrete decision procedure here; the general problem is undecidable since

$$\textsf{let}\quad (\mathcal{S},\,\mathbb{E},\,\textsf{Let}\ x = \vartheta\ \textsf{In}\ e) \longrightarrow (\mathcal{S},\,\mathbb{E}\cup\{x\mapsto\vartheta\},\,e)$$

$$\textsf{arith-op}\quad \frac{\vartheta \text{ and } \vartheta' \text{ are either integral or inchoate in } \mathbb{E}}{(\mathcal{S},\,\mathbb{E},\,\textsf{Let}\ x = (\vartheta\ aop\ \vartheta')\ \textsf{In}\ e) \longrightarrow (\mathcal{S},\,\mathbb{E}\cup\{x\mapsto(\vartheta\ aop\ \vartheta')\},\,e)}$$

$$\textsf{cond-op}\quad \frac{\vartheta \text{ and } \vartheta' \text{ are either integral or inchoate in } \mathbb{E}}{(\mathcal{S},\,\mathbb{E},\,\textsf{Let}\ x = (\vartheta\ rop\ \vartheta')\ \textsf{In}\ e) \longrightarrow (\mathcal{S},\,\mathbb{E}\cup\{x\mapsto(\vartheta\ rop\ \vartheta')\},\,e)}$$

$$\textsf{app-push-nonrec-call}\quad \frac{\mathcal{F} = \lambda_f x.\,e \quad \mathcal{F}\notin\mathcal{S}}{(\mathcal{S},\,\mathbb{E},\,\mathcal{F}\ \vartheta) \longrightarrow (\mathcal{S} : [\mathcal{F}]_{\mathbb{E}},\,\mathbb{E}\cup\{f\mapsto\mathcal{F},\,x\mapsto\vartheta\},\,\langle e\rangle)}$$

$$\textsf{app-prune-rec-call}\quad \frac{\mathcal{F} = \lambda_f x.\,e \quad \mathcal{F}\in\mathcal{S} \quad \lfloor e\rfloor = \vartheta_r}{(\mathcal{S},\,\mathbb{E},\,\mathcal{F}\ \vartheta) \longrightarrow (\mathcal{S},\,\mathbb{E}\cup\{x\mapsto\vartheta\},\,\vartheta_r)}$$

$$\textsf{app-inchoate}\quad \frac{f \text{ is inchoate in } \mathbb{E}}{(\mathcal{S},\,\mathbb{E},\,\textsf{Let}\ x = f\ \vartheta\ \textsf{In}\ e) \longrightarrow (\mathcal{S},\,\mathbb{E},\,e)}$$

$$\textsf{app-rerun-not-fixed-point}\quad \frac{\mathcal{F} = \lambda_f x.\,e \quad \lfloor e\rfloor = \vartheta_r \quad \mathbb{E}\neq\mathbb{E}^\natural}{\left(\mathcal{S} : [\mathcal{F}]_{\mathbb{E}^\natural},\,\mathbb{E},\,\langle\vartheta_r\rangle\right) \longrightarrow \left(\mathcal{S} : [\mathcal{F}]_{\mathbb{E}},\,\mathbb{E},\,\langle e\rangle\right)}$$

$$\textsf{app-pop-fixed-point}\quad \frac{\mathcal{F} = \lambda_f x.\,e \quad \lfloor e\rfloor = \vartheta_r}{\left(\mathcal{S} : [\mathcal{F}]_{\mathbb{E}},\,\mathbb{E},\,\langle\vartheta_r\rangle\right) \longrightarrow \left(\mathcal{S},\,\mathbb{E},\,\vartheta_r\right)}$$

$$\textsf{app-merge}\quad \frac{\begin{array}{l}\mathbb{E}\vDash f \Rrightarrow \{\overline{\mathcal{F}_k}\} \quad k\geq 1\\ \forall 1\leq i\leq k.\ (\mathcal{S},\,\mathbb{E},\,\mathcal{F}_i\ \vartheta) \longrightarrow^{n_i} (\mathcal{S},\,\mathbb{E}_i,\,\vartheta_i)\end{array}}{(\mathcal{S},\,\mathbb{E},\,\textsf{Let}\ x = f\ \vartheta\ \textsf{In}\ e) \longrightarrow \left(\mathcal{S},\,\bigcup_{1\leq i\leq k}\mathbb{E}_i\cup\{x\mapsto\vartheta_i\},\,e\right)}$$

$$\textsf{reflex}\quad (\mathcal{S},\,\mathbb{E},\,\vartheta) \longrightarrow^0 (\mathcal{S},\,\mathbb{E},\,\vartheta)$$

$$\textsf{if-true}\quad (\mathcal{S},\,\mathbb{E},\,\textsf{If}\ true\ \textsf{Then}\ e_1\ \textsf{Else}\ e_2) \longrightarrow (\mathcal{S},\,\mathbb{E},\,e_1)$$

$$\textsf{context}\quad \frac{(\mathcal{S},\,\mathbb{E},\,\kappa) \longrightarrow (\mathcal{S}',\,\mathbb{E}',\,\kappa')}{(\mathcal{S},\,\mathbb{E},\,R[\kappa]) \longrightarrow (\mathcal{S}',\,\mathbb{E}',\,R[\kappa'])}$$

$$\textsf{if-false}\quad (\mathcal{S},\,\mathbb{E},\,\textsf{If}\ false\ \textsf{Then}\ e_1\ \textsf{Else}\ e_2) \longrightarrow (\mathcal{S},\,\mathbb{E},\,e_2)$$

$$\textsf{trans}\quad \frac{\begin{array}{l}(\mathcal{S},\,\mathbb{E},\,e) \longrightarrow^{n-1} (\mathcal{S}',\,\mathbb{E}',\,e')\\ (\mathcal{S}',\,\mathbb{E}',\,e') \longrightarrow (\mathcal{S}'',\,\mathbb{E}'',\,e'')\end{array}}{(\mathcal{S},\,\mathbb{E},\,e) \longrightarrow^{n} (\mathcal{S}'',\,\mathbb{E}'',\,e'')}$$

$$\textsf{if-merge}\quad \frac{\begin{array}{l}\mathbb{E}\vDash_b \vartheta \Rrightarrow \{\overline{b_k}\} \quad 1\leq k\leq 2\\ \forall 1\leq i\leq k.\ (\mathcal{S},\,\mathbb{E},\,\textsf{If}\ b_i\ \textsf{Then}\ e_1\ \textsf{Else}\ e_2) \longrightarrow^{n_i} (\mathcal{S},\,\mathbb{E}_i,\,\vartheta_i)\end{array}}{\left(\mathcal{S},\,\mathbb{E},\,\textsf{Let}\ z = (\textsf{If}\ \vartheta\ \textsf{Then}\ e_1\ \textsf{Else}\ e_2)\ \textsf{In}\ e\right) \longrightarrow \left(\mathcal{S},\,\bigcup_{1\leq i\leq k}\mathbb{E}_i\cup\{z\mapsto\vartheta_i\},\,e\right)}$$

Fig. 6. Near-Concrete Interpretation (*N*CI) Rules

arbitrary Diophantine equations can be expressed, but in practice most problems are easily decidable. One could always use a conservative decision procedure which returns $\{true, false\}$ independent of the condition ϑ; or, one could employ a sophisticated decision procedure for high precision. This decision procedure is expressed as $\mathbb{E} \vDash_b \vartheta \Rrightarrow \{\overline{b_k}\}$, meaning that under $\mathbb{E}$, ϑ conservatively has boolean value(s) $\{\overline{b_k}\}$; the requirements of a sound decision procedure are given in Appendix A.

Definition 1 (Near-Concrete Interpretation (*N*CI)). *We say "e has a near-concrete interpretation" iff* $(\varnothing,\, \emptyset,\, e) \longrightarrow^n (\varnothing,\, \mathbb{E},\, \vartheta)$*, for some* n*,* $\mathbb{E}$*, and* ϑ*.*

The derivation tree of $(\varnothing,\, \emptyset,\, e) \longrightarrow^n (\varnothing,\, \mathbb{E},\, \vartheta)$ can be viewed as a finite state transition system or a finite automaton representing the *N*CI of e, as illustrated in Figure 2. It can also be viewed as a trace-based abstract interpretation of e.

3.3 Properties of Near-Concrete Interpretation

Proofs or proof-sketches of all ensuing lemmas are in Appendix B.

Lemma 1 (Termination of *N*CI). *NCI terminates on all input.*

We next prove the soundness of *N*CI by showing that it is a faithful simulation of CI for any e, modulo the soundness of the decision procedure employed. The *N*CI semantics in Figure 6 is a hybrid between small- and big-step semantics. The big-step semantics for *N*CI can defined as follows:

Definition 2 (Big-Step *N*CI). $(\mathcal{S},\, \mathbb{E},\, e) \Longrightarrow (\mathcal{S}',\, \mathbb{E}',\, e')$ *iff either*

1. *(trans).* $(\mathcal{S},\, \mathbb{E},\, e) \longrightarrow^n (\mathcal{S}',\, \mathbb{E}',\, e')$*, for some* n*, or*
2. *(app-merge).* $e = (\mathsf{Let}\ x = f\ \vartheta\ \mathsf{In}\ e_{next})$ *and* $e' = (\mathsf{Let}\ x = e''\ \mathsf{In}\ e_{next})$*, where* $\mathbb{E} \vDash f \leadsto \mathcal{F}$ *and* $(\mathcal{S},\, \mathbb{E},\, \mathcal{F}\ \vartheta) \Longrightarrow (\mathcal{S}',\, \mathbb{E}',\, e'')$*, or*
3. *(if-merge).* $e = \big(\mathsf{Let}\ y = (\mathsf{If}\ \vartheta\ \mathsf{Then}\ e_1\ \mathsf{Else}\ e_2)\ \mathsf{In}\ e_{next}\big)$ *and* $e' = (\mathsf{Let}\ y = e''\ \mathsf{In}\ e_{next})$*, where* $\mathbb{E} \vDash_b \vartheta \Rrightarrow \{\ldots,\, b_i,\, \ldots\}$ *and* $(\mathcal{S},\, \mathbb{E},\, \mathsf{If}\ b_i\ \mathsf{Then}\ e_1\ \mathsf{Else}\ e_2) \Longrightarrow (\mathcal{S}',\, \mathbb{E}',\, e'')$*.*

We can then define the simulation relation which relates an expression e with another expression e' given its environment *i.e.* $e'\backslash\mathbb{E}'$.

