Near-Concrete Program Interpretation

Paritosh Shroff Scott F. Smith
Johns Hopkins University
{pari,scott}@cs.jhu.edu

Christian Skalka
University of Vermont
skalka@cs.uvm.edu

Abstract

We develop *near-concrete* program interpretation (NCI), a higher-order program analysis that achieves high precision via close, yet decidable, simulation of operational (concrete) semantics. NCI models mutable heaps with possibly recursive structure, and achieves flow-, context- and path-sensitivity in a uniform setting. The analysis also extracts a *nugget* that characterizes the value bindings resulting from program execution; it can be used to statically determine a wide range of non-trivial properties. The technical novelty of the system lies in a *prune-rerun* technique for approximating recursive functions. To illustrate the generality and usefulness of the system, we show how it can be used to statically enforce temporal program safety, information flow security, and array bounds checking properties.

1. Introduction

Approaches to static program understanding have evolved to be ever more sophisticated. Initially there were two quite separate schools: program analysis and formal methods. Program analysis excelled at automatic calculation of simple low-level properties of production programs, and formal methods excelled at manual or semi-automatic calculation of more advanced properties of idealized programs, which were provably as opposed to experimentally (partially) validated. Much recent progress has been made on combining the strengths of these two approaches. For example, model-checking techniques applied to production programs now allow higher-level properties of programs to be automatically verified [15, 5]. Type systems, which existed both as simple systems in production languages and expressive systems in theorem provers, have been extended to allow high-level assertions about realistic languages expressed as types [12, 30, 14].

This paper also aims to combine the strengths of program analysis and formal methods as a framework for enforcement of sophisticated safety properties: we aim to automatically establish complex properties of higher-order, stateful programs via a provably correct analysis. We develop a new, fundamental method called *near-concrete* interpretation (NCI), which defines an abstraction of program operational semantics that cuts very closely to the actual program execution (the *concrete* interpretation). NCI is related to abstract interpretation [9, 10] in that an abstract execution is performed, but there are also many differences in technique and in applicability. For example, NCI applies to fully higher-order programs with a mutable heap whereas abstract interpretation focuses on first-order programs. We develop a novel *prune-rerun* technique for sound and decidable interpretation of higher-order recursive functions. Additionally, NCI returns an environment which we call the *nugget* of the program: it has distilled out the function calls, conditional branching, and mutability, and advanced program properties can then be directly "read" from the nugget as we will illustrate.

The paper presents NCI in multiple phases. Full NCI is notation-rich, so for simplicity we factor the presentation into a core system and extensions. We present intuitions about the core system and the prune-rerun technique in Section 2, and Section 3 formalizes the core and proves it is a sound abstraction of the concrete interpretation. Several extensions are then outlined in Section 4, including the addition of mutable state to the language, and refinements of NCI to incorporate context-sensitivity and path-sensitivity. Due to space limitations the formal details of the extensions are given in a technical report [25].

Why NCI? A multitude of program analyses have been defined; what is so special about NCI? We argue that it is unique in several dimensions. 1) It can be viewed as showing how abstract interpretation techniques can extend to fully higher-order programs. 2) The condensation of program behavior to a nugget provides unique insights into program properties and extends the class of properties that can be automatically verified; we concretely illustrate this with the examples of Section 5. 3) NCI is a single algorithm that can be used to answer vastly different program analysis questions, rather than defining a new analysis for every new problem.

To illustrate this generality of NCI, Section 5 presents three different applications. We first show how NCI is powerful enough to statically determine value range bounds for integers and other data, illustrating that it also can be applied to the array bounds problem [6]. We next show that information flow security properties can be directly read from the nugget, and that NCI gives results more accurate than previous approaches to the problem. Lastly we show how temporal program safety properties (*a.k.a.* typestate properties) can be directly verified.

The performance of NCI in other problem domains also compares favorably to existing state-of-the-art analyses. The NCI heap is destructively updated so it is a flow-sensitive analysis. The heap abstraction does not merge allocations and infers recursive structures for cyclic heaps, giving an expressiveness comparable to recent state-of-the-art shape analyses [24]. NCI path-sensitivity also infers correlations between conditional statements as do the path conditions of [15].

2. Overview

We start with a concrete motivating example. Consider the following program:

$$\begin{array}{l}\mathsf{let}\ \mathit{flag} = \mathsf{ref}\ \mathsf{false}\ \mathsf{in}\\ \mathsf{let}\ \mathit{fact} = \lambda_{\mathit{fact}} n.\ \mathsf{if}\ (n{=}{=}0)\ \mathsf{then}\ \mathit{flag} := \mathsf{true};\ 1\\ \qquad\qquad\qquad\qquad \mathsf{else}\ \mathit{fact}(n-1) * n\\ \mathsf{in}\ \mathit{fact}(\mathbf{x});\ !\mathit{flag}\end{array}$$

which computes to true if $\mathbf{x}$ is replaced with say 5, and computes forever if $\mathbf{x}$ is replaced with say -5. All flow-insensitive static analyses would infer the possible resultant values of the above program to be $\{\mathsf{false}, \mathsf{true}\}$. Analyses in the NCI family will be able to automatically infer the following properties for this example.

flow-sensitivity If the program terminates, it computes to exactly `true`. This fact is nontrivial to establish, NCI needs to determine that the base case of $fact(\mathbf{x})$ has to be executed in order for it to terminate.

nontermination detection If $\mathbf{x} < 0$ the program computes forever. Of course we will *not* be able to detect nontermination for all nonterminating programs, but simple nontermination patterns can be detected.

value range analysis 1. If $\mathbf{x} \geq 0$ then the range of values assigned to $fact$'s argument variable n during the course of computation is $[0, \mathbf{x}]$.

2. If $\mathbf{x} < 0$ then the range of values assigned to $fact$'s argument variable n during the course of computation is $[-\infty, \mathbf{x}]$.

A key to finding a sound and decidable approximation of recursive computation is an approximation of recursive function invocation, or as we say in *pruning* the arbitrarily many re-activations of the same function. A main contribution of NCI lies in approximating recursive computations in a minimally lossy manner, via the *prune-rerun* technique. For non-recursive programs, NCI is not an approximation: it is isomorphic to the operational semantics or *concrete interpretation* (CI) of such programs. We now informally define the core NCI algorithm over pure functional programs.

2.1 The Core NCI

Consider the following pure functional factorial function, placed in A-normal form [17] so that each program point has an associated program variable:

$$fact = \lambda_{fact} n.\ \mathsf{let}\ r = \mathsf{if}\ (n == 0)\ \mathsf{then}\ 1 \\ \mathsf{else\ let}\ y = fact\ (n-1)\ \mathsf{in}\ y * n \\ \mathsf{in}\ r,$$

Note that explicit notation for recursive function invocation is used here, but recursion encoded via higher-order constructs *e.g.* the Y-combinator achieves similar results. Figure 1 graphically shows how the CI and NCI relate. A Data- and Control-Flow Graph (DCFG) is shown for $fact\ (n)$ under the (a) CI and (b) NCI.

To understand the concrete execution shown in Figure 1(a), $fact\ (1000)$ would execute the "rec-call cycle" path in the figure a thousand times before the `true` branch is taken, after which the "rec-return cycle" path is followed another thousand times; the invocation $fact\ (-5)$ would loop in the rec-call cycle forever. It is impossible to predict if a recursive computation will terminate, and so NCI collapses each recursive computation into a single cycle with a fixed-point property, such that the latter is guaranteed to be a sound approximation of the former.

Figure 1(b) shows the DCFG of the NCI of $fact\ (n)$, directly illustrating how the NCI computation compares to the concrete computation. There are three major differences between the two DCFGs: 1) The NCI environments $\mathbb{E}$ contain symbolic mappings and are monotonically increasing in size, whereas in the CI each loop binds n to a different concrete value $1000, 999, 998, \ldots$; 2) The separate rec-call and rec-return cycles of the CI are combined in a single "rerun" cycle in NCI, turning stack-based computation into stack-free looping and setting up a cycle for which we will be able to iteratively find a fixed-point; and 3) there is no recursive invocation $fact\ (n-1)$ in the NCI. Instead, the recursive call is pruned using a sound technique we develop below, the symbolic argument mapping $n \mapsto n-1$ is added to the environment, and $y \mapsto r$, where r is the return variable of $fact$, is added to the environment to demarcate where the return value from the recursive call should eventually appear. The return variable r is initially only a placeholder for the return value of recursive call $fact\ (n-1)$, and it is filled in by iterating, as we will soon show. Since the environment is monotonically increasing in a finitely bounded ordering (as we will demonstrate), the NCI will always reach a fixed-point and terminate, whereas the CI may not.

The NCI Environment The NCI environment $\mathbb{E}$ is a set of symbolic mappings that can contain recursive structures such as $\{y \mapsto r,\ r \mapsto y * n\}$, and can also contain multiple bindings of the same variable, such as $\{r \mapsto 1,\ r \mapsto y*n\}$. 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 expressions such as $x-1$, $y*z$, $n==0$, $1+2$, $w \geq 0$, *etc.*.

