Value Range Analysis for Higher-Order Programs

Paritosh Shroff
The Johns Hopkins University
pari@cs.jhu.edu

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

Scott F. Smith
The Johns Hopkins University
scott@cs.jhu.edu

Abstract

We develop a value range analysis for higher-order programs. The analysis extracts a *nugget* that characterizes the value bindings resulting from program execution. This abstraction can be fed into a theorem prover to extract non-trivial inductive properties about programs, including the range of values assigned to variables during program execution. The paper incorporates several new technical developments, including a novel *prune-rerun* technique for approximating higher-order recursive functions. The nugget extracted from the analysis condenses higher-order programs into a first-order rule-based system. We show how the nugget can be interpreted as an inductively defined structure, and can be simply and directly encoded in the Isabelle/HOL theorem prover, where nontrivial properties of the program can be verified.

1. Introduction

Program analysis and theorem proving are two well-studied fields. Program analysis excels at automatic calculation of simple low-level properties of production programs, and theorem proving excels at manual or semi-automatic verification of more advanced properties of more idealized programs. Progress is now being made on combining the strengths of these two approaches (Chen and Xi 2005; Gronski et al. 2006; Might 2007). This paper also aims to combine the strengths of program analysis and theorem provers as a framework for automatic enforcement of sophisticated properties of higher-order programs.

Higher-order functional programming is a very powerful programming metaphor, but it is also complex from a program analysis standpoint: the actual low-level operations and the order they take place in is very far removed from the source code. On the other hand it is this simpler low-level view that is easiest for automated methods to apply to. In this paper we focus on defining new forms of sound program abstraction which strip the higher-order nature from functional programs; for any functional program we produce a sound abstraction which we call a *nugget*. Nuggets are simple first-order rewrite systems which contain much of the computational structure of the source program, but with the higher-order and other aspects abstracted. Nuggets can be automatically calculated, and they can also be automatically translated into an equivalent representation in a theorem prover, HOL in particular for this paper. Then, HOL can automatically prove desirable properties of the nugget. Putting these three steps together gives a method for automatically proving properties of programs.

We are defining a general approach which we believe can extend beyond pure functional programs and can also extend to more fine-grained abstractions, but in this paper we focus on solving the *value range* problem – deducing what range of values integer variables can take on. While this is a narrow problem, it is a nontrivial problem and so we believe it is a good testbed for our general approach.

2. Informal Overview

In this section we give an overview of our analysis. We start by showing the form that nuggets take, what they mean, and how properties can be proved about them. Next, we describe informally the nugget construction algorithm. Consider the following simple higher-order program which computes the factorial of 5, using recursion encoded by "self-passing",

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

Our goal is to statically analyze the range of values assignable to the variable n during the course of computation of the above higher-order program. Obviously this program will recurse for n from 5 down to 0, each time with the condition $(n\ != 0)$ holding until finally $n = 0$, and thus the range of values for n is $[0, \ldots, 5]$. The particular goal of this paper is to define an analysis which can infer such ranges about variables for this and more complex programs. For non-recursive functions it is not hard to completely track such ranges; the difficult case is under recursion, where the course of the computation depends on the guards in the conditional branching statements of the function.

2.1 Nuggets

There is a huge array of potential program abstractions: type systems, abstract interpretations, compiler analyses, etc. All of these can be viewed as abstracting away certain properties from program executions. Type systems abstract away all but the type of data; abstract interpretations generally make a finitary abstraction on infinite datatype domains, but preserve all other execution properties. Our approach is not a common point in this space of program abstractions: we wish to *abstract away* the higher-order nature of functional programs, but *preserve* as much of the other behavior as possible, including infinite data types.

The core of our analysis is the *nuggetizer*, which automatically generates *nuggets* from source programs. We are going to begin with a description of the nuggets themselves and how they can be used, and subsequently will discuss the nuggetizing process itself.

Nuggets are purely first-order; they may contain higher-order function data but in the nugget those functions are just atomic data. The higher-order flows that occurred in the original program are reduced to their underlying first-order actions on data.

We will illustrate the form and features of nuggets by considering the nugget produced by the nuggetizer on program (1):

$$\text{Nugget: } \left\{n \mapsto 5, n \mapsto (n-1)^{n\,!=0}\right\} \tag{2}$$

(We are leaving out the trivial mappings for f and $fact$ here.) As can be seen, nuggets are sets of mappings from variables to simple expressions – all of the higher-order functions in program (1) have been expanded. The mapping $n \mapsto 5$ corresponds to the initial value 5 passed in to n in the example. The mapping $n \mapsto (n-1)^{n\,!=0}$ additionally contains a *guard* $n\,! = 0$, which can be viewed as a precondition on when this particular mapping may apply. Also notice this is a recursive mapping: n maps to $n-1$. It corresponds to the fact that $n \mapsto (n-1)$ is the new binding for n upon recursive invocation, which only happens in the case $n\,! = 0$.

The denotational semantics of nuggets There are several equivalent ways to view the meanings of nuggets, which we term their *denotational semantics*. They can be viewed as inductive definitions of the sets of possible values for the variables: the least set of values indicated by the mappings such that the guards hold. They can equivalently be viewed as defining the rules of a context-free generative grammar with dependencies, where variables are the nonterminals of the grammar and data values (integers, booleans, function, *etc.*) and binary operators ($+, -, <, >, ==, \; !=$, *etc.*) comprise the terminals. The above nugget by either approach has the denotation

$$\{n \mapsto 5, n \mapsto 4, n \mapsto 3, n \mapsto 2, n \mapsto 1, n \mapsto 0\}$$

To justify this answer, observe $n \mapsto 0$ does not satisfy the guard $(n\,!= 0)$, and hence, cannot be used in the right side of mapping $n \mapsto (n-1)^{n\,!=0}$, to generate a new mapping for n. The above nugget in fact precisely denotes the range of values assignable to n during the course of computation of program (1).

The key soundness property is, a nugget N for a program p must map each variable x occurring in p to values that are at least the values that may occur at runtime in p (mapping extra values that never occur at runtime may also happen – nuggets are sound but not necessarily complete). With such a soundness property, it means if we prove something for all values in a nugget mapping of x, we have proven it for all values x could take on while running p. Thus the nugget above serves to bound the range of n in the example program to $[0, \ldots, 5]$, which in this case is also complete since n will take on no other values at runtime.

Defining and reasoning about nuggets in Isaballe/HOL Nugget properties can be manually calculated as we did above, but our goal is for more automated proofs of program properties. Since nuggets are nothing more than simple inductive definitions, any nugget may be directly translated into an Isabelle/HOL inductive definition which gives a meaning in Isabelle/HOL that is equivalent to their denotational semantics as outlined above. The Isabelle/HOL theorem prover may then be used to directly prove *e.g.* $0 \leq n \leq 5$ for this example. The Isabelle/HOL encoding of the above nugget is presented in Section 5, as is the general algorithm for translating nuggets into Isabelle/HOL.