Definition 3 (Simulation Relation: $\preccurlyeq$). $e \preccurlyeq e'\backslash\mathbb{E}'$ *iff* $e'\backslash\mathbb{E}' \hookrightarrow^* e''$*, for some* e''*, such that either* $e = e''$*, or* $e = R[e_{body}]$ *and* $e'' = R''[\langle e_{body}\rangle]$*, for some* R*,* R'' *and* e_{body}*.*

where, $e\backslash\mathbb{E} \hookrightarrow^* e'$ is the nondeterministic recursive inlining of mappings from $\mathbb{E}$ into e such that either $e = e'$, or $x \in \mathsf{free}(e)$, $x \mapsto \vartheta \in \mathbb{E}$ and $e[\vartheta/x]\backslash\mathbb{E} \hookrightarrow^* e'$. It essentially relates the redex's corresponding to the most recent function invocations, if any. Note that e corresponds to CI and hence does not contain delimiters '$\langle\rangle$'.

Lemma 2 (Simulation of CI by *N*CI). *If e has a near-concrete interpretation and* $e \rightarrowtail^n e_n$ *then* $(\varnothing,\, \emptyset,\, e) \Longrightarrow (\mathcal{S}'_n,\, \mathbb{E}'_n,\, e'_n)$ *such that* $e_n \preccurlyeq e'_n\backslash\mathbb{E}'_n$*.*

We say "CI of e is stuck" iff it is not a near-value and $\neg\exists e'.\ e \rightarrowtail e'$.

Lemma 3 (Soundness of *N*CI). *If a closed e has a NCI then its CI either computes to a value or computes forever.*

Lemma 4 (Complexity of *N*CI). *The runtime complexity of NCI is exponential in size of the program, modulo the complexity of the decision procedure.*

The above theoretical complexity result reflects the worst case scenario for *N*CI, and it is doubtful whether this order of complexity would be encountered in practice. The following result shows how the higher-order nature is responsible for the exponential bound.

Lemma 5 (Complexity of *N*CI for First-order Programs). *The runtime complexity of NCI is polynomial in size of first-order programs, modulo the complexity of decision procedure employed.*

Note that the non-elementary bound on simply typed λ-reduction [18] does not apply to *N*CI since there can be multiple simultaneous activations of the same function in the simply typed λ-calculus, as in the following example, where f_a is re-activated inside f_b:

$$\begin{array}{l} \mathsf{Let}\ f_a = \lambda b : \mathrm{int} \to \mathrm{int}.\, b\,(5) : \mathrm{int}\ \ \mathsf{In} \\ \quad \mathsf{Let}\ f_b = \lambda z : \mathrm{int}.\, f_a\,(\lambda x : \mathrm{int}.\, x : \mathrm{int}) : \mathrm{int}\ \ \mathsf{In}\ f_a\ (f_b) \end{array}$$

4 Extensions to *N*CI

A treatment of mutation and state is necessary for our analysis to apply to realistic programs, so we first extend *N*CI to incorporate a heap. We also extend *N*CI with context and path sensitivity, to obtain flexibility in practical settings. Due to the lack of space we develop them only via examples in the paper body; formal definitions are found in the Appendix. The extensions are conceptually orthogonal and can be combined to be used in conjunction.

4.1 Mutable State via an Abstract Heap

We incorporate mutable state by maintaining an abstract heap $\mathbb{H}$ in addition to the environment and call stack, yielding the system $N\mathrm{CI}^{\mathbb{H}}$. The abstract heap is a multi-mapping relation from *locations* to near-values. The prune-rerun strategy for recursive computations is extended to stabilize the abstract heap as well as the environment. Heap operations such as *ref*, *get* and *set* closely mimic that of the CI heap, except during reruns of recursive functions where abstract heap allocations are consistently reused. Consider the following stateful variation of $\mathit{sum}\,(n)$ employing memory based fixed-points (*a.k.a.* tying the knot):

$$\begin{array}{l} \mathsf{Let}\ \mathit{sum} = \mathsf{Ref}\ 0\ \mathsf{In} \\ \quad \mathit{sum} := \lambda n.\, \mathsf{If}\ (n == 0)\ \mathsf{Then}\ 0 \\ \qquad\qquad\quad \mathsf{Else}\ \mathsf{Let}\ \mathit{ptr} = \mathsf{Ref}\ (n-1)\ \mathsf{In}\ n + (!\mathit{sum})\,(!\mathit{ptr}); \\ \quad (!\mathit{sum})\,(5) \end{array}$$

The ';' is a syntactic sugar for sequencing operation. The second assignment to *sum* overwrites the previous assignment of 0, say at location *loc*, so that the heap contains $\mathit{loc} \mapsto (\lambda n.\mathsf{If} \ldots (!\mathit{sum})\,(!\mathit{ptr}))$ and the environment contains $\mathit{sum} \mapsto \mathit{loc}$. Since the abstract heap is not cumulative like the environment, the subsequent use of *sum* as a function reference is ensued by the interpretation. During the first run of $(!\mathit{sum})\,(5)$, the expression "$\mathit{ptr} = \mathsf{Ref}\ (n-1)$" results

in $n-1$ being stored in a fresh location loc' in the heap, *i.e.* $loc' \mapsto n-1$, and with $ptr \mapsto loc'$ being added to the environment. The same location loc' is then *reused* during reruns of the function body for interpretation of the same expression "$ptr = \mathsf{Ref}\ (n-1)$".

The notion of an abstract representation of the heap is found in various program analyses, including [26, 14, 6], but to our knowledge our system is the first flow-sensitive analysis to model cyclic pointer graph structures on the heap, an important step in accurately modeling recursive datatypes and complex object reference graphs. The above example has a cyclic heap structure during $\mathit{N}\mathrm{CI}^{\mathbb{H}}$: the location loc, which contains the $\lambda n.(\mathsf{If}\ldots)$ function itself contains variable sum which in turn points to loc, hence a cycle. Appendix C provides the formal details, including an extension of the argument for termination in the presence of non-monotonic heap growth during analysis.

4.2 Context Sensitivity via Argument Tagging

Core *N*CI does not distinguish applications of the same function in different contexts, so for example the expression:

$$(\lambda id.\, id\,(5) + 1;\ id\,(true) \wedge false)\ (\lambda x.\, x)$$

does not have an *N*CI: $id\,(true)$ is interpreted as both *true* and the previously accumulated *id* application result 5, hence the conjunction $\ldots \wedge false$ fails. The monotonically increasing environment is necessary to achieve fixed-points of recursive functions, but it limits expressiveness in such cases. This limitation can be surmounted by adding context sensitivity, the parametric polymorphism equivalent of type systems.

We incorporate context sensitivity in an extension $\mathit{N}\mathrm{CI}^{\psi}$, achieved by tagging function bodies with their concrete argument prior to invocation. So, $\lambda x.\, x$ would be tagged as $\lambda x^5.\, x^5$ before the first invocation and as $\lambda x^{true}.\, x^{true}$ prior to the second, resulting in the final environment containing $\{x^5 \mapsto 5,\ x^{true} \mapsto true\}$. This environment distinguishes the different calling contexts of *id* so the above example has an $\mathit{N}\mathrm{CI}^{\psi}$ interpretation. Since the tags are limited to subexpressions of the original program, at most polynomial-time complexity is added.

$\mathit{N}\mathrm{CI}^{\psi}$ is in the spirit of the Cartesian Product Algorithm (CPA) [1]: a different instantiation of the analysis is made for each different combination of function and argument reaching a call site. CPA's multiple-argument functions correspond to curried functions in our pure functional language, so consider how our algorithm behaves on a curried function:

$$\mathsf{Let}\ f = \lambda x.\, \lambda y.\, x + y\ \ \mathsf{In}\ \ f\ 0\ 5;\ f\ 1\ 8$$

The first application tags the outer function as $\lambda x^0.\, \lambda y^0.\, x^0 + y^0$ while the second one tags it as $\lambda x^1.\, \lambda y^1.\, x^1 + y^1$. Their subsequent applications tag the inner functions as $\lambda y^{0^5}.\, x^0 + y^{0^5}$ and $\lambda y^{1^8}.\, x^1 + y^{1^8}$ respectively. Hence the application of the same inner function $\lambda y.\, x + y$ in different calling contexts will be analyzed separately. Note that the stacked tags here are based on the *lexical* scoping in

the original program and not the dynamic scoping of functions, so the height of the stacked tags is strongly bounded. Appendix D provides formal details.