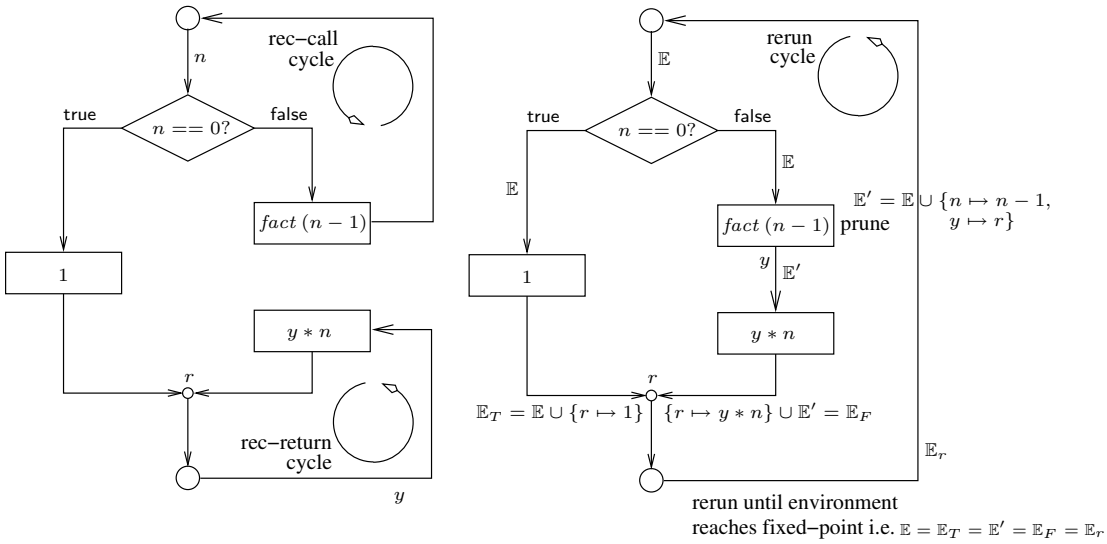
The environment can be viewed as defining the rules of an ambiguous context-free generative grammar, such that the variables in its domain are the nonterminals of the grammar, while the conventional values (functions, integers, booleans, *etc.*) and binary operators ($<$, $==$, $\geq$, *etc.*) comprise the terminals. Hence, possibly infinitely many argument (and return) values of function calls can be expressed concisely in the environment. Consider an environment with mappings $\{n \mapsto -5,\ n \mapsto n-1\}$: the infinite set $\{-5, (-5-1), (-5-1-1), \ldots\}$, which is equivalent to $\{-5, -6, -7, \ldots\}$, represents the language of the corresponding grammar with start symbol n, and is the set of all possible concrete values for n.

Since the environment can contain multiple bindings for the same variable, it is possible for the guard of a conditional branch to map to both true and false in the environment, 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 and $\mathbb{E}_r = \mathbb{E}_T \cup \mathbb{E}_F$.

It is essential to note that the environment grows monotonically during the analysis: mappings are only added and never removed. Furthermore, all values placed in the NCI 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 the monotonic environment growth is bounded during analysis. This implies that growth cannot continue forever but must eventually reach a fixed-point, and NCI will always terminate.

Pruning and re-running to a fixed-point Figure 2 shows the full sequence of steps of the NCI for $fact\ (-5)$. The boxes with dashed edges show the NCI environment state at various points in the analysis. A function call stack (not shown in the figure) is also 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 expression will appear on the NCI stack, so the stack size is bounded.

We now trace through the steps shown in the figure. The NCI stack will contain at most one activation per function expression. In our example the recursive call $fact\ (n-1)$ is pruned, not taken, and the argument value $(n-1)$ and return variable (r) mappings are added to the environment. The environment immediately after pruning in the initial run is marked ① in the figure. In the first pass of analyzing $fact(-5)$, the environment contains no return values, hence r, and transitively y, are not bound to any value in the environment – we say they are both *inchoate*, in a state of incomplete formedness. Inchoate variables bear some resemblance to program futures: computation can proceed, even though some result values are temporarily missing. Finally $r \mapsto y*n$ is added to the environment which completes the initial pass through the function body. Since the environments at the start and end of the just concluded run differ, NCI has not reached a fixed-point and we compute a second pass, marked ② in the figure, starting with the latest environment.



(a) Concrete Interpretation (CI) (b) Near−Concrete Interpretation (NCI)

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

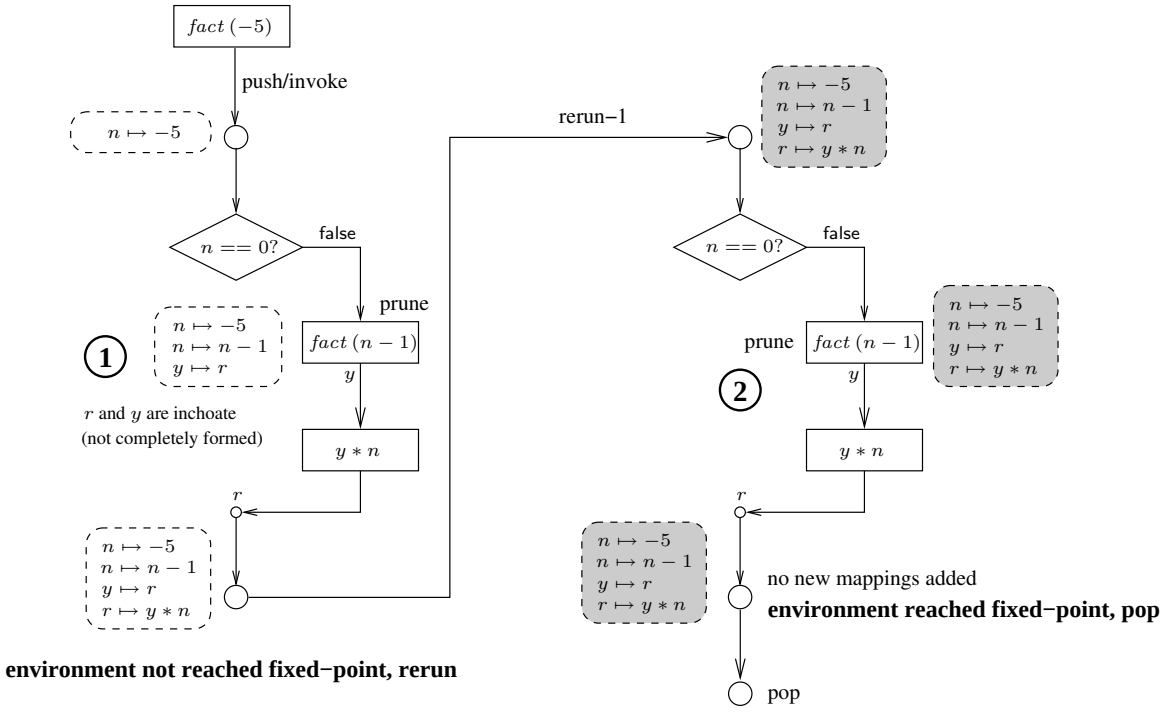


Figure 2. Near-concrete interpretation of $fact\ (-5)$. Rerun-1 (marked ②) represents the fixed-point cycle: no new mappings were added throughout the rerun, so further reruns will not change the environment.

At this point the environment mappings $\{n \mapsto -5,\ n \mapsto n-1\}$ intuitively imply that $n \leq -5$ and thus the condition $n == 0$ is always false; a decision procedure is incorporated into NCI which automatically deduces such simple arithmetic properties from the environment. Based on this result from the decision procedure, only the else-branch need be interpreted. No new mappings are added during the second run, implying the environment has stabilized to a fixed-point, and the analysis terminates. This final stabilized environment $\mathbb{E}$ is the *nugget* generated by NCI for this example, the distilled essence of value flows of the program, and properties can be read off of it as we now show.

Nontermination Detection Note that r does not have any concrete mappings in the nugget $\mathbb{E}$, as the recursive mappings $\{y \mapsto r, r \mapsto y * n\}$ lack a base case. This means that the invocation

$fact(-5)$ must never return. NCI can in general be used to detect nontermination in such cases when the decision procedure is able to determine that the base case(s) of a recursive computation cannot be reached based on the variable mappings in the environment.

Soundness The soundness of NCI is justified by the final fixed-point cycle. *e.g.* the cycle marked ② in Figure 2: rerun-1 faithfully simulates the entire concrete interpretation of $fact(-5)$, in that every node in its CI can be faithfully simulated by the corresponding node in rerun-1 and some value its expanded stable environment. What is perhaps surprising about the algorithm is how computing ahead with inchoate results is not unsound: the computation steps are being taken in a different order than in the CI, but since the environment is monotone and will be "filled in" by further iterations over the function body, the final stable environment resolves all bindings and faithfully simulates concrete recursions.

Termination and Complexity As observed above, environments grow monotonically during the analysis, towards an upper bound. This means that the 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. This bounds the number of potential stacks by the number of finite permutations of functions statically declared in the program. With finitely many environments, stacks, and subexpressions, the NCI is a finite-state system, and with finitely bounded reruns there are no infinite paths through this finite state space.

The runtime complexity of NCI 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.

Another Example Figure 3 shows the complete NCI of $fact(5)$. It is similar to Figure 2, but $\{n \mapsto 5, n \mapsto n-1\}$ implies $n \leq 5$, hence a sound interpretation of $n == 0$ in the second run (marked ②) has to yield both true and false implying both branches of the conditional must be interpreted in parallel and their resulting environments merged, adding $r \mapsto 1$ 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 (marked ③), no new mappings are added so the environment has stabilized at a fixed-point, and NCI terminates.

These two examples do not address many cases such as mutual recursion, passing functions as data, *etc*, but the fundamental algorithm is apparent from these simple examples. The next section presents the formal details for the general case.

3. Formalization of the Core NCI

3.1 Language Model and Concrete Interpretation

Our core programming language model (Figure 4) is a pure higher-order functional language with arithmetic, booleans, conditional branching and let bindings. $\lambda_f x.\, e$ is a function with f as 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 5 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 and relational operations ($\nu\ op\ \nu$) are evaluated lazily, hence the prefix "near-" in near-values: $(1+2)$ is just a step of simple reduction away from 3. So for example the reduction of $(1+2)$ to 3, denoted as $(1+2) \dashrightarrow 3$, is postponed until it is essential to do so for the computation to proceed, that is, the branching condition in the *if* rule needs to be resolved. 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. We write $\nu \dashrightarrow^* \nu'$ to denote the transitively closed reduction of near-value ν iff $\nu \dashrightarrow^n \nu'$, for some n, and $\neg\exists \nu''.\ \nu' \dashrightarrow \nu''$.