A more complex higher-order program To show that the nuggets can account for fancier higher-order recursion, consider a variation of the above program which employs a fixed-point combinator Z to perform recursion. Z is a version of the Y combinator, given by η-expansion on part of it, and can be used in call-by-value evaluation,

$$\mathsf{Z} = \lambda f.\, \big(\lambda x.\, f\ (\lambda y.\, x\ x\ y)\big)\ \big(\lambda x.\, f\ (\lambda y.\, x\ x\ y)\big)$$

$$\begin{array}{l}\mathsf{let}\ f' = \lambda fact.\lambda n.\ \mathsf{if}\ (n\,!= 0)\ \mathsf{then} \\ \qquad\qquad\qquad n * fact\ \ (n-1) \\ \qquad\qquad\quad \mathsf{else} \\ \qquad\qquad\qquad 1 \\ \mathsf{in}\ \mathsf{Z}\ f'\ 5\end{array} \tag{3}$$

The nugget at n as extracted by the nuggetizer is

$$\text{Nugget: } \left\{n \mapsto 5, n \mapsto y, y \mapsto (n-1)^{n\,!=0}\right\} \tag{4}$$

which, by transitive closure, maps n equivalently as in nugget (2). The more complex higher-order structure of the above program proves no more challenging to the nuggetizer.

A program with higher-order mutual recursion Consider another variation of (1), now with higher-order mutual recursion,

$$\begin{array}{l}\mathsf{let}\ g = \lambda fact'.\, \lambda m.\, fact'\ fact'\ (m-1)\ \mathsf{in} \\ \mathsf{let}\ f = \lambda fact.\, \lambda n.\ \mathsf{if}\ (n\,!= 0)\ \mathsf{then} \\ \qquad\qquad\qquad n * g\ fact\ n \\ \qquad\qquad\quad \mathsf{else} \\ \qquad\qquad\qquad 1 \\ \mathsf{in}\ f\ f\ 5\end{array} \tag{5}$$

The nugget at n and m as extracted by the nuggetizer is:

$$\text{Nugget: } \left\{n \mapsto 5, m \mapsto n^{n\,!=0}, n \mapsto (m-1)\right\} \tag{6}$$

The mutually recursive structure between the functions $\lambda n. \ldots$ and $\lambda m. \ldots$ in the above program is reflected as a mutual dependency between the mappings for n and m in the extracted nugget above. The denotational semantics at n and m for the above nugget are

$$\{n \mapsto 5, n \mapsto 4, n \mapsto 3, n \mapsto 2, n \mapsto 1, n \mapsto 0\}, \text{ and}$$
$$\{m \mapsto 5, m \mapsto 4, m \mapsto 3, m \mapsto 2, m \mapsto 1\},$$

respectively. Note that the binding $m \mapsto 0$ is not added because the guard $n\,! = 0$ on the mapping $m \mapsto n^{n\,!=0}$ fails – even though the mapping $n \mapsto 0$ is present, it does not satisfy the guard $(n\,! = 0)$ here.

Dealing with unknown inputs The examples above assume a concrete value such as 5 is flowing into the function. We want to make clear that this value could have also come from some input channel and properties can still be proven. Since we do not have input statements in our language, we only sketch how inputs can be handled. Imagine that $\mathbf{inp}$ contains the input value. Consider the nugget

$$\left\{n \mapsto \mathbf{inp}^{\mathbf{inp} \geq 0}, n \mapsto (n-1)^{n\,!=0}\right\}$$

which reflects a program which invokes the factorial function on under the guard of the conditional $\mathbf{inp} \geq 0$. The denotational semantics for n in this nugget is the range $[0, \ldots, \mathbf{inp}]$ showing that n will never become negative over any program run.

2.2 The Nuggetizer

We now describe the process for creating nuggets, via the *nuggetizer*. The Nuggetizer intuitively constructs the nugget via a collecting semantics – the nugget is incrementally accumulated over an abstract execution of the program. The abstract execution is a form of environment-based operational semantics, wherein the environment collects abstract mappings such as $n \mapsto (n-1)^{n\,!=0}$, and the final environment is the nugget. In fact, the abstract operational

semantics (AOS) structurally aligns very closely with the concrete operational semantics (COS); the AOS however is guaranteed to terminate on all programs. The nuggetizer needs to repeatedly run the AOS until the environment has stabilized.

Collecting the guards The nuggetizer keeps track of the guards that are active at all points of the abstract execution, and tags the abstract mappings with the guards in force at the point of their addition to the environment. So for example, when analyzing the then-branch of the above programs, the nuggetizer tags all mappings with the active guard $(n\,!=0)$, before adding them to the environment. The mappings $n \mapsto (n-1)^{n\,!=0}$, $y \mapsto (n-1)^{n\,!=0}$, and $m \mapsto n^{n\,!=0}$ in nuggets (2), (4) and (6) respectively, are collected by the nuggetizer during abstract execution of the then-branches of corresponding programs; while, the mappings $n \mapsto 5$, $n \mapsto y$ and $n \mapsto (m-1)$ in the above shown nuggets are collected by the nuggetizer when no guards are in force, and hence, the lack of associated guards on them.

The finiteness of abstract environments The domain and range of all mappings added to the environment during the abstract execution, *e.g.* the n, y, m, 5, $(n-1)$, and $(m-1)$ in the above shown nuggets, as well as the guards on those mappings such as $(n\,!=0)$, are fragments that *directly* appear in the corresponding source programs – no new subexpressions are *ever* created, either by substitution or otherwise, during the abstract execution of any program; hence, the maximum number of distinct mappings in the abstract environment of the AOS of any given program is finite. This property is critical for the convergence of the nuggetizer.

An Illustration We now discuss the abstract execution of program (1), placed in a (partial) normal form as follows:

$$\begin{array}{l}\mathsf{let}\ f = \lambda fact.\,\lambda n.\,\mathsf{let}\ r = \mathsf{if}\ (n\,!=0)\ \mathsf{then}\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{let}\ r' = fact\ fact\ (n-1)\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in}\ n * r'\\ \qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad 1\\ \qquad\qquad\qquad\quad \mathsf{in}\ r\\ \mathsf{in}\ f\ f\ 5\end{array} \tag{7}$$

The abstract execution of the above program takes eight steps, and is summarized in Figure 1. The column labeled "Stack" indicates the state of the stack at the corresponding step. The "Collected Mappings" column indicates the mappings collected by the nuggetizer, if any, during the indicated step. The collected mappings are added to the environment of the nuggetizer, and so the environment at any step is the union of all collected mappings up to the current step. The environment is initially empty. The "Current Guard(s)" column indicates the guard(s) in force, if any, at the corresponding step. We now highlight the significant steps in Figure 1.