4.3 Path-Sensitivity via Path Tagging

In cases when a branching condition is either unknown or resolves to both *true* and *false*, *N*CI interprets both the branches in parallel and merges their resultant environments. This approximation reflects a common approach in static program analyses. However, given an expression e of the form $\mathsf{If}\ x\ \mathsf{Then}\ e_1\ \mathsf{Else}\ e_2$, although x may be either *true* or *false*, within e_1 it *must* be *true*, and within e_2 it *must* be false. Recent work on *path sensitivity* [11, 13] allows program interpretations to take advantage of this property, since, though it may remain unknown whether branch e_1 or e_2 will be taken upon evaluation of e, greater precision can be obtained from the knowledge that the e_1 branch is taken iff x is *true*, and the e_2 branch is taken iff x is *false*. We extend *N*CI with *path tagging* to obtain path sensitivity in our analysis, yielding the system $N\mathrm{CI}^{\mathsf{p}}$. For example, consider the following expression, where *dump* is user input and unknown statically:

$$\mathsf{Let}\ x = (\mathsf{If}\ \ dump\ \mathsf{Then}\ 1\ \mathsf{Else}\ true)\ \mathsf{In}\ (\mathsf{If}\ \ dump\ \mathsf{Then}\ x+2\ \mathsf{Else}\ x \wedge false)$$

The core *N*CI analysis gets stuck at $x + 2$ since the environment contains both $x \mapsto 1$ and $x \mapsto true$ due to merging of the environments from the prior branches. It is unable to retain the value-to-branch correlation information which is based on the branching condition: if $dump = true$ then $x = 1$ at $x + 2$, and if $dump = false$ then $x = true$ at $x \wedge false$. Path tagging in $N\mathrm{CI}^{\mathsf{p}}$ will tag all near-values in the environment with the conditions in force when they were added to the environment, by recording the condition variable and value (true or false) along with the near-value itself. In the example, at the end of the first conditional the environment contains $\{x \mapsto 1^{\{dump^{true}\}}, x \mapsto true^{\{dump^{false}\}}\}$. The superscript $\{dump^{true}\}$ on the mapping of x to 1 indicates that this mapping was added in a branch of a conditional where the conditional variable was *dump* and its value was assumed to be *true*, since the then-case was being interpreted. The mapping of x to *true* is superscripted similarly. Now, when the then-case of the second conditional in the program is being analyzed, the path-tag is set to $\{dump^{true}\}$, meaning that near-values conflicting with this path-tag, that is, tagged with $dump^{false}$, should *not* be considered as having occurred. Since $x \mapsto true^{\{dump^{false}\}}$ conflicts with this path-tag, only $x \mapsto 1^{\{dump^{true}\}}$ is considered when analyzing $x + 2$; correspondingly, the path-tag is set to $\{dump^{false}\}$ when analyzing the else-branch and hence only $x \mapsto true^{\{dump^{false}\}}$ is considered for $x \wedge false$. Note that path-tags are represented as sets, and hence can denote multiple branches. Also, if $N\mathrm{CI}^{\mathsf{p}}$is combined with $N\mathrm{CI}^{\mathbb{H}}$ the mappings in the heaps are tagged as well. Appendix E provides the formal details.

Path sensitivity has been incorporated in several practical program verification systems to obtain more precise results [11, 13]. We relate our system to this work in the following section where we look at some particular applications.

5 Applications

Recall that *N*CI generates a finite automaton that closely simulates the program computation. Hence many static program analyses can be performed using *N*CI, to enforce both local and global properties of programs. The unique precision of the analysis is the combination of fully higher-order functions, mutable heap tracking, and flow- and path-sensitivity, and program analysis problems in this space will benefit the most from the algorithm. We now discuss enforcement of temporal safety properties and information flow analysis as two example applications of our program analysis, although we envision more general applications of the system.

5.1 Enforcement of Temporal Safety Properties

Static enforcement of temporal safety properties has been an area of active research interest [13, 25, 5], especially in languages with side effects where order of operations matters. A canonical example of a temporal program property is file management: files have to be opened before they are read or written to, and only previously opened files can be closed. Imagine an extension of our language model with dynamically generated file handles and primitive operations $_open$, $_close$, and $_read$ on file handles. Imagine also the inclusion of records, which are collections of labeled fields $\{l_1 = e_1; \ldots; l_n = e_n\}$, and assume a fail value that signals termination. A function *assert* defined as $\lambda x.\mathsf{If}\ x\ \mathsf{Then}\ 1\ \mathsf{Else}\ \mathsf{fail}$ will then block computation if some given boolean condition does not hold.

Given macros $\mathsf{open} \triangleq 1$ and $\mathsf{close} \triangleq 0$, we can model files as records with a file handle field labeled *fptr*, and an integer reference field *flag* referencing open or close. A safe higher-level file interface can be provided via *open*, *close*, and *read* functions that manipulate and check file flags appropriately, on the basis of blocking assertions *isopen* and *isclosed*:

$$\begin{aligned}
\mathit{isopen} &\triangleq \lambda x.\mathit{assert}(!(x.\mathit{flag}) == \mathsf{open}) \\
\mathit{isclosed} &\triangleq \lambda x.\mathit{assert}(!(x.\mathit{flag}) == \mathsf{close}) \\
\mathit{open} &\triangleq \lambda x.\mathit{isclosed}(x); _\mathit{open}(x.\mathit{fptr}); x.\mathit{flag} := \mathsf{open} \\
\mathit{close} &\triangleq \lambda x.\mathit{isopen}(x); _\mathit{close}(x.\mathit{fptr}); x.\mathit{flag} := \mathsf{close} \\
\mathit{read} &\triangleq \lambda x.\mathit{isopen}(x); _\mathit{read}(x.\mathit{fptr})
\end{aligned}$$

Consider then the code sequence $\mathit{open}(f); \mathit{process}(f); \mathit{read}(f)$, where f is some given closed file containing *flag* value p_{flag} and *process* interacts with f. Suppose on one hand that *process* is statically known to not invoke *close* on f, then the final *read* in the sequence above will succeed in $N\mathrm{CI}^{\mathbb{H}}$ if *open* succeeds since the abstract heap will contain only the mapping $\{p_{\mathit{flag}} \mapsto \mathsf{open}\}$ at that point in the analysis, having been added by *open*. On the other hand, if *process* may or may not invoke *close* based on some inherently dynamic condition, then the abstract heap at the point of *read* in the $N\mathrm{CI}^{\mathbb{H}}$ analysis will contain the multimapping $\{p_{\mathit{flag}} \mapsto \mathsf{open}, p_{\mathit{flag}} \mapsto \mathsf{closed}\}$, so the analysis will fail at that point.

$l := h;$ $l := 0;$	$\mathsf{Let}\ h' = h > 0\ \mathsf{In}$ $\quad \mathsf{Let}\ l = \mathsf{If}\ (h')\ \mathsf{Then}\ 1\ \mathsf{Else}\ 0$ $\quad \mathsf{In}\ l$	$\mathsf{Let}\ h' = h > 0\ \mathsf{In}$ $\quad \mathsf{Let}\ l = \mathsf{If}\ (h')\ \mathsf{Then}\ 1\ \mathsf{Else}\ 1$ $\quad \mathsf{In}\ l$
(a)	(b)	(c)

Fig. 7. Examples for Information Flow Analysis

Path-sensitivity We now show how the combination of $N\mathrm{CI}^{\mathfrak{p}}$ and $N\mathrm{CI}^{\mathbb{H}}$ can be used to obtain a highly precise path-sensitive analysis of temporal program properties. Consider the following example adapted from [11], where f is some given closed file containing *flag* value p_{flag} and *dump* is user input and unknown statically:

$$\begin{array}{l}\mathsf{Let}\ switch = \mathsf{If}\ dump\ \mathsf{Then}\ true\ \mathsf{Else}\ false\ \mathsf{In} \\ \quad (\mathsf{If}\ dump\ \mathsf{Then}\ open(f)\ \mathsf{Else}\ ()); (\mathsf{If}\ switch\ \mathsf{Then}\ close(f)\ \mathsf{Else}\ ())\end{array}$$

The path-sensitive analysis in [11] is not strong enough to detect that *switch* is a copy of *dump*, but $N\mathrm{CI}^{\mathfrak{p}}$ detects the correlation between *dump* and *switch* and thus the safety of the *close* in the last part of the expression. Since *dump* is statically unknown, the analysis interprets both branches of all the conditionals in parallel and tags and merges their environments and abstract heaps. After the first conditional is analyzed, $\left\{switch \mapsto true^{\{dump^{true}\}},\ switch \mapsto false^{\{dump^{false}\}}\right\}$ is part of the environment. Similarly $open(f)$ adds $p_{flag} \mapsto \mathsf{open}^{\{dump^{true}\}}$ to the abstract heap, while the other branch in the conditional results in $p_{flag} \mapsto \mathsf{close}^{\{dump^{false}\}}$. Now, even though p_{flag} has two possible values in the heap, they are tagged with the conditions under which they hold. Now the path tag during interpretation of $close(f)$ will be $\{dump^{true},\ switch^{true}\}$, restricting the possible mappings to only those whose range is *not* tagged with $dump^{false}$ or $switch^{false}$. Hence, only $p_{flag} \mapsto \mathsf{open}^{\{dump^{true}\}}$ is considered when analyzing $close(f)$, since the mapping $p_{flag} \mapsto \mathsf{close}^{\{dump^{false}\}}$ conflicts with the current path tag. We observe that all examples in [11] can be successfully analyzed using $N\mathrm{CI}^{\mathfrak{p}}$.

5.2 Information Flow Analysis

Information flow analysis is an important technique for discovering leaks of secure data in software [12, 24, 19]. Since NCI maintains a trace of all the data flow in the program, any direct flow of high security data into low security locations will be present in the transitive closure $\mathbb{E}^+$ of the final environment $\mathbb{E}$. To perform an information flow analysis on a program using NCI, assume that a mapping of all program variables to $\{\mathsf{high}, \mathsf{low}\}$ has been defined. We illustrate information flow via the examples in Figure 7, where we assume l is low and h, h' are high. Consider the stateful program in Figure 7(a). Since heap assignment is flow-sensitive in $N\mathrm{CI}^{\mathbb{H}}$, the final heap only contains $\{loc \mapsto 0\}$ where

the environment is $\{l \mapsto loc\}$; thus, h is not transitively reachable from l and so no high data is reachable from a low variable and there is no direct information leak. Flow-insensitive information flow type systems such as [24, 19] would not consider the order of assignments to l, and would erroneously report a leakage here.