$x,\ y,\ z,\ f,\ r$			*program variables*
i	$::=$	$\ldots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \ldots$	*integer*
b	$::=$	$\mathsf{true} \mid \mathsf{false}$	*boolean*
aop	$::=$	$+ \mid - \mid * \mid /$	*binary arithmetic operator*
rop	$::=$	$> \mid < \mid == \mid !=$	*binary relational operator*
op	$::=$	$aop \mid rop$	*binary operator*
σ	$::=$	$i \mid b \mid \nu\ op\ \nu \mid \lambda_f x.\, e$	*near-concrete value*
ν	$::=$	$x \mid \sigma$	*near-value*
κ	$::=$	$e \mid \nu\ \nu \mid \mathsf{if}\ \nu\ \mathsf{then}\ e\ \mathsf{else}\ e$	*atomic computation*
e	$::=$	$\nu \mid \mathsf{let}\ z = \kappa\ \mathsf{in}\ e$	*A-normalized expression*
$\mathcal{R}$	$::=$	$\bullet \mid \mathsf{let}\ z = \mathcal{R}\ \mathsf{in}\ e$	*reduction context*

Figure 4. Core Language: Syntax Grammar

$$\frac{(i_i\ aop\ i_2) = i}{(i_i\ aop\ i_2) \dashrightarrow i} \qquad \frac{(i_i\ rop\ i_2) = b}{(i_i\ rop\ i_2) \dashrightarrow b} \qquad \frac{\nu \dashrightarrow^{n-1} \nu' \quad \nu' \dashrightarrow \nu''}{\nu \dashrightarrow^{n} \nu''}$$

$$\frac{\nu_1 \dashrightarrow \nu_1'}{(\nu_1\ op\ \nu_2) \dashrightarrow (\nu_1'\ op\ \nu_2)} \qquad \frac{\nu_2 \dashrightarrow \nu_2'}{(i_1\ op\ \nu_2) \dashrightarrow (i_1\ op\ \nu_2')} \qquad \nu \dashrightarrow^0 \nu$$

$$\textit{let}\quad \mathsf{let}\ z = \nu\ \mathsf{in}\ e \rightarrowtail e[\nu/z] \qquad \textit{app}\quad (\lambda_f x.\, e)\ \nu \rightarrowtail e\left[(\lambda_f x.\, e)/f\right][\nu/x]$$

$$\textit{if-true}\quad \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \rightarrowtail e_1 \qquad \textit{if-false}\quad \mathsf{if}\ \mathsf{false}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \rightarrowtail e_2$$

$$\textit{if}\quad \frac{(\nu\ rop\ \nu') \dashrightarrow^n b}{\mathsf{if}\ (\nu\ rop\ \nu')\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \rightarrowtail \mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2}$$

$$\textit{context}\quad \frac{\kappa \rightarrowtail \kappa'}{\mathcal{R}[\kappa] \rightarrowtail \mathcal{R}[\kappa']} \qquad \textit{reflex}\quad e \rightarrowtail^0 e \qquad \textit{trans}\quad \frac{e \rightarrowtail^{n-1} e' \quad e' \rightarrowtail e''}{e \rightarrowtail^n e''}$$

Figure 5. Core Language: Operational Semantics Rules

e	$::=$	$\ldots \mid \langle e \rangle$	*A-normalized expression* (extd.)
$\mathcal{R}$	$::=$	$\ldots \mid \langle \mathcal{R} \rangle$	*reduction context* (extd.)
$\mathbb{E}$	$::=$	$\{\overline{x_i \mapsto \nu_i}\}$	*environment*
$\mathcal{F}$	$::=$	$\lambda_f x.\, e$	*function*
$\mathbb{S}$	$::=$	$\varnothing \mid \mathbb{S} : [\mathcal{F}]_{\mathbb{E}}$	*stack*

Figure 6. NCI: Syntax Grammar

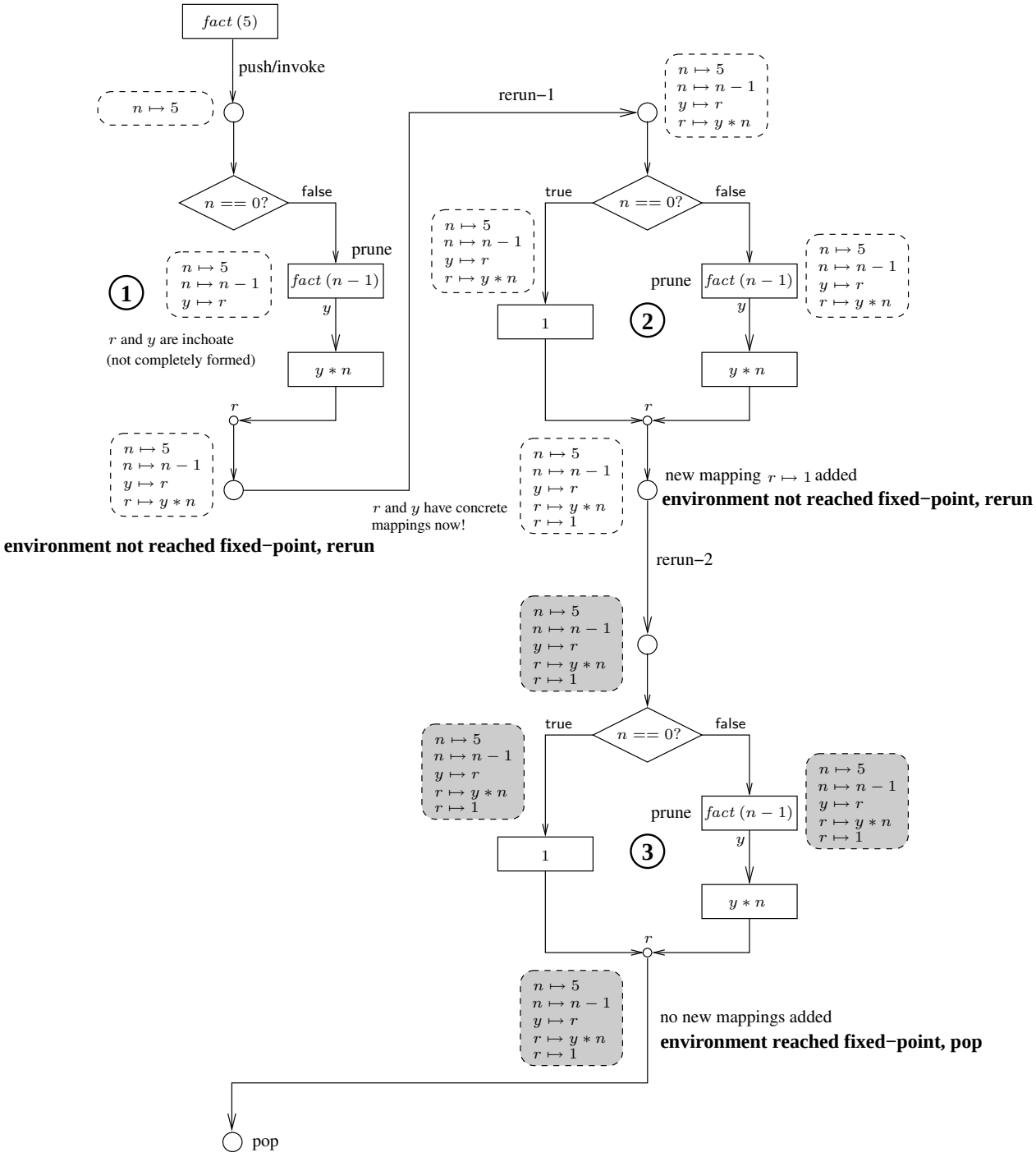


Figure 3. Near-concrete interpretation of $fact\ (5)$. ③ is the fixed-point cycle as no new mappings are added throughout the rerun, so further reruns will not change the environment.

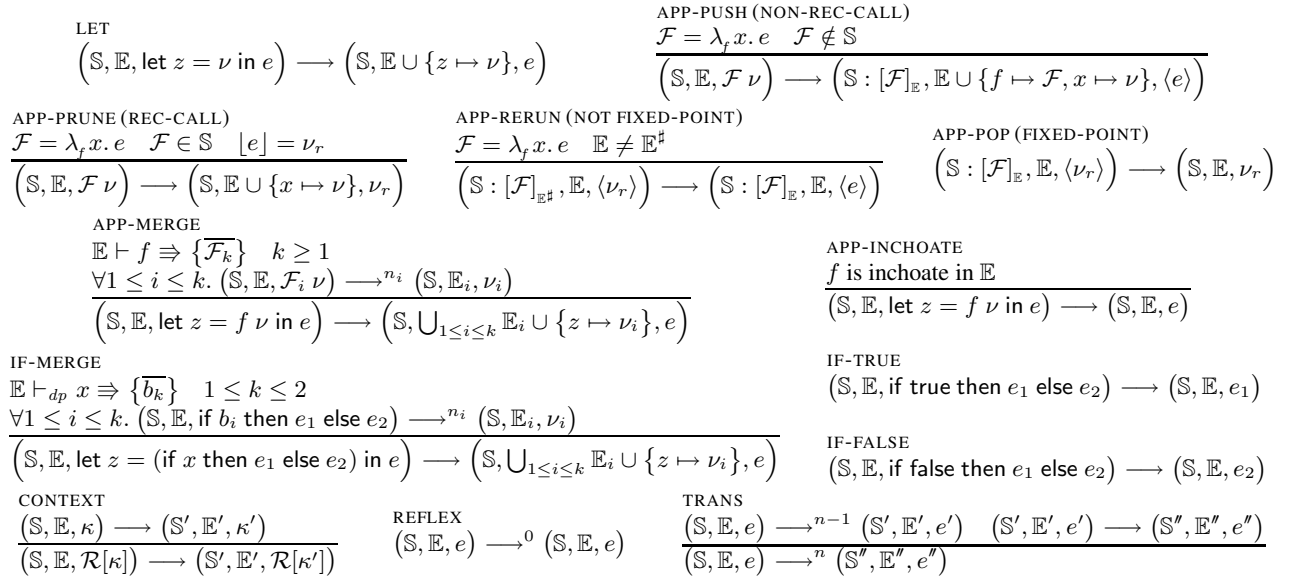


Figure 7. NCI Semantics Rules

3.2 Near-Concrete Program Interpretation

Figure 6 gives the language syntax extended with the notation needed for the core NCI. $\mathbb{E}$ is the environment, a multi-mapping relation from variables to near-values. The mappings can be recursive. $\mathbb{S}$ 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– the checkpointed environment is used to detect fixed-point cycles. $\varnothing$ is the empty stack. $\langle e \rangle$ is used to delimit function body e so that the end of its interpretation can be detected and appropriate action taken (depending upon whether a fixed-point has been reached or not). The NCI semantics, a hybrid between small- and big-step semantics, is given in Figure 7.