Setup and forking the if ***branches*** During step 1, The mapping of f to $(\lambda fact.\,\lambda n.\,\mathsf{let}\ r = \ldots\ \mathsf{in}\ r)$ is collected in the environment, and then f is invoked during step 2, which returns immediately with the corresponding nested function $(\lambda n.\,\mathsf{let}\ r = \ldots\ \mathsf{in}\ r)$, resulting in '$(\lambda n.\,\mathsf{let}\ r = \ldots\ \mathsf{in}\ r)\ 5$' being the redex before step 3. At step 4 the abstract execution is forked into two, such that the then- and the else-branches are analyzed in parallel under their corresponding guards, that is, $(n\,!=0)$ and $(n == 0)$, and under the subcolumns labeled 'T' and 'F', respectively. The step 5 under subcolumn labeled 'T', is similar to step 2, except, the collected mapping, $fact \mapsto fact^{n\,!=0}$, is tagged with the current guard, $n\,!=0$, as discussed above.

Pruning recursion Step 6 is crucial in ensuring convergence of the abstract execution – the recursive activation of the function $(\lambda n.\ \ldots)$, which is already on the stack, is *pruned*. The pruning of the function invocation, $(\lambda n.\ldots)\ (n-1)$, involves (a) collecting the mapping, $n \mapsto (n-1)^{n\,!=0}$, which captures the flow of the symbolic argument $(n-1)$, under the guard $n\,!=0$, to the parameter variable n of function $(\lambda n.\ldots)$, and (b) immediately collecting the mapping $r' \mapsto r$, indicating *immediate* return of from the recursion – the function is *not* recursively invoked as happens in the COS. Observe that the environment has not yet collected any mapping for r at this point (in fact no mapping for r appears until step 7), and hence, at this point in the abstract execution it only serves as a *placeholder* for the return value of the recursive call '$(\lambda n.\ldots)\ (n-1)$', to be filled in by later analysis; we say r is *inchoate* as of the end of step 6. Also, the mapping $r' \mapsto r$ is not tagged with any guard, another aspect of the pruning mechanism.

Merging branches and completing Step 7 merges the completed executions of the two branches by collecting the resulting values tagged with their corresponding guards, that is, '$(n * r')^{n\,!=0}$, and '$1^{n==0}$', as mappings $r \mapsto (n * r')^{n\,!=0}$ and $r \mapsto 1^{n==0}$ respectively, denoting the flow of each of the tagged resulting values into the outer let-binding $(\mathsf{let}\ r = (\mathsf{if}\ \ldots)\ \mathsf{in}\ r)$. The redex at the end of step 7 is, then, r. Finally step 8 pops the stack, and the abstract execution terminates.

Environment has a fixed-point The environment at the end of the abstract execution is

$$\begin{array}{l}\{f \mapsto (\lambda fact.\,\lambda n.\ \ldots), fact \mapsto f, n \mapsto 5, fact \mapsto fact^{n\,!=0},\\ n \mapsto (n-1)^{n\,!=0}, r' \mapsto r, r \mapsto (n * r')^{n\,!=0}, r \mapsto 1^{n==0}\}\end{array}$$

and this is the nugget. It is identical to (2) above but with the mappings elided there for simplicity now shown.

The nugget is, in general, the least fixed-point of the symbolic mappings collectable by the AOS for the program (7); this can be ascertained by *re-running* the AOS, but this time using the above environment as the initial environment. At the end of the re-run, the environment will be the same as at the start of execution, *i.e.*, it is a fixed-point. In general, however, the initial run need not yield a fixed-point of the environment.

The need for re-running In the case of higher-order recursive functions where the value returned is itself a function, the return value that is inchoate would be a function if it were present, and since it is *not* present, this returned function will not be executed *at all* in the first pass of the AOS. Here is an example to illustrate this case. Consider the following variation of program (7) where the return value of the function $(\lambda n.\ \ldots)$ is changed to be a function,

$$\begin{array}{l}\mathsf{let}\ f = \lambda fact.\,\lambda n.\,\mathsf{let}\ r = \mathsf{if}\ (n\,!=0)\ \mathsf{then}\\ \qquad\qquad\qquad\qquad\qquad\quad \mathsf{let}\ r' = fact\ fact\ (n-1)\ \mathsf{in}\\ \qquad\qquad\qquad\qquad\qquad\quad \mathsf{let}\ r'' = r'()\ \mathsf{in}\\ \qquad\qquad\qquad\qquad\qquad\quad \lambda y.\,(n * r'')\\ \qquad\qquad\qquad\qquad\quad \mathsf{else}\\ \qquad\qquad\qquad\qquad\qquad\quad \lambda x.\,1\\ \qquad\qquad\qquad\quad \mathsf{in}\ r\\ \mathsf{in}\ f\ f\ 5\ ()\end{array} \tag{8}$$

During the initial run of the AOS on the above program, the return variable r is inchoate in the analysis of the then-branch, as in the previous example. Given the mapping $r' \mapsto r$, r' is thus inchoate as well. Hence, when the redex is '$\mathsf{let}\ r'' = r'()\ \mathsf{in}\ (\lambda y.\,n * r'')$' in the initial run, the environment of the AOS has no known function mapping to r' which can be invoked. The AOS simply skips over the call site '$r'()$' and proceeds without adding any mapping for r'' either. At the merging of the branches the AOS finally adds the mappings $r \mapsto (\lambda y.\,n * r'')^{n\,!=0}$ and $r \mapsto (\lambda x.\,1)^{n==0}$ to the environment, and so r and r' (belatedly) have values. If the AOS is rerun starting with an initial environment being the environment at the end of the just concluded run, the redex '$\mathsf{let}\ r'' =$

Step	*Stack*		*Collected Mappings*		*Current Guard(s)*		*Redex*		*Next Action*	
0							let $f = (\lambda fact.\, \lambda n.\, \ldots)$ in $f\ f\ 5$		collect let-binding	
1			$f \mapsto (\lambda fact.\, \lambda n.\, \ldots)$				$f\ f\ 5$		invoke $(\lambda fact.\, \lambda n.\, \ldots)$	
2			$fact \mapsto f$				$(\lambda n.\, \ldots)\ 5$		invoke $(\lambda n.\, \ldots)$	
3	$(\lambda n.\, \ldots)$		$n \mapsto 5$				let $r = (\text{if } (n\,!=0)\ \ldots)$ in r		fork execution	
	T	F	T	F	T	F	T	F	T	F
4	$(\lambda n.\, \ldots)$	$(\lambda n.\, \ldots)$			$n\,!=0$	$n == 0$	let $r' =$ $fact\ fact\ (n-1)$ in $n * r'$	1	invoke $(\lambda fact.\, \lambda n.\, \ldots)$	nop
5	$(\lambda n.\, \ldots)$	$(\lambda n.\, \ldots)$	$fact \mapsto fact^{n\,!=0}$		$n\,!=0$	$n == 0$	let $r' =$ $(\lambda n.\, \ldots)\ (n-1)$ in $n * r'$	1	prune re-activation of $(\lambda n. \ldots)$	nop
6	$(\lambda n.\, \ldots)$	$(\lambda n.\, \ldots)$	$n \mapsto (n-1)^{n\,!=0}$ $r' \mapsto r$		$n\,!=0$	$n == 0$	$n * r'$	1	merge executions	
7	$(\lambda n.\, \ldots)$		$r \mapsto (n * r')^{n\,!=0}$ $r \mapsto 1^{n==0}$				r		pop	
8							r			