One aspect not covered in the previous example is *indirect* information flow: an assignment to a low variable under the guard of a high variable condition is indirectly gaining path information and thus information about the high data. Our path-sensitive $N\mathrm{CI}^{\mathfrak{p}}$ has this path information recorded in it already, and so indirect leaks may also be read directly from the environment, by transitively closing the environment as before but also closing through path-tag variables. Consider the example in Figure 7(b). $N\mathrm{CI}^{\mathfrak{p}}$ will have $\{h' \mapsto (h > 0), l \mapsto 1^{\{h'^{true}\}}, l \mapsto 0^{\{h'^{false}\}}\}$ in the final environment. Note that the mapping $l \mapsto 1^{\{h'^{true}\}}$ implies an indirect data dependence between l and h', in that $l = 1$ only when $h' = true$. Transitively following this indirect dependence leads to a dependence on $h' \mapsto (h > 0)$, implying the value of l is dependent on that of h; hence an indirect information leak has occurred. Interestingly, if l had been set to 1 in both branches of the conditional as in Figure 7(c), the path dependencies would be cancelled out since l ends up with the same value in all paths *i.e.* $l \mapsto 1^{\{\}}$ since $\{h'^{true}, h'^{false}\}$ cancels out to $\{\}$, and the analysis would conclude correctly that no information leakage had occurred. Existing information flow analyses are too weak to directly admit such subtleties. There has been recent work on flow-sensitive information flow analysis [15] which will admit 7(a), but this work does not incorporate path-sensitivity and so would fail on 7(c). One recent paper [3] will succeed on 7(c) if the programmer had manually inserted an assertion; our system can do it automatically. Another recent work [16] deals with only simple while-programs and does not incorporate path-sensitivity.

6 Conclusions

In a broad sense, NCI is an abstract interpretation [8, 9], being essentially an abstraction of interpreter computation. Our program setting is complicated by the presence of higher-order functions, and heap structures that may be recursive. Our system also differs from abstract interpretations in our rule-based approach, reflecting the ancestry of this work as a type closure computation [2, 23]: it arose from a type constraint and effect system where the effects were type constraint sets. It is most closely related to the school of trace-based abstract interpretations [22, 7], where an abstraction of the whole computation tree is produced by the analysis. An advantage of NCI is how well it deals with higher-order programs. Trace-based abstract interpretations are most naturally applicable to imperative, first order programs. There has been some work on higher-order abstract interpretations [10, 4, 20, 17] but these systems do not produce trace interpretations, and so are not as precise as our system. An OCaml implementation combining NCI, $N\mathrm{CI}^{\mathbb{H}}$ and $N\mathrm{CI}^{\psi}$ has been implemented and is available at `http://www.cs.jhu.edu/~pari/nci/`; $N\mathrm{CI}^{\mathfrak{p}}$ is being implemented.

6.1 Future Work

This paper is of a foundational nature, focusing on principles rather than applications. The long-term goal however is the development of a system that scales up to real applications. In Section 5 we outlined some applications to show its potential applicability.

Our analysis is potentially useful as a program analysis for compilers, but the analysis is more suited to applications that have a greater need for precision and can tolerate somewhat longer running times. We believe the analysis is highly suited to automated verifications of higher-order programs, along the lines of first-order systems such as ESP [13], ARCHER [25], and SLAM [5]. A particular advantage of our approach in this realm is the way in which higher-order functions are treated, and the relevant principles should also enable modeling of higher-order aspects of object-oriented languages, particularly dynamic dispatch.

One other area of future work is the incorporation of more explicit inductive reasoning about programs into the analysis. We are already performing primitive inductive reasoning about integers, as in the $sum(5)$ example in Section 2: the analysis could infer that $n \leq 5$. With stronger inductive inference we could have reached the stronger conclusion that $0 \leq n \leq 5$. Stronger inductive properties will enable more subtle program properties such as array bounds checks, to be automatically verified.

References

1. O. Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *ECOOP'95: Proceedings of the 9th European Conference on Object-Oriented Programming*, 1995.
2. A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *FPCA'93: Proceedings of the conference on Functional Programming Languages and Computer Architecture*, 1993.
3. T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. *SIGPLAN Not.*, 41(1):91–102, 2006.
4. J. M. Ashley. A practical and flexible flow analysis for higher-order languages. In *POPL'96: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1996.
5. T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model Checking of Software*, 2001.
6. B.-Y. E. Chang and K. R. M. Leino. Abstract interpretation with alien expressions and heap structures. In *VMCAI*, 2005.
7. C. Colby and P. Lee. Trace-based program analysis. In *POPL'96: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1996.
8. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77: Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1977.

9. P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In *IFIP'77: Conference on Formal Description of Programming Concepts*, 1977.
10. P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages), invited paper. In *ICCL'94: Proceedings of the International Conference on Computer Languages*, 1994.
11. M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 57–68, New York, NY, USA, 2002. ACM Press.
12. D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.
13. N. Dor, S. Adams, M. Das, and Z. Yang. Software validation via scalable path-sensitive value flow analysis. In *ISSTA '04: Proceedings of the ACM SIGSOFT international symposium on Software Testing and Analysis*, 2004.
14. J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Typestate verification: Abstraction techniques and complexity results. In *SAS'03: Proceedings of Static Analysis Symposium*, 2003.
15. C. Hammer, J. Krinke, and G. Snelting. Information flow control for java based on path conditions in dependence graphs. In *IEEE International Symposium on Secure Software Engineering*, 2006. To appear.
16. S. Hunt and D. Sands. On flow-sensitive security types. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 79–90, New York, NY, USA, 2006. ACM Press.
17. N. D. Jones and M. Rosendahl. Higher-order minimal function graphs. In *Journal of Functional and Logic Programming*, 1997.
18. H. G. Mairson. A simple proof of a theorem of Statman. *Theoretical Computer Science*, 1992.
19. A. C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.
20. H. R. Nielson and F. Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *POPL'97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997.
21. J. Robert H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.
22. D. A. Schmidt. Trace-based abstract interpretation of operational semantics. In *Lisp and Symbolic Computation*, 1998.
23. C. Skalka and S. Smith. History effects and verification. In *APLAS'04: The Second ASIAN Symposium on Programming Languages and Systems*, 2004.
24. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, Dec. 1996.
25. Y. Xie, A. Chou, and D. Engler. ARCHER: using symbolic, path-sensitive analysis to detect memory access errors. *SIGSOFT Software Engineering Notes*, 2003.
26. E. Yahav, T. Reps, M. Sagiv, and R. Wilhelm. Verifying temporal heap properties specified via evolution logic. In *ESOP'03: Proceedings of the European Symposium on Programming and Systems*, 2003.

A The Arithmetic Decision Procedure

The semantic model is as follows:

$$\mathbb{E} \vDash_{mdl} (\vartheta_1 \ rop\ \vartheta_2) \Rrightarrow \{\overline{b_k}\} \text{ iff } b \in \{\overline{b_k}\} \iff \exists \vartheta_1', \vartheta_2'.\ \big(\vartheta_1 \backslash \mathbb{E} \hookrightarrow^* \vartheta_1'\big) \land \big(\vartheta_2 \backslash \mathbb{E} \hookrightarrow^* \vartheta_2'\big) \land \big((\vartheta_1'\ rop\ \vartheta_2') \dashrightarrow b\big)$$

where, Figure 4 defines $(\vartheta\ rop\ \vartheta') \dashrightarrow b$. The decision procedure employed should be conservative with respect to the model.

Definition 4. *The decision procedure dp is sound iff* $\mathbb{E} \vDash_{mdl} (\vartheta_1\ rop\ \vartheta_2) \Rrightarrow \{\overline{b_j}\}$ *and* $\mathbb{E} \vDash_{dp} (\vartheta_1\ rop\ \vartheta_2) \Rrightarrow \{\overline{b_k}\}$ *implies* $\{\overline{b_j}\} \subseteq \{\overline{b_k}\}$.

So for example if $\mathbb{E} \vDash x \Rrightarrow \{-5, x-1\}$ and $\mathbb{E} \vDash y \Rrightarrow \{0\}$ then $\mathbb{E} \vDash_{mdl} (x == y) \Rrightarrow \{\mathit{false}\}$; however depending on the chosen decision procedure, either $\mathbb{E} \vDash_{dp} (x == y) \Rrightarrow \{\mathit{false}\}$ or $\mathbb{E} \vDash_{dp} (x == y) \Rrightarrow \{\mathit{true}, \mathit{false}\}$ is acceptable, but it must not be that $\mathbb{E} \vDash_{dp} (x == y) \Rrightarrow \{\mathit{true}\}$ or $\mathbb{E} \vDash_{dp} (x == y) \Rrightarrow \{\}$.

We can then define the relation $\mathbb{E} \vDash_b \vartheta \Rrightarrow \{\overline{b_m}\}$ which conservatively approximates the boolean values of ϑ given $\mathbb{E}$: $\mathbb{E} \vDash_b \vartheta \Rrightarrow \{\overline{b_m}\}$ iff either ϑ is inchoate in $\mathbb{E}$ and $\{\overline{b_m}\} = \{\mathit{true}, \mathit{false}\}$, or $\mathbb{E} \vDash \vartheta \Rrightarrow B_0 \cup \{\overline{\vartheta_k\ rop_k\ \vartheta_k'}\}$, $\overline{\mathbb{E} \vDash_{dp} (\vartheta_k\ rop_k\ \vartheta_k') \Rrightarrow B_k}$ and $\{\overline{b_m}\} = B_0 \cup B_1 \cup \ldots \cup B_k$, where $B := \{b \mid b \text{ is a } \mathit{boolean\ value}\}$.

B Proofs for *N*CI

Lemma 0 (Determinism of *N*CI). *If* $(\varnothing, \emptyset, e) \longrightarrow^n (\varnothing, \mathbb{E}, \vartheta)$ *and also* $(\varnothing, \emptyset, e) \longrightarrow^{n'} (\varnothing, \mathbb{E}', \vartheta')$ *then* $n = n'$, $\mathbb{E} = \mathbb{E}'$ *and* $\vartheta = \vartheta'$.

Proof. Directly by the rules in Figure 6.

We say "*N*CI is stuck at $(\mathcal{S}, \mathbb{E}, e)$" if none of the *N*CI rules are applicable to $(\mathcal{S}, \mathbb{E}, e)$.