We first define some notation used in the rules. $\overline{a_k}$ is short for $a_1, a_2, \ldots, a_k$ and $\{x \mapsto \overline{\nu_k}\}$ for $\{x \mapsto \nu_1, \ldots, x \mapsto \nu_k\}$. The instack relation $\mathcal{F} \in \mathbb{S}$ holds iff $\mathbb{S} = \mathbb{S}' : [\mathcal{F}']_{\mathbb{E}}$ and either $\mathcal{F} = \mathcal{F}'$ or $\mathcal{F} \in \mathbb{S}'$. The result placeholder of an expression is a near-concrete value defined by the following relation: $\lfloor \nu \rfloor = \nu$ and $\lfloor \mathsf{let}\ z = \kappa\ \mathsf{in}\ e \rfloor = \lfloor e \rfloor$. $\mathbb{E}^+$ is the transitive closure of $\mathbb{E}$: it is the least superset of $\mathbb{E}$ such that if $\{x \mapsto y, y \mapsto \nu\} \subseteq \mathbb{E}^+$ then $x \mapsto \nu \in \mathbb{E}^+$. The existential relation $\mathbb{E} \vdash x \rightsquigarrow \sigma$ holds iff $x \mapsto \sigma \in \mathbb{E}^+$, and $\mathbb{E} \vdash \sigma \rightsquigarrow \sigma$ holds trivially to make it total on near-values. The corresponding exhaustive relation is defined as $\mathbb{E} \vdash \nu \Rrightarrow \{\overline{\sigma_k}\}$ iff $\forall \sigma.\, \mathbb{E} \vdash \nu \rightsquigarrow \sigma \iff \sigma \in \{\overline{\sigma_k}\}$. We say "$\nu$ is inchoate in $\mathbb{E}$" iff $\mathbb{E} \vdash \nu \Rrightarrow \{\}$, implying that ν is a variable and it is not bound to any near-concrete value in $\mathbb{E}$.

We now give a brief overview of the key rules. Recall the overview (Section 2.1) for intuitions on the bigger picture of the algorithm. The LET rule adds the near-value ν, as is without reduction, analogous to *let* in Figure 5, to the environment. The APP-PUSH (NON-REC-CALL) only applies when the function $\mathcal{F}$ is not already on the call stack *i.e.* not a re-activation of $\mathcal{F}$. It pushes a new frame onto the call stack with the current (checkpoint) environment. 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 ν to $\mathbb{E}$ and returning the result placeholder ν_r of $\mathcal{F}$'s body, in effect simulating the return of $\mathcal{F}$'s re-activation without actually performing one. The argument ν will be used in later rerun(s) of $\mathcal{F}$. If ν_r is 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 such as 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. Also APP-PUSH (NON-REC-CALL) and APP-PRUNE (REC-CALL) ensure that at any point during NCI the call stack contains at most one frame per function. If the function being applied is itself not yet concrete, APP-INCHOATE pushes the NCI past such a call-site without doing anything– if the function gets concretized later, it will be invoked in later reruns before the fixed-point cycle is reached. A redex $\langle \nu \rangle$ indicates that a function invocation has completed its interpretation, and it either needs to be 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 the function invocation is rerun, the arguments collected during pruning of its re-activations in the just concluded run are now used for interpretation. Correspondingly the return values accumulated will now be available in the continuations after the re-activation sites. Once the environment fixed-point cycle is reached *i.e.* the environment did not change during the previous run, all further reruns are rendered redundant since NCI is deterministic, and the corresponding function invocation is popped. 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 an arithmetic decision procedure to compute (conservatively) the possible truth values of the condition variable x given $\mathbb{E}$. If x can resolve to both true and false, both of the branches are interpreted in parallel and their results are merged. This decision procedure is expressed as $\mathbb{E} \vdash_{dp} x \Rrightarrow \{\overline{b_k}\}$, meaning that under $\mathbb{E}$, x conservatively has boolean value(s) $\{\overline{b_k}\}$. We define no concrete decision procedure here. One could always be completely conservative and assume that the condition x resolves to both true and false regardless of $\mathbb{E}$ *i.e.* $\forall \mathbb{E}, x.\ \mathbb{E} \vdash_{dp} x \Rrightarrow \{\mathsf{true}, \mathsf{false}\}$. Indeed, the general problem is undecidable since for example arbitrary Diophantine equations can be expressed in the environment. However, useful approximations for the common practical cases can be imagined. At the simplest level, analogues to constant propagation could resolve relations involving variables

having only value bindings in the environment. More advanced techniques can be brought to bear in cases of environments such as $\{n \mapsto 5, n \mapsto n-1\}$, where simple inductive techniques allow the set of possible concrete bindings for n to be bound. One obvious approach is to invoke a theorem-prover capable of automatically establishing simple inductive facts, but it also should be possible to construct an *ab initio* decision procedure to capture common programming patterns. An exploration of this topic is beyond the scope of this paper, and is an important direction for future work.

Any given NCI environment $\mathbb{E}$ constitutes a set of production rules for an ambiguous context-free generative grammar $G = (N, \Sigma, P, S)$, with nonterminals $N = \{x \mid x \mapsto \nu \in \mathbb{E}\}$ and terminals $\Sigma = i \cup b \cup op \cup \{\lambda_f\, x.\, e\}$. Only the free variables within a function $\lambda_f\, x.\, e$ are nonterminals. Any variable in N can be chosen as the start symbol S. The language $L(G)$ then represents the set of all possible near-concrete values for S entailed by the corresponding environment. The grammar based view of the environment is analogous to the following environment inlining relation, which inlines mappings from the environment $\mathbb{E}$ into an expression e:

Definition 3.1 (Environment Inlining: $\hookrightarrow_{\mathbb{E}}$). *1. (reflex).* $e \hookrightarrow_{\mathbb{E}}^{0} e$,
2. (one-step). $e \hookrightarrow_{\mathbb{E}} e'$ *iff* $x \in \mathit{free}(e)$, $x \mapsto \nu \in \mathbb{E}$ *and* $e' = e[\nu/x]$,
3. (trans). $e \hookrightarrow_{\mathbb{E}}^{n} e_n$ *iff* $e \hookrightarrow_{\mathbb{E}}^{n-1} e_{n-1}$ *and* $e_{n-1} \hookrightarrow_{\mathbb{E}} e_n$.
4. (transitive closure). $e \hookrightarrow_{\mathbb{E}}^{*} e'$ *iff* $e \hookrightarrow_{\mathbb{E}}^{n} e'$, *for some* n, *and* $\neg\exists x, \nu.\, x \in \mathit{free}(e') \wedge x \mapsto \nu \in \mathbb{E}$.

The semantic model for the decision procedure can then be defined as:

Definition 3.2 (Semantic Model for Decision Procedure). $\mathbb{E} \vdash_{dp}^{mdl} \nu \Rrightarrow \{\overline{b_k}\}$ *iff*

1. $\forall b.\, b \in \{\overline{b_k}\} \implies \big(\exists \sigma.\, \nu \hookrightarrow_{\mathbb{E}}^{*} \sigma \wedge \sigma \dashrightarrow^{*} b\big)$, *and*
2. $\forall \sigma.\, \nu \hookrightarrow_{\mathbb{E}}^{*} \sigma \implies \big(\sigma \dashrightarrow^{*} b \wedge b \in \{\overline{b_k}\}\big)$.

Note that the Case 2 in above definition requires all possible near-concrete values for ν given the environment $\mathbb{E}$ to be reducible to a boolean value. Hence, if $\nu \hookrightarrow_{\mathbb{E}}^{*} 5$ then the decision procedure is stuck.

Definition 3.3 (Soundness of Decision Procedure). *A decision procedure dp is sound iff* $\mathbb{E} \vdash_{dp}^{mdl} \nu \Rrightarrow \{\overline{b_j}\}$ *and* $\mathbb{E} \vdash_{dp} \nu \Rrightarrow \{\overline{b_k}\}$ *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 $\mathbb{E} \vdash_{dp}^{mdl} (x == y) \Rrightarrow \{\mathsf{false}\}$. However depending on the chosen decision procedure, either $\mathbb{E} \vdash_{dp} (x == y) \Rrightarrow \{\mathsf{false}\}$ or $\mathbb{E} \vdash_{dp} (x == y) \Rrightarrow \{\mathsf{true}, \mathsf{false}\}$ is acceptable, but it must not be that $\mathbb{E} \vdash_{dp} (x == y) \Rrightarrow \{\mathsf{true}\}$ or $\mathbb{E} \vdash_{dp} (x == y) \Rrightarrow \{\}$. Correspondingly, in parts ② and ③ of Figure 3, $\mathbb{E} \vdash n \Rrightarrow \{5, n-1\}$ which implies $\mathbb{E} \vdash_{dp}^{mdl} (n == 0) \Rrightarrow \{\mathsf{true}, \mathsf{false}\}$ and hence $\mathbb{E} \vdash_{dp} (n == 0) \Rrightarrow \{\mathsf{true}, \mathsf{false}\}$, due to which both the branches were interpreted in parallel as per IF-MERGE. However, in part ① of Figure 3, $\mathbb{E} \vdash n \Rrightarrow \{5\}$ which implies $\mathbb{E} \vdash_{dp}^{mdl} (n == 0) \Rrightarrow \{\mathsf{false}\}$ and hence, assuming a strong enough decision procedure, $\mathbb{E} \vdash_{dp} (n == 0) \Rrightarrow \{\mathsf{false}\}$, due to which only the else-branch was interpreted.