Figure 1. Example: Abstract Execution of Program (7)

$r'()$ in $(\lambda y.\, n * r'')$' will lead to mappings $x \mapsto ()^{n\,!=0}$, $y \mapsto ()^{n\,!=0}$, $r'' \mapsto 1^{n==0}$, and $r'' \mapsto (n * r'')^{n\,!=0}$ to be collected since r and r' are no longer inchoate. The environment at the end of the second run will then be,

$$\begin{array}{l}\{f \mapsto (\lambda fact.\, \lambda n.\, \ldots), fact \mapsto f, n \mapsto 5, fact \mapsto fact^{n\,!=0},\\ n \mapsto (n-1)^{n\,!=0}, x \mapsto ()^{n\,!=0}, y \mapsto ()^{n\,!=0},\\ r' \mapsto r, r \mapsto (\lambda y.\, n * r'')^{n\,!=0}, r \mapsto (\lambda x.\, 1)^{n==0},\\ r'' \mapsto 1^{n==0}, r'' \mapsto (n * r'')^{n\,!=0}\}\end{array}$$

which is the least fixed-point of the symbolic mappings collectable by the nuggetizer for program (8) – the AOS is run one last time by the nuggetizer to verify this fact. As pointed out earlier, the domain, range and guards of the mappings collected by the nuggetizer are all fragments found in the original program, and the growth of the environment is also always increasing, thus it must eventually stop at a fixed point, ...the nugget. The number of re-runs required is dependent on the level of nesting of and dependency on higher-order recursive functions, and should in practice be small since the order of programs is generally small.

We believe this use of the re-run technique in the presence of higher-order recursive programs is a novel insight of our analysis.

Value range of return values The core analysis tracks function argument values well, but loses information on values returned from recursive functions. The nugget for the return variable r of the function $(\lambda n.\, \ldots)$ in the above environment for program (7) is,

$$\begin{array}{l}\{n \mapsto 5, n \mapsto (n-1)^{n\,!=0},\\ r \mapsto 1^{n==0}, r \mapsto (n * r)^{n\,!=0}\}\end{array} \quad (9)$$

Note, the mapping $r' \mapsto r$ is inlined into mapping $r \mapsto (n * r')^{n\,!=0}$ for simplicity of presentation. Observe how r in the range of the mapping $r \mapsto (n * r)^{n\,!=0}$ lacks a guard, in effect allowing *any* known value of r to be multiplied with *any* known value of n (besides 0) to generate a new value for r. The denotational semantics at n and r of the above nugget is

$$\{n \mapsto 5, n \mapsto 4, n \mapsto 3, n \mapsto 2, n \mapsto 1, n \mapsto 0\}, \text{ and}$$

$$\begin{array}{l}\{r \mapsto 1, r \mapsto 2, r \mapsto 6, r \mapsto 24, r \mapsto 120,\\ r \mapsto 5, r \mapsto 8, r \mapsto 18, r \mapsto 48, r \mapsto 600, \ldots\},\end{array}$$

which is sound but not complete: $r \mapsto 5$ because $n \mapsto 5$ and $r \mapsto 1$ are present, but 5 is obviously not in the range of a factorial function. The correlation between the argument and return values of recursive function invocations is not captured by the nuggetizer due to the pruning of all re-activations of any function, as shown in Step 6 of Figure 1 for $(\lambda n.\, \ldots)$; hence, precision of the analyzed return value of the variable r is lost. The nuggetizer can in fact be extended to capture the above mentioned dependency between the argument and return values of recursive functions, and thus perform a precise analysis on the range of return values as well; this extension is presented in Section 6.

Incompleteness To better show the scope of the analysis, we given an example of an incomplete nugget which no extension will handle. This program is inspired by a bidirectional bubble sort.

$$\begin{array}{l}\text{let } f = \lambda sort.\, \lambda x.\, \lambda limit.\, \text{if } (x < limit) \text{ then}\\ \qquad\qquad\qquad\qquad sort\ sort\ (x+1)\ (limit-1)\\ \qquad\qquad\qquad\quad \text{else}\\ \qquad\qquad\qquad\qquad 1\\ \text{in } f\ f\ 0\ 9\end{array} \quad (10)$$

The nugget at x and $limit$ as extracted by the nuggetizer is,

$$\begin{array}{l}\{x \mapsto 0, x \mapsto (x+1)^{x\,<\,limit},\\ limit \mapsto 9, limit \mapsto (limit-1)^{x\,<\,limit}\}\end{array}$$

and their corresponding denotational semantics are,

$$\{x \mapsto 0, x \mapsto 1, \ldots, x \mapsto 9, x \mapsto 9\}, \text{ and}$$

$$\{limit \mapsto 9, limit \mapsto 8, \ldots, limit \mapsto 1, limit \mapsto 0\}$$

respectively; while the exact ranges of values assigned to x and $limit$ during the computation of the above program are $[0, 5]$ and $[4, 9]$ respectively. The nuggetizer does not record the correlation between the order of assignments to x and $limit$ in the computation of the above program, that is, the fact that the assignment of $(x+1)$ to x is immediately followed by the assignment of $(limit - 1)$ to $limit$, and vice-versa. Note, however, that the analysis still manages to bound x to a narrow range – if x had been used as an index into an array of length 10, the above nugget would prove that all accesses to such an array would be in-bounds.

$$
\begin{array}{lll}
b & ::= \mathsf{true} \mid \mathsf{false} & \textit{boolean} \\
\oplus_i & ::= + \mid - \mid * \mid / & \textit{binary operator (ints)} \\
 & \quad \mid \ == \mid != \mid < \mid > & \\
\oplus_b & ::= \wedge \mid \vee & \textit{binary operator (bools)} \\
\oplus & ::= \oplus_i \mid \oplus_b & \textit{binary operator} \\
\mathcal{F} & ::= \lambda x.\, p & \textit{function} \\
\eta & ::= x \mid i \mid b \mid \mathcal{F} \mid x \oplus x & \textit{lazy value} \\
\kappa & ::= \eta \mid \mathsf{if}\ x\ \mathsf{then}\ p\ \mathsf{else}\ p \mid x\ x & \textit{atomic computation} \\
p & ::= x \mid \mathsf{let}\ x = \kappa\ \mathsf{in}\ p & \textit{A-normal program} \\
 & \quad \langle \eta, \mathrm{E} \rangle & \textit{concrete closure} \\
\mathrm{E} & ::= \{\overline{x \mapsto \langle \eta, \mathrm{E} \rangle}\} & \textit{concrete environment}
\end{array}
$$