Lemma 1 (Termination of *N*CI). *NCI terminates on all input.*

Proof. The recurse check premise, *i.e.* $\mathcal{F} \notin \mathcal{S}$ in app-push–nonrec-call, does not allow two instances of the same function on the call stack and *N*CI never creates new functions by inlining their free variables with corresponding near-values from the environment, in effect bounding the maximum possible stack depth to the number of unique functions statically found in the program. Also, the environment is monotonic and no new expressions are created during *N*CI, implying that the maximum size of the environment is limited to $|x| * |\vartheta|$, where $|x|$ is the number of distinct program variables and $|\vartheta|$ is the number of distinct near-values statically found in the program. Once the environment reaches its upper bound it can neither grow nor shrink (due to monotonicity), implying future reruns will never invoke app-rerun–not-fixed-point as the environment cannot change during them. The app-prune–rec-call already prunes all recursive invocations, in effect eliminating all back edges from the *N*CI, forcing it to only sequentially move forward. Hence the *N*CI of all future function invocations is guaranteed to terminate. Since a program can have only finitely many statements, its *N*CI will eventually terminate or get stuck. ◻

Lemma 2 (Simulation of CI by *N*CI). *If* e *has a near-concrete interpretation and* $e \rightarrowtail^n e_n$ *then* $(\varnothing, \emptyset, e) \Longrightarrow (\mathcal{S}'_n, \mathbb{E}'_n, e'_n)$ *such that* $e_n \preccurlyeq e'_n \backslash \mathbb{E}'_n$*.*

Proof-Sketch. Proceed by induction on the length of $e \rightarrowtail^n e_n$, assuming the result holds for all smaller n. CI invokes a fresh copy of the function body at each invocation and hence might have multiple continuations of the same function active in the current expression, while *N*CI reuses the function body for recursive invocations due to pruning and reruning, hence it has at most one copy of a function body in the expression at any point. Due to this asymmetry only the most recent function invocations are compared, which is still sound since the simulation always starts with an empty call stack, hence in effect all function invocations are compared during the course of the simulation. As mentioned in Section 2.1 recursive computation in CI is simulated by the corresponding fixed-point cycle in *N*CI, while non-recursive computation is directly simulated by *N*CI.

Lemma 3 (Soundness of *N*CI). *If a closed* e *has a NCI then its CI either computes to a value or computes forever.*

Proof-Sketch. As per Lemma 2 the *N*CI of e is a conservative approximation of its CI, implying the latter cannot get stuck at a point if its *N*CI is not stuck at the corresponding simulating node.

Lemma 4 (Complexity of *N*CI). *The runtime complexity of NCI is exponential in size of the program, modulo the complexity of the decision procedure.*

Proof. For a given program of size n, let $|x| = n$, $|\vartheta| = n$ and $|\mathcal{F}| = n$, where $|x|$ is the number of distinct variables, $|\vartheta|$ is the number distinct near-values and $|\mathcal{F}|$ is the number of unique functions statically in the program. Hence as proved in Lemma 1 the maximum size of the environment $\mathsf{max}(\mathbb{E}) = |x| * |\vartheta| = n^2$. Also, the maximum size of the call stack is $|\mathcal{F}| = n$, implying the maximum possible distinct stack configurations by direct permutation is $n!$. Due to pruning and reruning of recursive invocations, at most one copy of any function body is ever laid out in the intermediate expression during *N*CI *i.e.* there is no code duplication, implying the maximum possible configurations of intermediate expressions during the course of *N*CI is n. Hence the maximum possible distinct (stack, environment, expression) configurations by simple combinatorics is $n! * \mathsf{max}(\mathbb{E}) * n = n! * n^2 * n$, implying the maximum number of distinct states in the *N*CI is $n! * n^2 * n$. Using Stirling's approximation for large factorials, assuming sufficiently large n, $n! * n^2 * n = O(2^{n \, log \, n})$, the runtime complexity of *N*CI. □

Lemma 5 (Complexity of *N*CI for First-order Programs). *The runtime complexity of NCI is polynomial in size of first-order programs, modulo the complexity of decision procedure employed.*

Proof. For a given first-order program of size n, let $|x| = n$, $|\vartheta| = n$ and $|\mathcal{F}| = n$, where $|x|$ is the number of distinct variables, $|\vartheta|$ is the number distinct near-values and $|\mathcal{F}|$ is the number of unique functions statically in the program. Now

loc	$::=$	$loc_1 \mid loc_2 \mid \ldots$	*heap location*
ψ	$::=$	$\ldots \mid loc$	*near-concrete value (extended)*
κ	$::=$	$\ldots \mid \mathsf{Ref}\ \vartheta \mid !\vartheta \mid \vartheta := \vartheta'$	*atomic computation (extended)*
$\mathbb{H}$	$=$	$\{loc \mapsto \vartheta \mid loc$ is a *heap location* and ϑ is a *near-value*$\}$	*heap*
$\mathcal{S}$	$::=$	$\varnothing \mid \mathcal{S} : [\mathcal{F}]^{\mathbb{H}}_{\mathbb{E}\mathbb{H}\mathbb{H}}$	*stack (redefined)*

Fig. 8. $\mathcal{N}$CI$^{\mathbb{H}}$: Language Syntax Grammar (Extended/Redefined)

since the program is first-order the sequential order of function definitions in the program is critical, in that a function can only invoke itself and functions that are defined before it. Hence, the maximum number of possible stack configurations during $\mathcal{N}$CI, taking into account that that re-activations are pruned, is only $(1 + 2 + 3 + \ldots + n) = n(n-1)/2$ – there only one stack configuration of size n with all n functions invoked in reverse order of their definition, and there are only n stack configurations of size 1. We already know that the maximum size of the environment is n^2 and the number of intermediate $\mathcal{N}$CI expressions is n. Hence the maximum number of distinct (stack, environment, expression) configurations by simple combinatorics is $n(n-1)/2 * n^2 * n = O(n^5)$, the runtime complexity of $\mathcal{N}$CI for first-order programs. □

C Mutable State via Abstract Heap ($\mathcal{N}$CI$^{\mathbb{H}}$)

Figure 8 gives the language syntax grammar extended with notation needed for adding mutable state to $\mathcal{N}$CI. Each stack frame is tagged with three heaps in addition to the (checkpointed) environment. The heap in the superscript (called the *collecting invocation heap*) is used to collect the heaps at the corresponding recursive call sites while pruning, to be used later during rerun. The two heaps in the subscript denote the checkpointed heaps: at invocation (called *checkpoint invocation heap*) and return sites (called *checkpoint return heap*) respectively. $\mathcal{S}$ is isomorphic to $\mathcal{S} : \varnothing$.

Figures 9 and 10 give the corresponding semantics rules for $\mathcal{N}$CI$^{\mathbb{H}}$. The notation $\mathbb{H}(loc)$ corresponds to heap lookup and $\mathbb{H}[loc \mapsto \vartheta]$ implies overwriting of the near-value in location loc with ϑ.

Termination The heap does not grow monotonically during the analysis, in that mappings can be overwritten and fresh locations generated during the course of $\mathcal{N}$CI$^{\mathbb{H}}$. Note that both $\mathfrak{ref}^{\mathbb{H}}$ and $\mathfrak{set}^{\mathbb{H}}$ add near-values, which are part of the program syntax, to the heap. Also recall the same program code is interpreted during reruns, implying that the order of heap operations during each of the reruns remains the same. Now $\mathfrak{ref}^{\mathbb{H}}$ reuses heap locations during rerun. The $\mathfrak{set}^{\mathbb{H}}$ rule writes the same near-value to the heap during each rerun, albeit at possibly more locations each time; but those locations are entailed by the environment. Hence, if the environment does not change across reruns, the locations that $\mathfrak{set}^{\mathbb{H}}$

$$\frac{\textbf{let}^{\mathbb{H}}}{\big(\mathcal{S},\, \mathbb{E},\, \mathbb{H},\, \textsf{Let } x = \vartheta \textsf{ In } e\big) \longrightarrow \big(\mathcal{S},\, \mathbb{E} \cup \{x \mapsto \vartheta\},\, \mathbb{H},\, e\big)}$$

arith-op$^{\mathbb{H}}$

$$\frac{\vartheta \text{ and } \vartheta' \text{ are either integral or inchoate in } \mathbb{E}}{\big(\mathcal{S},\, \mathbb{E},\, \mathbb{H},\, \textsf{Let } x = (\vartheta\ aop\ \vartheta') \textsf{ In } e\big) \longrightarrow \big(\mathcal{S},\, \mathbb{E} \cup \{x \mapsto (\vartheta\ aop\ \vartheta')\},\, \mathbb{H},\, e\big)}$$

cond-op$^{\mathbb{H}}$

$$\frac{\vartheta \text{ and } \vartheta' \text{ are either integral or inchoate in } \mathbb{E}}{\big(\mathcal{S},\, \mathbb{E},\, \mathbb{H},\, \textsf{Let } x = (\vartheta\ rop\ \vartheta') \textsf{ In } e\big) \longrightarrow \big(\mathcal{S},\, \mathbb{E} \cup \{x \mapsto (\vartheta\ rop\ \vartheta')\},\, \mathbb{H},\, e\big)}$$

app-push–nonrec-call$^{\mathbb{H}}$

$$\frac{\mathcal{F} = \lambda_f x.\, e \quad \mathcal{F} \notin \mathcal{S}}{\big(\mathcal{S},\, \mathbb{E},\, \mathbb{H},\, \mathcal{F}\ \vartheta\big) \longrightarrow \big(\mathcal{S} : [\mathcal{F}]^{\mathbb{H}}_{\mathbb{E}\mathbb{H}\emptyset},\, \mathbb{E} \cup \{f \mapsto \mathcal{F},\, x \mapsto \vartheta\},\, \mathbb{H},\, \langle e \rangle\big)}$$