We say "ν is integral in $\mathbb{E}$" iff either ν is inchoate in $\mathbb{E}$, or $\mathbb{E} \vdash \nu \Rrightarrow \{\overline{\sigma_k}\}$ and for all $1 \leq i \leq k$, either σ_i is an integer, or $\sigma_i = \nu_i\ aop_i\ \nu_i'$, for some aop_i, such that ν_i and ν_i' are themselves integral in $\mathbb{E}$. We say "ν is consistent in $\mathbb{E}$", iff either ν is inchoate in $\mathbb{E}$, or $\mathbb{E} \vdash \nu \Rrightarrow \{\overline{\sigma_k}\}$ and for all $1 \leq i \leq k$, either σ_i is an integer, or a boolean, or a function, or $\sigma_i = \nu_i\ op_i\ \nu_i'$, for some op_i, such that ν_i and ν_i' are integral in $\mathbb{E}$. Consistency of ν in $\mathbb{E}$ implies that ν does not represent semantically invalid near-concrete values, like $\mathsf{true} + 5$ given $\mathbb{E}$.

Definition 3.4 (Near-Concrete Interpretation (NCI)). *We say "*e *has a near-concrete interpretation" iff* e *is closed,* $(\varnothing, \emptyset, e) \longrightarrow^{n} (\varnothing, \mathbb{E}, \nu)$ *and* ν *is consistent in* $\mathbb{E}$, *for some* n.

The derivation tree of $(\varnothing,\ \emptyset,\ e) \longrightarrow^{n} (\varnothing,\ \mathbb{E},\ \nu)$ can be viewed as a finite state transition system or a finite automaton representing the NCI of e, as illustrated in Figures 2 and 3. 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 [25].

Lemma 3.5 (Termination of NCI). *NCI terminates on all input.*

Proof. The recurse check premise, *i.e.* $\mathcal{F} \notin \mathbb{S}$ in APP-PUSH (NON-REC-CALL), does not allow two instances of the same function on the call stack and NCI 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 NCI, implying that the maximum size of the environment is limited to $|x| * |\nu|$, where $|x|$ is the number of distinct program variables and $|\nu|$ 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, in effect eliminating all back edges from the NCI and forcing it to only sequentially move forward. The APP-PRUNE (REC-CALL) already prunes all recursive invocations; hence the NCI of all future function invocations is guaranteed to terminate. Since a program can have only finitely many statements, its NCI will eventually terminate or get stuck. □

We next prove the soundness of NCI by showing that it is a faithful simulation of CI for any e, modulo the soundness of decision procedure employed. The big-step semantics for NCI can be defined as follows:

Definition 3.6 (NCI Big-Step Semantics). $(\mathbb{S}, \mathbb{E}, e) \Longrightarrow (\mathbb{S}', \mathbb{E}', e')$ *iff either,*

1. (TRANS, REFLEX). $(\mathbb{S}, \mathbb{E}, e) \longrightarrow^{n} (\mathbb{S}', \mathbb{E}', e')$, *for some* n, *or*
2. (APP-MERGE). $e = \mathcal{R}[\mathsf{let}\ z = f\ \nu\ \mathsf{in}\ e_{next}]$, $e' = \mathcal{R}[\mathsf{let}\ z = e_{sub}\ \mathsf{in}\ e_{next}]$, $\mathbb{E} \vdash f \rightsquigarrow \mathcal{F}_i$ *and* $(\mathbb{S}, \mathbb{E}, \mathcal{F}_i\ \nu) \Longrightarrow (\mathbb{S}', \mathbb{E}', e_{sub})$, *or*
3. (IF-MERGE). $e = \mathcal{R}\big[\mathsf{let}\ z = (\mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ \mathsf{in}\ e_{next}\big]$, $e' = \mathcal{R}[\mathsf{let}\ z = e_{sub}\ \mathsf{in}\ e_{next}]$, $\mathbb{E} \vdash_{dp} x \Rrightarrow \{\overline{b_k}\}$, $b_i \in \{\overline{b_k}\}$ *and* $(\mathbb{S}, \mathbb{E}, \mathsf{if}\ b_i\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) \Longrightarrow (\mathbb{S}', \mathbb{E}', e_{sub})$.
4. (trans). $(\mathbb{S}, \mathbb{E}, e) \longrightarrow^{n} (\mathbb{S}_n, \mathbb{E}_n, e_n)$ *and* $(\mathbb{S}_n, \mathbb{E}_n, e_n) \Longrightarrow (\mathbb{S}', \mathbb{E}', e')$, *for some* n, $\mathbb{S}_n$, $\mathbb{E}_n$ *and* e_n.

We can then define the simulation relation which relates an expression e_1 (corresponding to CI) with another expression e_2 (corresponding to NCI) given an environment $\mathbb{E}$.

Definition 3.7 (Simulation Relation: $\sqsubseteq_{\mathbb{E}}$). $e_1 \sqsubseteq_{\mathbb{E}} e_2$ *iff* $e_2 \hookrightarrow_{\mathbb{E}}^{*} e_{2_{conc}}$ *and either,*

1. $e_1 = e_{2_{conc}}$, *or*
2. $e_1 = \mathcal{R}_1[e_{1_{sub}}]$ *and* $e_{2_{conc}} = \mathcal{R}_2\big[\langle e_{1_{sub}} \rangle\big]$, *for some* $\mathcal{R}_1$ *and* $\mathcal{R}_2$.

It essentially relates the redex's corresponding to the most recent function invocations (if any). Note that since e_1 and $e_{1_{sub}}$ correspond to CI they do not contain delimiters '$\langle\rangle$'.

Lemma 3.8 (Simulation of CI by NCI). *If e has a near-concrete interpretation and $e \rightarrowtail^n e_n$ then $(\varnothing, \emptyset, e) \Longrightarrow (\mathbb{S}'_n, \mathbb{E}'_n, e'_n)$ such that $e_n \sqsubseteq_{\mathbb{E}'_n} e'_n$.*

Lemma 3.9 (Soundness of NCI). *If e has a near-concrete interpretation then its concrete interpretation either computes to a value or computes forever.*

Lemma 3.10 (Complexity of NCI). *The runtime complexity of NCI is exponential in size of the program, modulo the complexity of the decision procedure.*

The above is a theoretical bound reflecting the worst case scenario for NCI. The following example realizes the worst case performance:

$$\begin{array}{l}\mathsf{let}\ x = \mathsf{if}\ c\ \mathsf{then}\ \lambda x_1.\, x_1\ x_1 \\ \qquad\quad \mathsf{else\ if}\ c'\ \mathsf{then}\ \lambda x_2.\, x_2\ x_2 \quad \mathsf{in}\ x\ x \\ \qquad\qquad \mathsf{else}\ \lambda x_3.\, x_3\ x_3\end{array}$$

where c and c' are some conditions which resolve to both true and false. Hence NCI interprets all the branches and merges their results— the environment at '$x\ x$' contains mappings $\{x \mapsto \lambda x_1.\, x_1\ x_1, x \mapsto \lambda x_2.\, x_2\ x_2, x \mapsto \lambda x_3.\, x_3\ x_3\}$ for x. As a result NCI is forced to successively invoke them in all possible permutations *i.e.* 3!, resulting in 3! different stack configurations. Generalizing the above example to n such functions would result in $n!$ different stack configurations.

All of the exponential programs we have been able to construct are highly unnatural; it is an open question whether any common programming patterns would realize the exponential bound. The following result shows how higher-order functions, which allows programs to permute functions in possibly exponential number of ways as in the above example, is responsible for the exponential bound.

Lemma 3.11 (Complexity of NCI for First-order Programs). *The runtime complexity of NCI is polynomial in size of first-order programs, modulo the complexity of the decision procedure.*

Note that the non-elementary bound on simply typed λ-reduction [21] does not apply to NCI since there can still be multiple simultaneous activations of the same function in the simply typed λ-calculus. Consider the following example, where f_a is re-activated inside f_b:

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

[21] encodes primitive recursion in simply typed λ-calculus and shows the complexity of its reduction as non-elementary, while NCI prunes all recursive invocations including ones such as the above.

4. Extensions to the Core NCI

A treatment of mutation and state is necessary for our analysis to apply to realistic programs, so we first extend NCI to incorporate a mutable heap. We also extend NCI with context- and path-sensitivity, to obtain higher precision in practical settings. The core framework of the prune-rerun technique guarantees termination, and allows these extensions to be incorporated with relative ease.

4.1 Mutable State via an Abstract Heap

We incorporate mutable state by maintaining an abstract heap $\mathbb{H}$ with destructive updates in addition to the environment and call stack; we call this analysis NCI$^{\mathbb{H}}$. A single step transition in NCI$^{\mathbb{H}}$ is of the form $(\mathbb{S}, \mathbb{E}, \mathbb{H}, e) \longrightarrow (\mathbb{S}', \mathbb{E}', \mathbb{H}', e')$, adding a heap to the state. The syntax of atomic computations is extended to include heap allocation, dereference and update operations: $\kappa ::= \ldots \mid \mathsf{ref}\ \nu \mid\ !\nu \mid \nu := \nu$. The concrete heap in CI$^{\mathbb{H}}$ (concrete execution, CI, with state) is the usual one-to-one mapping relation from locations to near-values, while the abstract heap in NCI$^{\mathbb{H}}$ is a multi-mapping relation from locations to near-values — both have the same syntax $\mathbb{H} ::= \{\overline{loc_i \mapsto \nu_i}\}$, where loc is a heap location. The heap operations in NCI$^{\mathbb{H}}$ are flow-sensitive and mimic those of the CI$^{\mathbb{H}}$— $loc := \nu$ destructively updates the near-value in loc to be ν, removing any previous mappings, and ref ν generates a "fresh" heap location (the quotes on fresh will be explained below) with ν in it.