Figure 2. Language: Syntax Grammar

$$
\begin{array}{lll}
P & ::= b \mid \eta = \eta \mid P \wedge P \mid P \vee P & \textit{predicate} \\
 & \quad \langle \eta, P \rangle & \textit{abstract closure} \\
\mathcal{E} & ::= \{\overline{x \mapsto \langle \eta, P \rangle}\} & \textit{abstract environment} \\
\mathcal{S} & ::= \{\overline{\langle \mathcal{F}, P \rangle}\} & \textit{abstract "stack"}
\end{array}
$$

Figure 4. AOS: Syntax Grammar

3. The Language Model and Semantics

This section details the language and the concrete operational semantics, the COS. Our programming language model, Figure 2, is a pure higher-order functional language with arithmetic, booleans, conditional branching and let-bindings, with variables x and integers i. The grammar assumes expressions are already in A-normal form (Flanagan et al. 1993), so each program point has an associated program variable. This grammar thus reflects an "internal" language which the original source program has been translated into. Since the differences between the original source and this A-normal form are minor we have left out the original source language. $\langle \eta, \mathrm{E} \rangle$ represents a closure, for η a (lazy) value (discussed below), and E an environment. Figure 3 gives an environment based concrete operational semantics (COS) for our language. The semantics is *mixed-step*, that is, a combination of both small- (in the `let` rule) and big-step (in the `if` and `app` rules) reductions. The big-step semantics is used in the `app` rule to clearly demarcate the scope of function invocations, eliminating the need for a stack. The big-step semantics in the `if` rule serves mainly to align it with the corresponding rule in the abstract operational semantics (presented later). In general, the mixed-step semantics is used to align with the AOS and hence, to facilitate the proof of soundness by a direct simulation argument between the two; neither all-big nor all-small step will give an elegant alignment. The mixed-step reduction relation $\longrightarrow$ is defined over configurations, which are tuples, (E, p); while $\longrightarrow^n$ is the n-step reflexive (if $n = 0$) and transitive (otherwise) closure of $\longrightarrow$.

The environment lookup function on variables is the partial function defined as, $\mathrm{E}(x) = \langle \eta', \mathrm{E}' \rangle$ iff $x \mapsto \langle \eta', \mathrm{E}' \rangle$ is the only binding for x in E. The transitively closed λ-lookup function on variables and function values is inductively defined as, $\mathrm{E}(x)^{+\lambda} = \mathrm{E}'(\eta')^{+\lambda}$ iff $\mathrm{E}(x) = \langle \eta', \mathrm{E}' \rangle$, and $\mathrm{E}(\mathcal{F})^{+\lambda} = \langle \mathcal{F}, \mathrm{E} \rangle$ respectively.

The arithmetic, relational and logical operations ($x \oplus x$) are evaluated in a maximally lazy fashion. So for example, the reduction of the abstract value $x + y$ given its environment $\{x \mapsto \langle 1, \emptyset \rangle, y \mapsto \langle 2, \emptyset \rangle\}$, to integer 3, is postponed until it is essential to do so for the computation to proceed, that is, the branching condition of the `if` rule needs to be resolved. In the meantime it is stored in the environment as $\langle x + y, \{x \mapsto \langle 1, \emptyset \rangle, y \mapsto \langle 2, \emptyset \rangle\} \rangle$. We term them *lazy values* for this reason. The environment lookup is also maximally lazy in that mappings are only taken out of the environment and used when critically needed, as can be seen from this example. For this reason, every lazy value, not just functions, are closures.

The following total function reduces a closure to its smallest equivalent form, or we say "grounds" it.

Definition 3.1 (Ground of a Concrete Closure). $[\![\langle \eta, \mathrm{E} \rangle]\!]$ *is a total function inductively defined as,*

1. $[\![\langle x, \mathrm{E} \rangle]\!] = [\![\mathrm{E}(x)]\!]$*; and,* $[\![\langle i, \mathrm{E} \rangle]\!] = \langle i, \emptyset \rangle$*; and,* $[\![\langle b, \mathrm{E} \rangle]\!] = \langle b, \emptyset \rangle$*; and,* $[\![\langle \mathcal{F}, \mathrm{E} \rangle]\!] = \langle \mathcal{F}, \mathrm{E}' \rangle$*, where* $free(\mathcal{F}) \cap dom(\mathrm{E}) = \{\overline{x_k}\}$ *and* $\mathrm{E}' = \{\overline{x_k \mapsto [\![\langle x_k, \mathrm{E} \rangle]\!]}\}$*; and,*
2. $[\![\langle x_1 \oplus_i x_2, \mathrm{E} \rangle]\!] = \langle \eta, \emptyset \rangle$*, if* $[\![\langle x_1, \mathrm{E} \rangle]\!] = \langle i_1, \emptyset \rangle$*,* $[\![\langle x_2, \mathrm{E} \rangle]\!] = \langle i_2, \emptyset \rangle$*, and* $i_1 \oplus_i i_2 = \eta$*, where* η *is either integral or boolean; else,* $[\![\langle x_1 \oplus_i x_2, \mathrm{E} \rangle]\!] = \langle x_1 \oplus_i x_2, \mathrm{E}' \rangle$*, where* $\mathrm{E}' = \{x_1 \mapsto [\![\langle x_1, \mathrm{E} \rangle]\!], x_2 \mapsto [\![\langle x_2, \mathrm{E} \rangle]\!]\}$*; and,*
3. $[\![\langle x_1 \oplus_b x_2, \mathrm{E} \rangle]\!] = \langle b, \emptyset \rangle$*, if* $[\![\langle x_1, \mathrm{E} \rangle]\!] = \langle b_1, \emptyset \rangle$*,* $[\![\langle x_2, \mathrm{E} \rangle]\!] = \langle b_2, \emptyset \rangle$*, and* $b_1 \oplus_b b_2 = b$*; else,* $[\![\langle x_1 \oplus_b x_2, \mathrm{E} \rangle]\!] = \langle x_1 \oplus_b x_2, \mathrm{E}' \rangle$*, where* $\mathrm{E}' = \{x_1 \mapsto [\![\langle x_1, \mathrm{E} \rangle]\!], x_2 \mapsto [\![\langle x_2, \mathrm{E} \rangle]\!]\}$.

A closure $\langle p, \mathrm{E} \rangle$ is considered *closed* iff $free(p) \subseteq dom(\mathrm{E})$ and all $\langle \eta, \mathrm{E}' \rangle$ in the range of E are, in turn, closed.

The following defines the *canonical* constructs in our language model, the well-formed constructs we will restrict reduction to. (Here, $locals(p)$, formally defined in Appendix A, is the set of all local variables in program p, and $dom(\mathrm{E})$ denotes the domain of E.)