app-prune–rec-call$^{\mathbb{H}}$

$$\frac{\mathcal{F} = \lambda_f x.\, e \quad \lfloor e \rfloor = \vartheta_r}{\Big(\mathcal{S} : [\mathcal{F}]^{\mathbb{H}_a}_{\mathbb{E}^\natural \mathbb{H}_a^\natural \mathbb{H}_r^\natural} : \mathcal{S}',\, \mathbb{E},\, \mathbb{H},\, \mathcal{F}\ \vartheta\Big) \longrightarrow \Big(\mathcal{S} : [\mathcal{F}]^{\mathbb{H}_a \cup \mathbb{H}}_{\mathbb{E}^\natural \mathbb{H}_a^\natural \mathbb{H}_r^\natural} : \mathcal{S}',\, \mathbb{E} \cup \{x \mapsto \vartheta\},\, \mathbb{H}_r^\natural,\, \vartheta_r\Big)}$$

app-rerun–not-fixed-point$^{\mathbb{H}}$

$$\frac{\mathcal{F} = \lambda_f x.\, e \quad \lfloor e \rfloor = \vartheta_r \quad \Big(\mathbb{E},\, \mathbb{H}_a,\, \mathbb{H}_r\Big) \neq \Big(\mathbb{E}^\natural,\, \mathbb{H}_a^\natural,\, \mathbb{H}_r^\natural\Big)}{\Big(\mathcal{S} : [\mathcal{F}]^{\mathbb{H}_a}_{\mathbb{E}^\natural \mathbb{H}_a^\natural \mathbb{H}_r^\natural},\, \mathbb{E},\, \mathbb{H}_r,\, \langle \vartheta_r \rangle\Big) \longrightarrow \Big(\mathcal{S} : [\mathcal{F}]^{\mathbb{H}_a}_{\mathbb{E}\mathbb{H}_a\mathbb{H}_r},\, \mathbb{E},\, \mathbb{H}_a,\, \langle e \rangle\Big)}$$

app-pop–fixed-point$^{\mathbb{H}}$

$$\frac{\mathcal{F} = \lambda_f x.\, e \quad \lfloor e \rfloor = \vartheta_r}{\Big(\mathcal{S} : [\mathcal{F}]^{\mathbb{H}_a}_{\mathbb{E}\mathbb{H}_a\mathbb{H}_r},\, \mathbb{E},\, \mathbb{H}_r,\, \langle \vartheta_r \rangle\Big) \longrightarrow \Big(\mathcal{S},\, \mathbb{E},\, \mathbb{H}_r,\, \vartheta_r\Big)}$$

app-inchoate$^{\mathbb{H}}$

$$\frac{f \text{ is inchoate in } \mathbb{E}}{\big(\mathcal{S},\, \mathbb{E},\, \mathbb{H},\, \textsf{Let } x = f\ \vartheta \textsf{ In } e\big) \longrightarrow \big(\mathcal{S},\, \mathbb{E},\, \mathbb{H},\, e\big)}$$

app-merge$^{\mathbb{H}}$

$$\frac{\mathbb{E} \vDash f \Rrightarrow \{\overline{\mathcal{F}_k}\} \quad k \geq 1 \qquad \forall 1 \leq i \leq k.\ \big(\mathcal{S},\, \mathbb{E},\, \mathbb{H},\, \mathcal{F}_i\ \vartheta\big) \longrightarrow^{n_i} \big(\mathcal{S},\, \mathbb{E}_i,\, \mathbb{H}_i,\, \vartheta_i\big)}{\Big(\mathcal{S},\, \mathbb{E},\, \mathbb{H},\, \textsf{Let } x = f\ \vartheta \textsf{ In } e\Big) \longrightarrow \Big(\mathcal{S},\, \bigcup_{1 \leq i \leq k} \mathbb{E}_i \cup \{x \mapsto \vartheta_i\},\, \bigcup_{1 \leq i \leq k} \mathbb{H}_i,\, e\Big)}$$

Fig. 9. NCI$^{\mathbb{H}}$ rules: Near-Concrete Interpretation with Mutable State (more rules in Figure 10)

writes to also remain the same. Hence, the heaps are guaranteed to reach a fixed-point once the environment reaches a fixed-point. Since the environment is monotonically increasing it must ultimately reach an upper bound, and the heaps will stabilize in the next iteration, in case they have not done so already.

D Context-Sensitivity via Argument Tagging (NCI$^{\psi}$)

The grammar for variables is redefined to be $x ::= x^{\varnothing} \mid x^{\psi}$, where $x^{\varnothing}$ corresponds to untagged variables, as in NCI, and x^{ψ} denotes tagged variables.

$$\mathfrak{if\text{-}true}^{\mathbb{H}}$$
$$\left(\mathcal{S},\, \mathbb{E},\, \mathbb{H},\, \mathsf{If}\ \mathit{true}\ \mathsf{Then}\ e_1\ \mathsf{Else}\ e_2\right) \longrightarrow \left(\mathcal{S},\, \mathbb{E},\, \mathbb{H},\, e_1\right)$$

$$\mathfrak{if\text{-}false}^{\mathbb{H}}$$
$$\left(\mathcal{S},\, \mathbb{E},\, \mathbb{H},\, \mathsf{If}\ \mathit{false}\ \mathsf{Then}\ e_1\ \mathsf{Else}\ e_2\right) \longrightarrow \left(\mathcal{S},\, \mathbb{E},\, \mathbb{H},\, e_2\right)$$

$$\mathfrak{if\text{-}merge}^{\mathbb{H}}$$
$$\frac{\mathbb{E} \vDash_b \vartheta \Rrightarrow \{\overline{b_k}\} \quad 1 \le k \le 2 \qquad \forall 1 \le i \le k.\ \left(\mathcal{S},\, \mathbb{E},\, \mathbb{H},\, \mathsf{If}\ b_i\ \mathsf{Then}\ e_1\ \mathsf{Else}\ e_2\right) \longrightarrow^{n_i} \left(\mathcal{S},\, \mathbb{E}_i,\, \mathbb{H}_i,\, \vartheta_i\right)}{\left(\mathcal{S},\, \mathbb{E},\, \mathbb{H},\, \mathsf{Let}\ z = (\mathsf{If}\ \vartheta\ \mathsf{Then}\ e_1\ \mathsf{Else}\ e_2)\ \mathsf{In}\ e\right) \longrightarrow \left(\mathcal{S},\, \bigcup_{1\le i\le k} \mathbb{E}_i \cup \{z \mapsto \vartheta_i\},\, \bigcup_{1\le i\le k} \mathbb{H}_i,\, e\right)}$$

$$\mathfrak{ref}^{\mathbb{H}}$$
$$\frac{\mathbb{E} \vDash x \Rrightarrow \{loc\},\ \text{or fresh}\ loc \iff \neg\exists loc'.\, \mathbb{E} \vDash x \rightsquigarrow loc'}{\left(\mathcal{S},\, \mathbb{E},\, \mathbb{H},\, \mathsf{Let}\ x = \mathsf{Ref}\ \vartheta\ \mathsf{In}\ e\right) \longrightarrow \left(\mathcal{S},\, \mathbb{E},\, \mathbb{H} \cup \{loc \mapsto \vartheta\},\, \mathsf{Let}\ x = loc\ \mathsf{In}\ e\right)}$$

$$\mathfrak{get}^{\mathbb{H}}$$
$$\frac{\mathbb{E} \vDash \vartheta \Rrightarrow \{\overline{loc_k}\} \quad \overline{\mathbb{H}\left(loc_k\right) = \vartheta_k}}{\left(\mathcal{S},\, \mathbb{E},\, \mathbb{H},\, \mathsf{Let}\ x = !\vartheta\ \mathsf{In}\ e\right) \longrightarrow \left(\mathcal{S},\, \mathbb{E} \cup \{x \mapsto \overline{\vartheta_k}\},\, \mathbb{H},\, e\right)}$$

$$\mathfrak{set}^{\mathbb{H}}$$
$$\frac{\mathbb{E} \vDash \vartheta \Rrightarrow \{\overline{loc_k}\}}{\left(\mathcal{S},\, \mathbb{E},\, \mathbb{H},\, \mathsf{Let}\ x = \left(\vartheta := \vartheta'\right)\ \mathsf{In}\ e\right) \longrightarrow \left(\mathcal{S},\, \mathbb{E},\, \mathbb{H}\left[\overline{loc_k} \mapsto \vartheta'\right],\, \mathsf{Let}\ x = \vartheta'\ \mathsf{In}\ e\right)}$$

$$\mathfrak{reflex}^{\mathbb{H}}$$
$$\left(\mathcal{S},\, \mathbb{E},\, \mathbb{H},\, \vartheta\right) \longrightarrow^{0} \left(\mathcal{S},\, \mathbb{E},\, \mathbb{H},\, \vartheta\right)$$

$$\mathfrak{context}^{\mathbb{H}}$$
$$\frac{\left(\mathcal{S},\, \mathbb{E},\, \mathbb{H},\, \kappa\right) \longrightarrow \left(\mathcal{S}',\, \mathbb{E}',\, \mathbb{H}',\, \kappa'\right)}{\left(\mathcal{S},\, \mathbb{E},\, \mathbb{H},\, R[\kappa]\right) \longrightarrow \left(\mathcal{S}',\, \mathbb{E}',\, \mathbb{H}',\, R[\kappa']\right)}$$

$$\mathfrak{trans}^{\mathbb{H}}$$
$$\frac{\left(\mathcal{S},\, \mathbb{E},\, \mathbb{H},\, e\right) \longrightarrow^{n-1} \left(\mathcal{S}',\, \mathbb{E}',\, \mathbb{H}',\, e'\right) \qquad \left(\mathcal{S}',\, \mathbb{E}',\, \mathbb{H}',\, e'\right) \longrightarrow \left(\mathcal{S}'',\, \mathbb{E}'',\, \mathbb{H}'',\, e''\right)}{\left(\mathcal{S},\, \mathbb{E},\, \mathbb{H},\, e\right) \longrightarrow^{n} \left(\mathcal{S}'',\, \mathbb{E}'',\, \mathbb{H}'',\, e''\right)}$$

Fig. 10. (Continued from Figure 9). $\mathcal{N}$CI$^{\mathbb{H}}$ rules: Near-Concrete Interpretation with Mutable State

Figure 11 gives the new rules for $\mathcal{N}$CI$^{\psi}$, the other rules are identical to those of $\mathcal{N}$CI.