As an example, consider the following stateful variation of $fact\,(n)$ employing a memory-based fixed-points (*a.k.a.* tying the knot) and dynamically allocating memory in its body:

$$\begin{array}{l}\mathsf{let}\ fact = \mathsf{ref}\ 0\ \mathsf{in} \\ \quad fact := \lambda n.\ \mathsf{if}\ (n{==}0)\ \mathsf{then}\ 1 \\ \qquad\qquad \mathsf{else\ let}\ ptr = \mathsf{ref}\ (n-1)\ \mathsf{in} \\ \qquad\qquad\qquad n * (!fact)\,(!ptr) \qquad ; \\ \quad (!fact)\,(5)\end{array}$$

The ';' is syntactic sugar for sequencing. Consider the NCI$^{\mathbb{H}}$ of this example. The initial ref statement generates a fresh heap location, say loc, so the heap then contains $loc \mapsto 0$, and $fact \mapsto loc$ is added to the environment. The second assignment to $fact$ *overwrites* 0 so that the heap now contains only a single mapping $loc \mapsto (\lambda n.\mathsf{if} \ldots (!fact)\,(!ptr))$. During the first run of $(!fact)\,(5)$, the expression "$ptr = \mathsf{ref}\ (n-1)$" causes the mapping $loc' \mapsto n-1$ to be added to the abstract heap for new location loc', and $ptr \mapsto loc'$ is also added to the environment. The same location loc' is then *reused* in rerun cycles, instead of generating a fresh location for re-analysis of the same allocation statement. The final fixed point heap produced by NCI$^{\mathbb{H}}$ is $\{loc \mapsto (\lambda n.\mathsf{if} \ldots (!fact)\,(!ptr)), loc' \mapsto n-1\}$, where all the dynamically allocated locations created by "$ptr = \mathsf{ref}\ (n-1)$" in CI$^{\mathbb{H}}$ are fused into one NCI$^{\mathbb{H}}$ heap location loc'.

The NCI$^{\mathbb{H}}$ heap is a multi-mapping relation due the fact that heaps must merge results of conditionals and applications, via rules IF-MERGE$^{\mathbb{H}}$ and APP-MERGE$^{\mathbb{H}}$ (the NCI$^{\mathbb{H}}$ equivalents of IF-MERGE and APP-MERGE). For example, in:

$$\mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in}\ \ \mathsf{if}\ c\ \mathsf{then}\ x := 5\ \mathsf{else}\ x := \mathsf{true}$$

where c is some condition which statically resolves to both true and false, the resultant NCI$^{\mathbb{H}}$ heap is $\{loc \mapsto 5, loc \mapsto \mathsf{true}\}$ with environment $\{x \mapsto loc\}$. The abstract heap does not contain $loc \mapsto 0$ which is killed by update operation in each of the branches. If only one branch performed an update and the other did not, then $loc \mapsto 0$ would remain in the heap.

The prune-rerun strategy for recursive computations is extended to stabilize the heap along with the environment. The challenge with stabilizing the heap state is that the CI$^{\mathbb{H}}$ is executing in the order of Figure 1(a), taking the rec-call and rec-return cycles, whereas the NCI$^{\mathbb{H}}$ needs to follow Figure 1(b) with a single cycle. For computing the environment this re-ordering of execution is transparent because it is monotonically increasing, but the heap is not. The solution is to checkpoint the heap state at *two different points* in the recursion, before and after the recursive calls, and then at recursive call prune points switch from one heap to the other. This "tricks the NCI$^{\mathbb{H}}$ heap" into thinking that the computation is happening in the correct (CI$^{\mathbb{H}}$) order.

The abstract stack is used to hold the two checkpoint heaps, so the NCI stack frame $[\mathcal{F}]_{\mathbb{E}}$ is extended to $[\mathcal{F}]^{\mathbb{H}}_{\mathbb{E}\mathbb{H}}$ in NCI$^{\mathbb{H}}$. The heap in the superscript is the *checkpoint invocation heap*, corresponding to CI's heap in the rec-call cycle, and it checkpoints the heap at the function invocation points (both at the start of the function and the start of recursive calls). The heap in the subscript is the *checkpoint return heap*, corresponding to CI's heap in rec-return cycle, and it

checkpoints the heap at the end of the function body. A fixed-point is reached when both heaps stabilize.

To illustrate this in more detail, consider the example from Section 2 with x replaced by 5:

$$\begin{array}{l}\mathsf{let}\ \mathit{flag} = \mathsf{ref}\ \mathsf{false}\ \mathsf{in}\\ \mathsf{let}\ \mathit{fact} = \lambda_{\mathit{fact}} n.\ \mathsf{if}\ (n{==}0)\ \mathsf{then}\ \mathit{flag} := \mathsf{true};\ 1\\ \qquad\qquad\qquad\qquad \mathsf{else}\ \mathit{fact}(n-1) * n\\ \mathsf{in}\ \mathit{fact}(5);\ !\mathit{flag}\end{array}$$

The NCI$^{\mathbb{H}}$ heap is $\mathbb{H}_i = \{\mathit{loc} \mapsto \mathsf{false}\}$ and the environment is $\mathbb{E} = \{\mathit{flag} \mapsto \mathit{loc}, \mathit{fact} \mapsto \lambda_{\mathit{fact}} n. \ldots\}$ just prior to the invocation of $\mathit{fact}(5)$. The APP-PUSH (NON-REC-CALL)$^{\mathbb{H}}$ rule pushes stack frame $[\lambda_{\mathit{fact}} n. \ldots]^{\mathbb{H}_i}_{\mathbb{E}\emptyset}$ to initially invoke $\mathit{fact}(5)$ – note the checkpoint return heap is initially empty since there is no information about it yet. At this point (given a sufficiently precise decision procedure) the else-branch is taken since n is only 5 in this first iteration. There the APP-PRUNE (REC-CALL)$^{\mathbb{H}}$ rule prunes the recursive call, saves the current heap ($\mathbb{H}_i$, which has not changed) as the checkpoint invocation heap (the rec-call cycle heap), and switches the current heap to be the checkpoint return heap, *i.e.* $\emptyset$, since we are now past the recursive call and have switched to the rec-return cycle of Figure 1(a). The heap at the end of the initial run is $\emptyset$, and to re-run from the start we switch back to the rec-call cycle of Figure 1(a) so the heaps need to be swapped again. The current heap is checkpointed in the subscript, and $\mathbb{H}_i$ is pulled out for the rerun. During the rerun both the branches are interpreted in parallel— the "$\mathit{flag} := \mathsf{true}$" assignment updates the heap $\mathbb{H}_i$ to be $\{\mathit{loc} \mapsto \mathsf{true}\}$ at the end of the then-branch, while the heap at the end of the else-branch is still $\emptyset$; the two heaps are merged by IF-MERGE$^{\mathbb{H}}$ resulting in the heap $\{\mathit{loc} \mapsto \mathsf{true}\}$ which is again checkpointed as the return heap in the subscript.

The heaps and the environment stabilize in the next rerun with the stabilized return heap being $\{\mathit{loc} \mapsto \mathsf{true}\}$. Hence, $!\mathit{flag}$ is determined to be true. Note that the stabilized invocation heap remains $\mathbb{H}_i$ — in the CI of the above example the concrete heap is not modified in the rec-call cycle.

Note that all aliases are directly visible in the transitively closed environment, so for example if x and y are aliases to the location loc the closed environment contains $\{x \mapsto \mathit{loc}, y \mapsto \mathit{loc}\}$. Conversely if two variables do *not* map to the same location they must not be aliases in any concrete program run.

A technical report [25] provides the formal details of this extension, including soundness and termination lemmas. Intuitively, stabilization of the heap is guaranteed because only finitely many locations can be added due to reuse of locations in rerun cycles, and heap locations are read/written via variables in the environment, hence stabilization of the heap follows via stabilization of the environment. Also, since NCI$^{\mathbb{H}}$ reuses heap locations during reruns, at most polynomial-time complexity is added.

The notion of an abstract representation of the heap is found in various program analyses, including [24, 16, 7]. Of these only [24] also accurately models cyclic pointer graph structures on the heap, an important step in accurately modeling recursive datatypes and complex object reference graphs. NCI$^{\mathbb{H}}$ extends this to higher-order programs.

4.2 Context-Sensitivity via Context Tagging

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

$$\left(\lambda id.\ id\ (5) + 1;\ id\ (\mathsf{true}) \wedge \mathsf{false}\right)\ (\lambda x.\ x)$$

does not have an NCI: $id\ (\mathsf{true})$ is interpreted as both true and the previously accumulated id application result 5, hence the conjunction $\ldots \wedge \mathsf{false}$ fails. The monotonically increasing environment is necessary to achieve fixed-points of recursive functions, but it limits its 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 NCI$^{\sigma}$, achieved by tagging function bodies with their concrete arguments prior to invocation. So, $\lambda x.\ x$ would be tagged as $\lambda x^5.\ x^5$ before the first invocation and as $\lambda x^{\mathsf{true}}.\ x^{\mathsf{true}}$ prior to the second, resulting in the final environment containing $\{x^5 \mapsto 5,\ x^{\mathsf{true}} \mapsto \mathsf{true}\}$. This environment distinguishes the different calling contexts of id so the above example has an NCI$^{\sigma}$ interpretation. Since the tags are limited to subexpressions of the original program, at most polynomial-time complexity is added.

NCI$^{\sigma}$ 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:

$$\mathsf{let}\ f = \lambda x.\ \lambda y.\ x + y\ \ \mathsf{in}\ \ f\ 0\ 1\,;\ f\ 1\ 0$$

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 $\lambda y^{0^1}.\ x^0 + y^{0^1}$ and $\lambda y^{1^0}.\ x^1 + y^{1^0}$ 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. [25] provides formal details.