Definition 3.2 (Canonical Constructs). *1. (Canonical Program). A program p is canonical iff each of its local variables is distinct.*

2. *(Canonical Environment). An environment* E *is canonical iff* E *is a set of single-valued mappings, and all $\langle \eta', \mathrm{E}' \rangle$ in the range of* E *are, in turn, canonical.*
3. *(Canonical Closure). A closure $\langle p, \mathrm{E} \rangle$ is canonical iff p and* E *are each canonical, and $locals(p) \cap dom(\mathrm{E}) = \emptyset$.*

The condition $locals(p) \cap dom(\mathrm{E}) = \emptyset$ for a canonical closure above allows new mappings to be simply appended to the environment in the semantics rules in Figure 3, as opposed to overwriting any previous bindings – by keeping variables distinct enough we can reduce the amount of renaming needed in the COS rules to zero.

Definition 3.3 (Well-Formed Configuration (E, p)). *A configuration (E, p) is well-formed iff the closure $\langle p, \mathrm{E} \rangle$ is closed and canonical.*

The "deep" containment relation is a technical definition used below; it is inductively defined as, $x \mapsto \langle \eta', \mathrm{E}' \rangle \in \mathrm{E}$ iff either, $x \mapsto \langle \eta', \mathrm{E}' \rangle \in \mathrm{E}$, or $\exists \langle \eta'', \mathrm{E}'' \rangle \in range(\mathrm{E}).\ x \mapsto \langle \eta', \mathrm{E}' \rangle \in \mathrm{E}''$.

The partial function $next(\mathsf{let}\ x = \kappa\ \mathsf{in}\ p) = p$, returns the next redex in p. We write $next^n()$ as shorthand for n successive applications of $next()$. The *length* of a program is defined as $len(x) = 0$, and $len(\mathsf{let}\ x = \kappa\ \mathsf{in}\ p) = 1 + len(p)$. The variable serving as a placeholder for the final result value of a program is defined as $\lfloor p \rfloor = next^n(p)$, where $n = len(p)$. The following technical properties of the COS reflect the precise structure of configurations under reduction, and are needed to establish the COS-AOS correspondence in the next section.

Lemma 3.4 (Properties of COS).

1. *(Invariants).*
   (a) *(1-step). If $(\mathrm{E}, p) \longrightarrow (\mathrm{E}', p')$ then $\mathrm{E} \subseteq \mathrm{E}'$ and $next(p) = p'$.*
   (b) *(n-step). If $(\mathrm{E}, p) \longrightarrow^n (\mathrm{E}_n, r)$ then $\mathrm{E} \subseteq \mathrm{E}_n$, $\lfloor p \rfloor = r$ and $len(p) = n$.*

$$\frac{}{(\mathrm{E}, \mathsf{let}\ x = \eta\ \mathsf{in}\ p) \longrightarrow \big(\mathrm{E} \cup \{x \mapsto \langle \eta, \mathrm{E}\rangle\}, p\big)}\texttt{let} \qquad \frac{[\![\langle x, \mathrm{E}\rangle]\!] = \langle b_i, \emptyset\rangle \qquad (b_1, b_2) = (\mathsf{true}, \mathsf{false}) \qquad (\mathrm{E}, p_i) \longrightarrow^n (\mathrm{E}', y')}{\big(\mathrm{E}, \mathsf{let}\ y = (\mathsf{if}\ x\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2)\ \mathsf{in}\ p\big) \longrightarrow \big(\mathrm{E}' \cup \{y \mapsto \langle y', \mathrm{E}'\rangle\}, p\big)}\texttt{if}$$

$$\frac{\mathrm{E}(f)^{+\lambda} = \langle \lambda x.\, p, \mathrm{E}_f\rangle \qquad \big(\mathrm{E}_f \cup \{x \mapsto \langle x', \mathrm{E}\rangle\}, p\big) \longrightarrow^n (\mathrm{E}', r')}{(\mathrm{E}, \mathsf{let}\ r = f\ x'\ \mathsf{in}\ p_{next}) \longrightarrow \big(\mathrm{E} \cup \{r \mapsto \langle r', \mathrm{E}'\rangle\}, p_{next}\big)}\texttt{app}$$

Figure 3. Concrete Operational Semantics (COS) Rules

$$\frac{}{\big(\mathcal{S}, \mathcal{E}, P, \mathsf{let}\ x = \eta\ \mathsf{in}\ p\big) \longrightarrow \Big(\mathcal{S}, \mathcal{E} \cup \big\{x \mapsto \langle \eta, P\rangle\big\}, P \wedge (x = \eta), p\Big)}\textit{let}$$

$$\frac{\begin{array}{c}P_1 = P \wedge (x = \mathsf{true}) \qquad P_2 = P \wedge (x = \mathsf{false}) \\ \big(\mathcal{S}, \mathcal{E}, P_1, p_1\big) \longrightarrow^{n_1} \big(\mathcal{S}, \mathcal{E}_1, P_1', y_1'\big) \qquad \big(\mathcal{S}, \mathcal{E}, P_2, p_2\big) \longrightarrow^{n_2} \big(\mathcal{S}, \mathcal{E}_2, P_2', y_2'\big) \qquad P' = \big(P_1' \wedge (y = y_1')\big) \vee \big(P_2' \wedge (y = y_2')\big)\end{array}}{\Big(\mathcal{S}, \mathcal{E}, P, \mathsf{let}\ y = (\mathsf{if}\ x\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2)\ \mathsf{in}\ p\Big) \longrightarrow \Big(\mathcal{S}, \mathcal{E}_1 \cup \big\{y \mapsto \langle y_1', P_1'\rangle\big\} \cup \mathcal{E}_2 \cup \big\{y \mapsto \langle y_2', P_2'\rangle\big\}, P', p\Big)}\textit{if}$$

$$\frac{\begin{array}{c}\mathcal{E}(f)^{+\lambda} = \{\overline{\langle \mathcal{F}_k, P_k\rangle}\} \qquad \overline{\mathcal{F}_k = \lambda x_k.\, p_k} \\ \forall 1 \le i \le k \qquad \mathit{CALL}\big(\mathcal{S}, \langle \mathcal{F}_i, P_i\rangle\big) = p_i' \qquad \mathcal{S}_i = \mathcal{S} \cup \{\langle \mathcal{F}_i, P_i\rangle\} \qquad \Big(\mathcal{S}_i, \mathcal{E} \cup \big\{x_i \mapsto \langle x', P\rangle\big\}, P_i, p_i'\Big) \longrightarrow^{n_i} \big(\mathcal{S}_i, \mathcal{E}_i, P_i', r_i'\big)\end{array}}{\big(\mathcal{S}, \mathcal{E}, P, \mathsf{let}\ r = f\ x'\ \mathsf{in}\ p_{next}\big) \longrightarrow \Big(\mathcal{S}, \mathcal{E} \cup \bigcup_{1 \le i \le k} \mathcal{E}_i \cup \big\{r \mapsto \langle r_i', P_i'\rangle\big\}, P, p_{next}\Big)}\textit{app}$$