Definition 5 (Tagging).

1. $tag_{\psi}\left(\lambda_{f} x.\, e\right) = \lambda_{f^{\psi}} x^{\psi}.\, tag_{\psi}\left(e\left[f^{\psi}/f\right]\left[x^{\psi}/x\right]\right)$;
2. $tag_{\psi}\left(\vartheta\right) = \vartheta$, *if* ϑ *is not a function;*
3. $tag_{\psi}\left(\vartheta_1\ \vartheta_2\right) = \left(tag_{\psi}\left(\vartheta_1\right)\right)\ \left(tag_{\psi}\left(\vartheta_2\right)\right)$;
4. $tag_{\psi}\left(\mathsf{If}\ \vartheta\ \mathsf{Then}\ e_1\ \mathsf{Else}\ e_2\right) = \mathsf{If}\ tag_{\psi}\left(\vartheta\right)\ \mathsf{Then}\ tag_{\psi}\left(e_1\right)\ \mathsf{Else}\ tag_{\psi}\left(e_2\right)$;
5. $tag_{\psi}\left(\mathsf{Let}\ x = \kappa\ \mathsf{In}\ e\right) = \mathsf{Let}\ x^{\psi} = tag_{\psi}\left(\kappa\right)\ \mathsf{In}\ tag_{\psi}\left(e\left[x^{\psi}/x\right]\right)$.

The $\mathfrak{app\text{-}merge}^{\psi}$ rule interprets each of the functions flowing into a call-site with each of the concrete arguments flowing into it, in parallel and merges their resultant environments.

$$\frac{\mathcal{F} = \lambda_f x.\, e \quad \mathcal{F}' = tag_\psi(\mathcal{F}) = \lambda_{f^\psi} x^\psi.\, e' \quad \mathcal{F}' \notin \mathcal{S}}{(\mathcal{S}, \mathbb{E}, \mathcal{F}\ \psi) \longrightarrow (\mathcal{S} : [\mathcal{F}']_{\mathbb{E}}, \mathbb{E} \cup \{f^\psi \mapsto \mathcal{F}, x^\psi \mapsto \psi\}, \langle e \rangle)}\ \textsf{app-push–nonrec-call}^\psi$$

$$\frac{\mathcal{F} = \lambda_f x.\, e \quad \mathcal{F}' = tag_\psi(\mathcal{F}) = \lambda_{f^\psi} x^\psi.\, e' \quad \mathcal{F}' \in \mathcal{S} \quad \lfloor e' \rfloor = \vartheta'_r}{(\mathcal{S}, \mathbb{E}, \mathcal{F}\ \psi) \longrightarrow (\mathcal{S}, \mathbb{E} \cup \{x^\psi \mapsto \psi\}, \vartheta'_r)}\ \textsf{app-prune–rec-call}^\psi$$

$$\frac{\vartheta \text{ or } \vartheta' \text{ is inchoate in } \mathbb{E}}{(\mathcal{S}, \mathbb{E}, \mathsf{Let}\ x = \vartheta\ \vartheta'\ \mathsf{In}\ e) \longrightarrow (\mathcal{S}, \mathbb{E}, e)}\ \textsf{app-inchoate}^\psi$$

$$\frac{\begin{array}{l}\mathbb{E} \vDash \vartheta \Rrightarrow \{\overline{\mathcal{F}_k}\} \quad k \geq 1 \qquad \mathbb{E} \vDash \vartheta' \Rrightarrow \{\overline{\psi_n}\} \quad n \geq 1 \\ \forall 1 \leq i \leq k.\, \forall 1 \leq j \leq n.\ (\mathcal{S}, \mathbb{E}, \mathcal{F}_i\ \psi_j) \longrightarrow^{n_{ij}} (\mathcal{S}, \mathbb{E}_{ij}, \vartheta_{ij})\end{array}}{(\mathcal{S}, \mathbb{E}, \mathsf{Let}\ x = \vartheta\ \vartheta'\ \mathsf{In}\ e) \longrightarrow \left(\mathcal{S}, \bigcup_{1 \leq i \leq k} \bigcup_{1 \leq j \leq n} \mathbb{E}_{ij} \cup \{x \mapsto \vartheta_{ij}\}, e\right)}\ \textsf{app-merge}^\psi$$

Fig. 11. NCI$^\psi$ (new) rules: Near-Concrete Interpretation with Context-Sensitivity

$\mathfrak{p}$	$::=$	$\{\vartheta^b \mid \vartheta$ is a *near-value* and b is a *boolean value*$\}$	*path-tag*
E	$=$	$\{x \mapsto \vartheta^{\mathfrak{p}} \mid x$ is a *variable*, ϑ is a *value*, $\mathfrak{p}$ is a *path-tag*$\}$	*sub-environment*
$\mathbb{E}$	$=$	$E \lceil \mathfrak{p}$	*environment (redefined)*

Fig. 12. NCI$^{\mathfrak{p}}$: Language Syntax Grammar (Extended/Redefined)

Termination The stacked tags are based on the lexical scoping in the original program and not the dynamic scoping of functions, so the height of the stacked tags is strongly bound. Also, the tags are concrete near-values occurring in the program syntax implying the number of distinct tagged functions is also strongly bound. As argued for NCI, the environment is strongly bound as well. Hence, NCI$^\psi$ terminates on all input.

E Path-Sensitivity via Path Tagging (NCI$^{\mathfrak{p}}$)

Figure 12 gives the language syntax grammar extended with the notation needed for adding path-sensitivity to NCI. The environment is redefined to be $E\lceil\mathfrak{p}$, read as "the sub-environment E restricted to the path $\mathfrak{p}$". The $\lceil\mathfrak{p}$ notation can be thought of as a *filter* on E which only allows the values feasible (not conflicting) given path $\mathfrak{p}$ to be used for interpretation. Figure 13 gives only $\mathfrak{app\text{-}merge}^{\mathfrak{p}}$ and $\mathfrak{if\text{-}merge}^{\mathfrak{p}}$ rules for NCI$^{\mathfrak{p}}$; other rules are identical to those of NCI.

Definition 6 (Canonical Path-Tag). *Path-tag* $\mathfrak{p}$ *is canonical iff for all* ϑ *and* b *we have* $\vartheta^b \in \mathfrak{p} \implies \vartheta^{\neg b} \notin \mathfrak{p}$.

app-merge$^{\mathfrak{p}}$

$$\frac{E\lceil\mathfrak{p} \vDash f \Rrightarrow \{\overline{\mathcal{F}_k^{\mathfrak{p}_k}}\} \quad k \geq 1 \qquad \forall 1 \leq i \leq k.\ \big(\mathcal{S},\ E\lceil(\mathfrak{p}\sqcup\mathfrak{p}_i),\ \mathcal{F}_i\ \vartheta\big) \longrightarrow^{n_i} \big(\mathcal{S},\ E_i\lceil\mathfrak{p}'_i,\ \vartheta_i\big) \quad E_i \Leftarrow \mathfrak{p}'_i = E'_i}{\Big(\mathcal{S},\ E\lceil\mathfrak{p},\ \mathsf{Let}\ x = f\ \vartheta\ \mathsf{In}\ e\Big) \longrightarrow \Big(\mathcal{S},\ \Big(\biguplus_{1\leq i\leq k} E'_i \cup \big\{x \mapsto \vartheta_i^{\mathfrak{p}'_i}\big\}\Big)\Big\lceil \bigsqcup_{1\leq i\leq k}\mathfrak{p}'_i,\ e\Big)}$$

if-merge$^{\mathfrak{p}}$

$$\frac{E\lceil\mathfrak{p} \vDash_b \vartheta \Rrightarrow \{\overline{b_k^{\mathfrak{p}_k}}\} \quad 1 \leq k \leq 2 \qquad \forall 1 \leq i \leq k.\ \Big(\mathcal{S},\ E\Big\lceil\big(\mathfrak{p}\sqcup\mathfrak{p}_i\sqcup\{\vartheta^{b_i}\}\big),\ \mathsf{If}\ b_i\ \mathsf{Then}\ e_1\ \mathsf{Else}\ e_2\Big) \longrightarrow^{n_i} \Big(\mathcal{S},\ E_i\lceil\mathfrak{p}'_i,\ \vartheta_i\Big) \quad E_i \Leftarrow \mathfrak{p}'_i = E'_i}{\Big(\mathcal{S},\ E\lceil\mathfrak{p},\ \mathsf{Let}\ z = (\mathsf{If}\ \vartheta\ \mathsf{Then}\ e_1\ \mathsf{Else}\ e_2)\ \mathsf{In}\ e\Big) \longrightarrow \Big(\mathcal{S},\ \Big(\biguplus_{1\leq i\leq k} E'_i \cup \big\{z \mapsto \vartheta_i^{\mathfrak{p}'_i}\big\}\Big)\Big\lceil \bigsqcup_{1\leq i\leq k}\mathfrak{p}'_i,\ e\Big)}$$

Fig. 13. *N*CI$^{\mathfrak{p}}$ (new) rules: Near-Concrete Interpretation with Path-Sensitivity

We write $\mathfrak{p} \bowtie \mathfrak{p}'$ iff $\mathfrak{p} \cup \mathfrak{p}'$ is canonical, that is, paths $\mathfrak{p}$ and $\mathfrak{p}'$ are non-conflicting– $\mathfrak{p}$ does not contain a then-branch if corresponding else-branch is present in $\mathfrak{p}'$, and vice-versa.