4.3 Path-Sensitivity via Path Tagging

In cases when the branching condition is either unknown or resolves to both true and false, NCI 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 '$\mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$', where x may be bound to either true or false at run-time, within e_1 it *must* be true, and within e_2 it *must* be false. Recent work on path sensitivity [13, 15] 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 bound to true, and the e_2 branch is taken iff x is bound to false. We extend NCI with *path tagging* to obtain path-sensitivity in our analysis, yielding the system NCI$^{\mathbb{P}}$. For example, consider the following expression, where inp is user input and unknown statically:

$$\begin{array}{l}\mathsf{let}\ x = (\mathsf{if}\ \mathit{inp}\ \mathsf{then}\ 1\ \mathsf{else}\ \mathsf{true})\ \mathsf{in}\\ \quad \mathsf{if}\ \mathit{inp}\ \mathsf{then}\ x + 2\ \mathsf{else}\ x \wedge \mathsf{false}\end{array}$$

The core NCI analysis gets stuck at $x + 2$ since the environment contains both $x \mapsto 1$ and $x \mapsto \mathsf{true}$ 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 $\mathit{inp} = \mathsf{true}$ then $x = 1$ at $x + 2$, and if $\mathit{inp} = \mathsf{false}$ then $x = \mathsf{true}$ at $x \wedge \mathsf{false}$. Path tagging in NCI$^{\mathbb{P}}$ tags the near-values with the path conditions in force when they are added to the environment, by recording the condition variable and value (true or false) along with the near-value itself. The syntax of the environment is defined to be $\mathbb{E} ::= \left\{x_i \mapsto \overline{\nu_i^{\mathbb{P}_i}}\right\}$ where $\mathbb{P} ::= \left\{\overline{y_i \mapsto b_i}\right\}$, with y_i's being the branch condition variables, is the path-tag or the set of path conditions. In the example, at the end of the first conditional the environment contains $\left\{x \mapsto 1^{\{\mathit{inp} \mapsto \mathsf{true}\}},\ x \mapsto \mathsf{true}^{\{\mathit{inp} \mapsto \mathsf{false}\}}\right\}$. The superscript $\{\mathit{inp} \mapsto \mathsf{true}\}$ on the mapping of x to 1 indicates that this mapping was added in a branch of a

conditional where the conditional variable was inp 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-filter is set to $\{inp^{\mathsf{true}}\}$, meaning that only near-values whose path-tags are subsets of the path-filter are to be considered as having occurred. Since $x \mapsto \mathsf{true}^{\{inp \mapsto \mathsf{false}\}}$ is not allowable as per the path-filter, only $x \mapsto 1^{\{inp \mapsto \mathsf{true}\}}$ is considered when analyzing $x + 2$; correspondingly, the path-filter is set to $\{inp \mapsto \mathsf{false}\}$ when analyzing the else-branch and hence only $x \mapsto \mathsf{true}^{\{inp \mapsto \mathsf{false}\}}$ is considered for $x \wedge \mathsf{false}$.

Note that all the condition variables in path-tags and -filters are statically found in the program, hence the size of $\mathbb{P}$ is bound, adding only polynomial-time complexity. For example in,

```
if x then
    if x1 then ... else ...
else
    if x2 then ... else ...
```

the maximum possible size of the path-tag is 6 *i.e.* 3 condition variables, each with two possible truth values; it represents a union of all possible paths. $\{x \mapsto \mathsf{true}, x_1 \mapsto \mathsf{false}\}$ represents a distinct path through the program, while a path-filter containing, say both $x_1 \mapsto \mathsf{true}$ and $x_1 \mapsto \mathsf{false}$, allows values resulting from either of the corresponding branches (assuming it also contains $x \mapsto \mathsf{true}$) to be used for interpretation.

Semantic Interpretation of NCI$^{\mathbb{P}}$ Environment The NCI$^{\mathbb{P}}$ environment with path tags embodies significant information about the branching structure of programs, and so NCI$^{\mathbb{P}}$ nuggets contain much more useful information than simple NCI nuggets. For example, the NCI$^{\mathbb{P}}$ nugget of $fact(5)$ contains the mappings $\{n \mapsto 5, n \mapsto (n-1)^{\{x \mapsto \mathsf{false}\}}, x \mapsto (n==0)\}$ for n. The conceptual interpretation of mappings with path tags is as follows: $x \mapsto \nu^{\mathbb{P}}$ can be used to generate concrete values for x only when the path conditions in $\mathbb{P}$ are satisfied. So the mapping $n \mapsto (n-1)^{\{x \mapsto \mathsf{false}\}}$ can be used to generate concrete values for n only when x is false. Now x is in turn dependent on the value of n. Hence, we need a notion of "current value" of n for which x is false, and then this current value of n can be used to generate the a value using the mapping $n \mapsto (n-1)^{\{x \mapsto \mathsf{false}\}}$.

The grammar based interpretation used for NCI environments presented in Section 3.2 denotes sets of concrete values for variables with no dependence between them. A more refined semantic interpretation closer to the semantics of CI is required for the NCI$^{\mathbb{P}}$ environment in order to infer stronger inductive properties about the variables in its domain, for example whether $n \geq 0$ for all possible bindings of n given this environment.

Under the new semantic interpretation the NCI$^{\mathbb{P}}$ environment is considered as the set of transition rules of a state transition system $(\mathcal{S}, \longrightarrow_{\mathbb{E}})$, where each state in $\mathcal{S}$ is a concrete environment $\mathcal{E}$ defined as a one-to-one mapping relation between variables and near-values, *i.e.* $\mathcal{E} ::= \{\overline{x_i \mapsto \nu_i}\}$. The state transition system essentially simulates the CI of the original program such that each of its transitions has a corresponding step in CI. A state transition $\mathcal{E} \longrightarrow_{\mathbb{E}} \mathcal{E}'$ occurs iff $x \mapsto \nu^{\mathbb{P}} \in \mathbb{E}$, the path-conditions $\mathbb{P}$ are satisfied in $\mathcal{E}$ and $\mathcal{E}' = \mathcal{E}[x \mapsto \nu]$. The concrete environment update operation $\mathcal{E}[x \mapsto \nu]$ is defined as: a) $\mathcal{E}[x \mapsto \nu] = \mathcal{E} \cup \{x \mapsto \nu\}$ if there is no mapping for x in $\mathcal{E}$, or b) $\mathcal{E}[x \mapsto \nu] = (\mathcal{E} - \{x \mapsto \nu_{old}\}) \cup \{x \mapsto \nu[\nu_{old}/x]\}$ if $x \mapsto \nu_{old} \in \mathcal{E}$.

Consider the above set of mappings for n. Let the initial (empty) state be $\mathcal{E}_0 = \{\}$. Now using the mapping $n \mapsto 5 \in \mathbb{E}$ we can generate the transition $\mathcal{E}_0 \longrightarrow_{\mathbb{E}} \mathcal{E}_1$, where $\mathcal{E}_1 = \{n \mapsto 5\}$. Also we can add $x \mapsto (n == 0) \in \mathbb{E}$ to $\mathcal{E}_1$ such that $\mathcal{E}_1 \longrightarrow_{\mathbb{E}} \mathcal{E}_2$ and $\mathcal{E}_2 = \{n \mapsto 5, x \mapsto (n == 0)\}$. Now $\mathcal{E}_2$ implies that $x \mapsto \mathsf{false}$, hence $n \mapsto (n-1)^{\{x \mapsto \mathsf{false}\}} \in \mathbb{E}$ can be used to generate $\mathcal{E}_3$ such that $\mathcal{E}_2 \longrightarrow_{\mathbb{E}} \mathcal{E}_3$ and $\mathcal{E}_3 = \mathcal{E}_2[n \mapsto (n-1)] = (\mathcal{E}_2 - \{n \mapsto 5\}) \cup \{n \mapsto (5-1)\} = \{n \mapsto (5-1), x \mapsto (n == 0)\}$. Proceeding in this manner recursively reducing n by 1 finally leads to state $\mathcal{E}_7 = \{n \mapsto ((((5-1)-1)-1)-1)-1, x \mapsto (n==0)\}$ such that $\mathcal{E}_3 \longrightarrow^4_{\mathbb{E}} \mathcal{E}_7$. The mapping $n \mapsto (n-1)^{\{x \mapsto \mathsf{false}\}}$ can not be applied to $\mathcal{E}_7$ as $x \mapsto \mathsf{true}$ is implied by $\mathcal{E}_7$. Observe that each of the states from $\mathcal{E}_0$ to $\mathcal{E}_7$ simulates the concrete environment in the CI of $fact(5)$ as the latter recursively invokes itself with $fact(0)$ being the last recursive call. The state transition system for n from $\mathcal{E}_0 \longrightarrow^7_{\mathbb{E}} \mathcal{E}_7$ clearly implies the range of all possible values for n is $[0, 5]$ and that for the condition variable x is $\{\mathsf{true}, \mathsf{false}\}$.

Certain subtleties exist with the interpretation of return values not covered by this approach, but a minor modification described in the technical report addresses the problem [25]. Also, decision procedures for deciding environment properties must be adapted to this new semantic interpretation of the environment. Since the interpretation is based on a simple rewrite system, the problem complexity is roughly similar to the context-free grammar based interpretation of NCI environments. As in that setting, the development of decision procedures for NCI$^{\mathbb{P}}$ is an important topic for future work.

For some programs it could appear that the nugget is not too different than the original source program; the factorial program indeed has much of its structure in its nugget. What is important however is what is removed: there is no function application, and no mutation in the nugget. So, the task of proving properties of programs with higher-order functions and state has been reduced to a far simpler transition system problem.

5. Applications

We now illustrate the generality and applicability of NCI by showing how it can be applied to three quite different static analysis problems: value range analysis, secure information flow, and static enforcement of temporal safety properties.

5.1 Value Range Analysis and Array Bounds Checking

Value range analysis refers to statically inferring the range of possible integer values assignable to a program variable during the course of program computation. Consider the $fact(-5)$ example which recursively assigns values to n starting from -5 and going forever till $-\infty$ — it does so because the terminating branching condition $n==0$ is never satisfied during the computation. On the other hand, in $fact(5)$ the $n == 0$ condition is satisfied during the recursive call $fact(0)$, and so no further assignments to n are made. The core NCI of $fact(5)$ is able to infer that the upper bound on n is 5, but loses its lower bound information since it does not record the correlation between the mappings and the branching condition under which they were added to the environment.