Figure 5. Abstract Operational Semantics (AOS) Rules

2. *(Preservation of Well-Formedness). If* (E, p) *is well-formed, and* $(\mathrm{E}, p) \longrightarrow^n (\mathrm{E}_n, p_n)$ *then* (E_n, p_n) *is well-formed as well.*
3. *(Preservation of Bindings). If* (E', p') *is a node in the derivation tree of* $(\mathrm{E}, p) \longrightarrow^n (\mathrm{E}_n, p_n)$ *and* $x \mapsto \langle \eta, \mathrm{E}_{clo}\rangle \in \mathrm{E}'$ *then* $x \mapsto \langle \eta, \mathrm{E}_{clo}\rangle \notin \mathrm{E}_n$.
4. *(Fixed-Point of Subexpressions). If* (E', p') *is a node in the derivation tree of* $(\emptyset, p) \longrightarrow^n (\mathrm{E}_n, p_n)$ *then* p' *is a subexpression of of* p*, and for all* $x \mapsto \langle \eta, \mathrm{E}_{clo}\rangle \notin \mathrm{E}'$*,* x *and* η *are subexpressions of* p *as well.*

Proof. Follows by induction on the respective derivations. □

4. The AOS and Nuggetizer

Figure 4 defines additional syntax needed for the AOS. The abstract environment $\mathcal{E}$ is a set of mappings from variables to abstract closures; it may map the same variable multiple times due to imprecision in the execution. Unlike the COS case, abstract closures $\langle \eta, P\rangle$ do not come with full environments but with highly restricted forms, predicates P which are simple propositional formulae. The P was informally called the "guard" in Section 2, and notated slightly differently: the $n \mapsto (n-1)^{n\,!=0}$ there for example is formally $n \mapsto \langle (n-1), n\,!=0\rangle$ here. By replacing the concrete E with an abstract P in closures, the abstract closures are significantly weaker, but the global abstract environment $\mathcal{E}$ is on the other hand strengthened: it is the sole variable mapping in the AOS. Thus the nested concrete environments have been flattened to a single global abstract environment. The abstract "stack" $\mathcal{S}$ is a set of abstract function closures; this stack is not used as a normal reduction stack, it is only used by the AOS to detect and prune recursive calls.

Figure 5 presents the AOS rules; observe how the AOS rules structurally align with the COS rules of Figure 3. AOS reduction $\longrightarrow$ is defined over configurations which are 4-tuples, $(\mathcal{S}, \mathcal{E}, P, p)$, adding a predicate P and stack $\mathcal{S}$ to the COS state elements. The P in abstract configurations indicate the constraints in force *right now*, for the current function activation only.

The transitively closed λ-lookup function on variables, $\mathcal{E}(x)^{+\lambda}$, is inductively defined to be the smallest set $\{\overline{\langle \mathcal{F}_k, P_k\rangle}\}$, such that $\forall x \mapsto \langle y, P\rangle \in \mathcal{E}.\ \mathcal{E}(y)^{+\lambda} \subseteq \{\overline{\langle \mathcal{F}_k, P_k\rangle}\}$, and $\forall x \mapsto \langle \mathcal{F}, P\rangle \in \mathcal{E}.\ \langle \mathcal{F}, P\rangle \in \{\overline{\langle \mathcal{F}_k, P_k\rangle}\}$.

4.1 The AOS Rules

We now elaborate on the AOS rules.

let The *let* rule collects the let-binding in the abstract closure as $x \mapsto \langle \eta, P\rangle$, analogous to the mapping of x in the concrete closure rule. The P is the guard on the new mapping of x here, reflecting the constraints in the current execution context. The predicate P is also updated to reflect the just executed let-assignment by conjoining $(x = \eta)$, because this is a new constraint hereafter in force in this function activation. This equality predicate was not shown in any guards in Section 2 for simplicity of presentation; it was was not required to obtain a precise analysis on examples presented. To show why it is helpful to add these equations, consider the following variation of program (1):

$$\begin{array}{l}\mathsf{let}\ f = \lambda fact.\, \lambda n.\, \mathsf{let}\ z = n\ \mathsf{in} \\ \qquad\qquad\qquad\quad \mathsf{if}\ (n\,!=0)\ \mathsf{then} \\ \qquad\qquad\qquad\quad\ \ n * fact\ fact\ (z-1) \\ \qquad\qquad\qquad\quad \mathsf{else} \\ \qquad\qquad\qquad\quad\ \ 1 \\ \mathsf{in}\ f\ f\ 5\end{array} \tag{11}$$

(For simplicity, the above program is not completely satisfying the grammar of our language.) The nuggetizer produces the following nugget at n and z:

$$\text{Nugget: } \big\{n \mapsto 5, z \mapsto n, n \mapsto (z-1)^{n\,!=0\ \wedge\ z=n}\big\} \tag{12}$$

Note the conjunct $(z = n)$ in the guard of the mapping, $n \mapsto (z-1)^{n\,!=0\,\wedge\,z=n}$, which ties together the values of z and n. Without this addition, the guard would miss the correlation and the nugget would lose precision.

if The *if* rule performs abstract execution of the then- and else-branches in parallel under the current predicate appended with their respective guards, as discussed in Section 2.2, and merges their resulting environments, values, and predicates. Observe how the then-case has $x = \mathsf{true}$ added to P, indicating the current context of execution proceeds under this additional constraint, and similarly for the else-case. The final predicate P' in the *if* rule logically expresses the current constraints to be either the then- or the else-case predicate, and the final value y to be either the then-value y'_1 or the else-value y'_2.

app The *app* rule performs abstract execution of all possible function invocations at the corresponding call-site under their respective predicates in parallel (recall that $\mathcal{E}$ may map a variable multiply), and merges their resulting environments and values. Observe various analogies between the app and *app* rules – for example, the app rule pulls concrete environment E_f from the concrete closure of the corresponding function being invoked, while the *app* rule pulls predicate P_i from the corresponding abstract closure. $\textsc{call}\big(\mathcal{S}, \langle \mathcal{F}, P\rangle\big)$ here is a function returning the redex to be executed when invoking the abstract function closure $\langle \mathcal{F}, P\rangle$ given the abstract stack $\mathcal{S}$. In the case that it is not a recursive call, the body of $\mathcal{F}$ is returned, but in the case of the recursive call we need to *prune* at this point, as discussed in Section 2.2, and the answer is just a placeholder for the return value. The formal definition is as follows: for $\mathcal{F} = \lambda x.\, p$, $\textsc{call}\big(\mathcal{S}, \langle \mathcal{F}, P\rangle\big) = p$ if $\langle \mathcal{F}, P\rangle \notin \mathcal{S}$, and $\textsc{call}\big(\mathcal{S}, \langle \mathcal{F}, P\rangle\big) = \lfloor p \rfloor$ if $\langle \mathcal{F}, P\rangle \in \mathcal{S}$. When $\mathcal{E}(f)^{+\lambda} = \emptyset$ we say f is *inchoate* in $\mathcal{E}$ since it is undefined, and the *app* rule simply skips over the call-site and steps the AOS over to p_{next}; as discussed in Section 2.2 this skipping over call-sites is sound from the point of view of the nuggetizer as later steps will fill in the appropriate values which will then be analyzed in later rerun(s). This case was illustrated in example (8) of Section 2.2, where variable r' was inchoate in the initial run due to prior pruning of the re-activation of function $(\lambda n.\ \ldots)$, but attains a value in the next run.