Definition 7 (Canonical E). *The sub-environment E is canonical iff for all $x, \vartheta, \mathfrak{p}$ and $\mathfrak{p}'$ we have $\{x \mapsto v^{\mathfrak{p}},\ x \mapsto v^{\mathfrak{p}'}\} \subseteq E \implies (\mathfrak{p} = \mathfrak{p}') \land \mathfrak{p}$ is canonical.*

We write Ψ for $\psi^{\mathfrak{p}}$ and B for $\{b^{\mathfrak{p}} \mid b$ is a *boolean value* and $\mathfrak{p}$ is a *path-tag*$\}$. The path merge operator $\sqcup$ merges two path-tags, ensuring the resultant path-tag is canonical: $\mathfrak{p}\sqcup\mathfrak{p}' = \mathfrak{p}''$ iff $\forall\vartheta, b.\ (\vartheta^b \in \mathfrak{p} \cup \mathfrak{p}') \land (\vartheta^{\neg b} \notin \mathfrak{p} \cup \mathfrak{p}') \iff \vartheta^b \in \mathfrak{p}''$. The environment merge operator $\uplus$ merges two sub-environments ensuring the resultant sub-environment is canonical: $E \uplus E' = E''$ iff $\forall x, \vartheta, \mathfrak{p}, \mathfrak{p}'.\{x \mapsto \vartheta^{\mathfrak{p}},\ x \mapsto \vartheta^{\mathfrak{p}'}\} \subseteq E \cup E' \iff x \mapsto \vartheta^{\mathfrak{p}\sqcup\mathfrak{p}'} \in E''$. Also, $(E\lceil\mathfrak{p}) \uplus E' = (E \uplus E')\lceil\mathfrak{p}$. The environment tag operator '$\Leftarrow$' tags all the near-values in E with the path-tag $\mathfrak{p}$: $E \Leftarrow \mathfrak{p} = E'$ iff $\forall x, \vartheta, \mathfrak{p}'.\ x \mapsto \vartheta^{\mathfrak{p}'} \in E \iff x \mapsto \vartheta^{\mathfrak{p}'\sqcup\mathfrak{p}} \in E'$. E^+ is the transitive closure of E: it is the least superset of E such that $\{x \mapsto y^{\mathfrak{p}}, y \mapsto \vartheta^{\mathfrak{p}'}\} \subseteq E^+$ then $x \mapsto \vartheta^{\mathfrak{p}\sqcup\mathfrak{p}'} \in E^+$. The existential relation $E\lceil\mathfrak{p} \vDash x \rightsquigarrow \psi^{\mathfrak{p}'}$ iff $x \mapsto \psi^{\mathfrak{p}'} \in E^+$ and $\mathfrak{p} \bowtie \mathfrak{p}'$; observe how it only allows near-concrete values with path-tags not conflicting with the current path-tag $\mathfrak{p}$. The corresponding exhaustive relation can be defined as: $\mathbb{E} \vDash x \Rrightarrow \{\overline{\Psi_k}\}$ iff $\forall\Psi.\ \mathbb{E} \vDash x \rightsquigarrow \Psi \iff \Psi \in \{\overline{\Psi_k}\}$, and $\mathbb{E} \vDash \psi \Rrightarrow \{\psi^{\emptyset}\}$ makes the relation total on near-values. We say "ϑ is inchoate in $\mathbb{E}$" iff $\mathbb{E} \vDash \vartheta \Rrightarrow \{\}$. We say "$\vartheta$ is integral in $\mathbb{E}$", iff $\mathbb{E} \vDash \vartheta \Rrightarrow \{\overline{\psi_k^{\mathfrak{p}_k}}\}$, $k \geq 1$ and for all $1 \leq i \leq k$, either ψ_i is an integer, or $\psi_i = \vartheta_i\ aop_i\ \vartheta'_i$, for some ϑ_i, aop_i and ϑ'_i.

Analogous to *N*CI, there is no complete decision procedure for *N*CI$^{\mathfrak{p}}$; the (uncomputable) semantic model is as follows:

$$\begin{aligned}\mathbb{E} \vDash_{mdl} (\vartheta_1\ rop\ \vartheta_2)^{\mathfrak{p}} \Rrightarrow B \text{ iff } \Psi \in B \iff \exists\Psi_1, \Psi_2.\ &\big(\vartheta_1^{\mathfrak{p}} \backslash \mathbb{E} \hookrightarrow^* \Psi_1\big) \land \\ &\big(\vartheta_2^{\mathfrak{p}} \backslash \mathbb{E} \hookrightarrow^* \Psi_2\big) \land\end{aligned}$$

$$\big((\Psi_1 \ rop \ \Psi_2) \dashrightarrow \Psi\big)$$

where, $e^{\mathfrak{p}}\backslash\mathbb{E} \hookrightarrow^* e_n^{\mathfrak{p}_n}$ is the nondeterministic recursive inlining of mappings from $\mathbb{E}$ into e such that either $e^{\mathfrak{p}} = e_n^{\mathfrak{p}_n}$, or $x \in \mathsf{free}(e)$, $\mathbb{E} \vDash x \rightsquigarrow \vartheta^{\mathfrak{p}_x}$ and $(e[\vartheta/x])^{\mathfrak{p}\sqcup\mathfrak{p}_x}\backslash\mathbb{E} \hookrightarrow^* e_n^{\mathfrak{p}_n}$. Also $\psi^{\mathfrak{p}} \ rop \ \psi^{\mathfrak{p}'} \dashrightarrow b^{\mathfrak{p}\sqcup\mathfrak{p}'}$ iff $\psi \ rop \ \psi' \dashrightarrow b$. The decision procedure employed must be conservative with respect to the model.

Definition 8. *The decision procedure dp is sound iff* $\mathbb{E} \vDash_{mdl} (\vartheta_1 \ rop \ \vartheta_2)^{\mathfrak{p}} \Rrightarrow B$ *and* $\mathbb{E} \vDash_{dp} (\vartheta_1 \ rop \ \vartheta_2)^{\mathfrak{p}} \Rrightarrow B'$ *implies* $B <: B'$.

where, $B <: B'$ iff $\forall b, \mathfrak{p}, \mathfrak{p}'.\ b^{\mathfrak{p}} \in B \implies b^{\mathfrak{p}'} \in B' \wedge \mathfrak{p}' \subseteq \mathfrak{p}$. So for example, if $\mathbb{E} \vDash x \Rrightarrow \{5^{\mathfrak{p}_x}\}$ and $\mathbb{E} \vDash y \Rrightarrow \{5^{\mathfrak{p}_y}\}$ then $\mathbb{E} \vDash_{mdl} (x == y)^{\mathfrak{p}} \Rrightarrow \{true^{\mathfrak{p}\sqcup\mathfrak{p}_x\sqcup\mathfrak{p}_y}\}$; however depending upon the chosen decision procedure, either $\mathbb{E} \vDash_{mdl} (x == y)^{\mathfrak{p}} \Rrightarrow \{true^{\mathfrak{p}\sqcup\mathfrak{p}_x\sqcup\mathfrak{p}_y}\}$ or $\mathbb{E} \vDash_{mdl} (x == y)^{\mathfrak{p}} \Rrightarrow \{true^{\mathfrak{p}_1}, false^{\mathfrak{p}_2}\}$ is acceptable, for any $\mathfrak{p}_2$, and $\mathfrak{p}_1 \subseteq \mathfrak{p}\sqcup\mathfrak{p}_x\sqcup\mathfrak{p}_y$. However, suppose if $\mathbb{E} \vDash x \Rrightarrow \{5^{\mathfrak{p}_x}, 6^{\mathfrak{p}'_x}\}$ and $\mathbb{E} \vDash y \Rrightarrow \{5^{\mathfrak{p}_y}\}$ then $\mathbb{E} \vDash_{mdl} (x == y)^{\mathfrak{p}} \Rrightarrow \{true^{\mathfrak{p}\sqcup\mathfrak{p}_x\sqcup\mathfrak{p}_y}, false^{\mathfrak{p}\sqcup\mathfrak{p}'_x\sqcup\mathfrak{p}_y}\}$. Similarly if, $\mathbb{E} \vDash x \Rrightarrow \{5^{\mathfrak{p}_x}, (x-1)^{\mathfrak{p}'_x}\}$ and $\mathbb{E} \vDash y \Rrightarrow \{0^{\mathfrak{p}_y}\}$ then $\mathbb{E} \vDash_{mdl} (x == y)^{\mathfrak{p}} \Rrightarrow \{true^{\mathfrak{p}\sqcup\mathfrak{p}_x\sqcup\mathfrak{p}'_x\sqcup\mathfrak{p}_y}, false^{\mathfrak{p}\sqcup\mathfrak{p}_x\sqcup\mathfrak{p}'_x\sqcup\mathfrak{p}_y}\}$.

We can then define the relation $\mathbb{E} \vDash_b \vartheta \Rrightarrow B$ which conservatively approximates the boolean values of ϑ given $\mathbb{E}$: $\mathbb{E} \vDash_b \vartheta \Rrightarrow B$ iff either ϑ is inchoate in $\mathbb{E}$ and $B = \{true^{\emptyset}, false^{\emptyset}\}$, or $\mathbb{E} \vDash \vartheta \Rrightarrow B' \cup \{\overline{(\vartheta_k \ rop_k \ \vartheta'_k)^{\mathfrak{p}_k}}\}$, $\overline{\mathbb{E} \vDash_{dp} (\vartheta_k rop_k \vartheta'_k)^{\mathfrak{p}_k} \Rrightarrow B_k}$ and $B = B' \uplus B_1 \uplus \ldots \uplus B_k$, where $B \uplus B' = B''$ iff $\forall b, \mathfrak{p}, \mathfrak{p}'.\ \{b^{\mathfrak{p}}, b^{\mathfrak{p}'}\} \subseteq B \cup B' \iff b^{\mathfrak{p}\sqcup\mathfrak{p}'} \in B''$.