NCI$^{\mathbb{P}}$ however can infer more structure of $fact(5)$. Since it is a path-sensitive analysis as outlined in Section 4.3, it tags all mappings with the branching conditions in force at the point of their addition to the environment. The semantic interpretation of the NCI$^{\mathbb{P}}$ environment, discussed in Section 4.3, also uses these conditions to refine the possible states of the environment. Hence properties such as $0 \leq n \leq 5$ for $fact(5)$ can be automatically verified by NCI$^{\mathbb{P}}$, provided the decision procedure in the semantic interpretation is powerful enough.

Note that static array index bounds analysis is essentially a value range analysis and our system can perform value range analysis without relying on programmer annotations as does [29]. It also applies to higher-order recursive programs unlike [6].

	let $h' = h > 0$ in	let $h' = h > 0$ in
$l := h;$	let $l =$ if (h') then 1	let $l =$ if (h') then 1
$l := 0;$	else 0	else 1
	in l	in l
(a)	(b)	(c)

Figure 8. Examples for Information Flow Analysis

5.2 Information Flow Analysis

Information flow analysis is an important technique for discovering leaks of secure data in software [27, 23]. 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 nugget, 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 8, where we assume l is low and h, h' are high. Consider the stateful program in Figure 8(a). Since heap assignment is flow-sensitive in NCI$^{\mathbb{H}}$, 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 [27, 23] 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 NCI$^{\mathbb{P}}$ has this path information recorded in it already, and so indirect leaks may also be read directly from the nugget, by transitively closing the environment as before but also closing through path-tag variables. Consider the example in Figure 8(b). NCI$^{\mathbb{P}}$ will have $\{h' \mapsto (h > 0), l \mapsto 1^{\{h' \mapsto \mathsf{true}\}}, l \mapsto 0^{\{h' \mapsto \mathsf{false}\}}\}$ in its nugget. Note that the mapping $l \mapsto 1^{\{h' \mapsto \mathsf{true}\}}$ implies an indirect data dependence between l and h', in that $l = 1$ only when $h' = \mathsf{true}$. 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 8(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 $\{h' \mapsto \mathsf{true}, h' \mapsto \mathsf{false}\}$ 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 [18] which will admit 8(a), but this work does not incorporate path-sensitivity and so would fail on 8(c). One recent paper [3] will succeed on 8(c) if the programmer had manually inserted an assertion; our system can do it automatically. Another recent work [19] deals with only simple while-programs and does not incorporate path-sensitivity.

5.3 Enforcement of Temporal Safety Properties

Static enforcement of temporal safety properties has been an area of active research interest [15, 5], especially in languages with side effects where order of operations matters. These temporal properties involve statically tracking runtime state changes, and are also known as *typestate properties* for the case that the temporal property is embodied in a mutable variable. 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 $\lambda x.$ if x then 1 else 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 $isclosed$:

$$
\begin{array}{rcl}
isopen & \triangleq & \lambda x.\, assert(!(x.flag) == \mathsf{open}) \\
isclosed & \triangleq & \lambda x.\, assert(!(x.flag) == \mathsf{close}) \\
open & \triangleq & \lambda x.\, isclosed(x); \_open(x.fptr); x.flag := \mathsf{open} \\
close & \triangleq & \lambda x.\, isopen(x); \_close(x.fptr); x.flag := \mathsf{close} \\
read & \triangleq & \lambda x.\, isopen(x); \_read(x.fptr)
\end{array}
$$

Consider then the code sequence $open(f); process(f); read(f)$, 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 NCI$^{\mathbb{H}}$ if $open$ succeeds since the abstract heap will contain only the mapping $\{p_{flag} \mapsto \mathsf{open}\}$ 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 NCI$^{\mathbb{H}}$ analysis will contain the multimapping $\{p_{flag} \mapsto \mathsf{open}, p_{flag} \mapsto \mathsf{closed}\}$, so the analysis will fail at that point.

Path-sensitivity We now show how the combination of NCI$^{\mathbb{P}}$ and NCI$^{\mathbb{H}}$ can be used to obtain a highly precise path-sensitive analysis of temporal program properties. Consider the following example adapted from [13], where f is some given closed file containing $flag$ value p_{flag} and inp is user input and unknown statically:

```
let flag = ref 0 in
  if inp then flag := true else flag := false;
  if inp then open(f) else ();
  if flag then close(f) else ()
```

The path-sensitive analysis in [13] is not strong enough to detect that $flag$ is a copy of inp, but NCI$^{\mathbb{P}}$ detects the correlation between inp and $flag$ and thus the safety of $close(f)$ in the last part of the expression. Since inp 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, $\{flag \mapsto \mathsf{true}^{\{inp \mapsto \mathsf{true}\}}, flag \mapsto \mathsf{false}^{\{inp \mapsto \mathsf{false}\}}\}$ is part of the environment. Similarly $open(f)$ adds $p_{flag} \mapsto \mathsf{open}^{\{inp \mapsto \mathsf{true}\}}$ to the abstract heap, while the other branch in the conditional results in $p_{flag} \mapsto \mathsf{close}^{\{inp \mapsto \mathsf{false}\}}$. 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 $\{inp \mapsto \mathsf{true}, flag \mapsto \mathsf{true}\}$, restricting the possible mappings to only those whose range is tagged with $inp \mapsto \mathsf{true}$ or $flag \mapsto \mathsf{true}$. Hence, only $p_{flag} \mapsto \mathsf{open}^{\{inp \mapsto \mathsf{true}\}}$ is considered when analyzing $close(f)$, since the mapping $p_{flag} \mapsto \mathsf{close}^{\{inp \mapsto \mathsf{false}\}}$ conflicts with the current path tag. We observe that all examples in [13] can be successfully analyzed using NCI$^{\mathbb{P}}$.

6. Conclusions

We have defined NCI, a powerful and general form of program analysis. It extracts a *nugget* from a program that is a sound, decidable approximation of value bindings resulting from program execution. The nugget can be used to infer nontrivial properties of programs such as the value ranges of variables. It shares the concept of abstract execution with abstract interpretation [9, 10], and additionally incorporates higher-order features. There has been some work on higher-order abstract interpretation [11, 4, 22, 20] but these systems do not produce trace interpretations [8], and so lack precision with respect to temporal properties of programs. We have obtained promising results on some hard analysis problems, as shown in Section 5.

NCI is suited to answering questions that need significant precision and can tolerate somewhat longer running times. We believe it is most appropriate for verifying in higher-order programs the sorts of properties verified by systems such as ESP [15], SLAM [5], recent heap shape analyses [24] and static array bounds check analyses [6]. A particular advantage of our approach in this realm is the way in which higher-order functions and the heap are accurately modeled, and the precision in the analysis due to path- and context-sensitivity.

Probably the most closely related analyses for higher-order programs are type systems. NCI in fact evolved from a type system [26]: it can be viewed roughly as a type constraint and effect system where the effects are themselves sequences of type constraints, and the NCI algorithm is a type closure computation [2] for this type system. Several type-based approaches have been developed which solve problems similar to those addressed by NCI. Type systems can verify temporal safety properties as discussed in Section 5.3 [12, 28, 14, 26], and dependent type systems can assert inductive invariants in types [30] such as the value ranges of Section 5.1. Type systems have also been developed for enforcing information flow security in higher-order settings [23]. An appeal of NCI is that it gives very precise, automatic answers to these problems, in a uniform setting.

6.1 Future Work

This paper is of a foundational nature, focusing on theory rather than implementation. We do have an OCaml implementation combining NCI, NCI$^{\mathbb{H}}$ and NCI$^{\sigma}$ which is available at `http://www.cs.jhu.edu/~pari/nci/`; NCI$^{\mathbb{P}}$ is being implemented. The most important task now is to extend the implementation to more realistic languages and to improve its efficiency.

Another interesting direction for future work is the refinement of decision procedures for interpreting environments. Using a trivial, maximally conservative decision procedure still yields a powerful analysis as shown with information flow analysis and enforcement of temporal safety properties; but the precision of NCI, especially for stronger inductive properties like value range analysis, is directly related to the quality of the decision procedure.

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] R. Bodik, R. Gupta, and V. Sarkar. Abcd: eliminating array bounds checks on demand. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 321–333, New York, NY, USA, 2000. ACM Press.

[7] B.-Y. E. Chang and K. R. M. Leino. Abstract interpretation with alien expressions and heap structures. In *VMCAI*, 2005.

[8] 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.

[9] 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.

[10] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In *IFIP'77: Conference on Formal Description of Programming Concepts*, 1977.

[11] 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.

[12] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 262–275, New York, NY, 1999.

[13] 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.

[14] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI01)*, pages 59–69, 2001.

[15] 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.

[16] 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.

[17] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI'93: Proceedings ACM SIGPLAN conference on Programming Language Design and Implementation*, 1993.

[18] 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.

[19] 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.

[20] N. D. Jones and M. Rosendahl. Higher-order minimal function graphs. In *Journal of Functional and Logic Programming*, 1997.

[21] H. G. Mairson. A simple proof of a theorem of Statman. *Theoretical Computer Science*, 1992.

[22] 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.

[23] F. Pottier and V. Simonet. Information flow inference for ML. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–330, New York, NY, USA, 2002. ACM Press.

[24] T. Reps, M. Sagiv, and R. Wilhelm. Static program analysis via 3-valued logic. In *Proc. Int. Conf. on Computer-Aided Verification*, pages 15–30, 2004.

[25] P. Shroff, S. Smith, and C. Skalka. Near-concrete program interpretation. Technical report, Department of Computer Science, Johns Hopkins Univeristy, 2006. `http://www.cs.jhu.edu/~pari/nci/`.

[26] C. Skalka and S. Smith. History effects and verification. In *APLAS'04: The Second ASIAN Symposium on Programming Languages and Systems*, 2004.

[27] 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.

[28] D. Walker. A type system for expressive security policies. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 254–267, New York, NY, USA, 2000. ACM Press.

[29] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 249–257, New York, NY, USA, 1998. ACM Press.

[30] H. Xi and F. Pfenning. Dependent types in practical programming. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 214–227, New York, NY, 1999.