4.2 Properties

We now formally prove some properties of the AOS, and define and show the nuggetizer is computable and sound. We start by extending the partial function $next$ to $next(P, p)$ which both gives the next redex and the updated P in force at that point.

Definition 4.1 ($next(P, p)$).

1. $next(P, \mathsf{let}\ x = \eta\ \mathsf{in}\ p) = (P', p)$, *where* $P' = P \wedge (x = \eta)$.
2. $next\big(P, \mathsf{let}\ y = (\mathsf{if}\ x\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2)\ \mathsf{in}\ p\big) = (P', p)$, *where* $P_1 = P \wedge (x = \mathsf{true})$, $P_2 = P \wedge (x = \mathsf{false})$, $n_1 = len(p_1)$, $n_2 = len(p_2)$, $next^{n_1}(P_1, p_1) = (P'_1, y'_1)$, $next^{n_2}(P_2, p_2) = (P'_2, y'_2)$, *and* $P' = \big(P'_1 \wedge (y = y'_1)\big) \vee \big(P'_2 \wedge (y = y'_2)\big)$.
3. $next(P, \mathsf{let}\ r = f\ x\ \mathsf{in}\ p) = (P, p)$.

$\lfloor (P, p) \rfloor$ is then defined as $\lfloor (P, p) \rfloor = next^n(P, p)$, where $n = len(p)$.

Basic properties of the AOS include the following. The most important property is monotonicity: abstract environments only add mappings over the course of computation, a critical change from the concrete to the abstract semantics.

Lemma 4.2 (Properties of AOS).

1. *(Determinism). If* $(\mathcal{S}, \mathcal{E}, P, p) \longrightarrow (\mathcal{S}', \mathcal{E}', P', p')$ *and additionally* $(\mathcal{S}, \mathcal{E}, P, p) \longrightarrow (\mathcal{S}'', \mathcal{E}'', P'', p'')$ *then* $(\mathcal{S}', \mathcal{E}', P', p') = (\mathcal{S}'', \mathcal{E}'', P'', p'')$.
2. *(Monotonicity). If* $(\mathcal{S}', \mathcal{E}', P', p')$ *is a node in the derivation tree of* $(\mathcal{S}, \mathcal{E}, P, p) \longrightarrow (\mathcal{S}'', \mathcal{E}'', P'', p'')$ *then* $\mathcal{S} \subseteq \mathcal{S}'$ *and* $\mathcal{E} \subseteq \mathcal{E}' \subseteq \mathcal{E}''$.
3. *(Invariants).*
   (a) *(1-step). If* $(\mathcal{S}, \mathcal{E}, P, p) \longrightarrow (\mathcal{S}', \mathcal{E}', P', p')$ *then* $\mathcal{S} = \mathcal{S}'$, $\mathcal{E} \subseteq \mathcal{E}'$ *and* $next(P, p) = (P', p')$.
   (b) *(n-step). If* $(\mathcal{S}, \mathcal{E}, P, p) \longrightarrow^n (\mathcal{S}_n, \mathcal{E}_n, P_n, x_n)$ *then* $\mathcal{S} = \mathcal{S}_n$, $\mathcal{E} \subseteq \mathcal{E}_n$, $\lfloor (P, p) \rfloor = (P_n, x_n)$ *and* $len(p) = n$.

Proof. Follows by induction on the respective derivations. □

The following lemma states that the AOS has a derivation for all inputs. We are not concerned now with catching type errors, and having this property simplifies the mathematics.

Lemma 4.3 (Existence of Derivation). *For all 4-tuples* $(\mathcal{S}, \mathcal{E}, P, p)$ *there exists* $\mathcal{S}'$, $\mathcal{E}'$, P', r' *and* n, *such that* $(\mathcal{S}, \mathcal{E}, P, p) \longrightarrow^n (\mathcal{S}', \mathcal{E}', P', r')$.

The proofs of the this and all subsequent Lemmas can be found in the attached document.

We now formalize the nuggetizer and prove its soundness. We start by defining the nugget and nuggetizer.

Definition 4.4 (Nugget). *The nugget of a 3-tuple* $(\mathcal{E}, P, p)$ *is the the smallest set* $\mathcal{E}'$ *such that* $(\emptyset, \mathcal{E}, P, p) \longrightarrow^n (\emptyset, \mathcal{E}_n, P_n, r)$, *for some* n, P_n *and* r, *and either* $\mathcal{E} = \mathcal{E}_n = \mathcal{E}'$, *or inductively,* $\mathcal{E}'$ *is the nugget of 3-tuple* $(\mathcal{E}_n, P, p)$.

Since this is a monotone inductive definition the nugget always exists, but it could in theory require infinitely many re-runs. We will show below that is not the case. The nuggetizer is then defined as the function that builds a nugget starting from an empty environment. We will show below that the nuggetizer is a computable function.

Definition 4.5 (The Nuggetizer). $nuggetizer(p) = \mathcal{E}$, *where* $\mathcal{E}$ *is the nugget of 3-tuple* $(\emptyset, \mathsf{true}, p)$.

We need to show the AOS simulates the COS, so we need to relate concrete and abstract closures.

Definition 4.6 (Predicate Satisfaction Relation: $\mathrm{E} \vdash P$).

1. $\mathrm{E} \vdash \mathsf{true}$;
2. $\mathrm{E} \vdash \eta_1 = \eta_2$, *iff* $[\![\langle \eta_1, \mathrm{E}\rangle]\!] = [\![\langle \eta_2, \mathrm{E}\rangle]\!]$;
3. $\mathrm{E} \vdash P \wedge P'$, *iff* $\mathrm{E} \vdash P$ *and* $\mathrm{E} \vdash P'$;
4. $\mathrm{E} \vdash P \vee P'$, *iff either,* $\mathrm{E} \vdash P$, *or* $\mathrm{E} \vdash P'$.

The denotational semantics of the abstract environment may now be defined as

Definition 4.7 (Denotational Semantics of $\mathcal{E}$: $[\![\mathcal{E}]\!]$). $[\![\mathcal{E}]\!]$ *is the smallest set* E *such that,*

1. $x \mapsto \langle \eta, \emptyset\rangle \in \mathrm{E}$, *if* $x \mapsto \langle \eta, P\rangle \in \mathcal{E}$, $\emptyset \vdash P$, *and* $\langle \eta, \emptyset\rangle$ *is closed; and,*
2. $x \mapsto \langle \eta', \mathrm{E}'\rangle \in \mathrm{E}$, *if* $x \mapsto \langle \eta', P'\rangle \in \mathcal{E}$, $\mathrm{E}' \subseteq \mathrm{E}$, $\mathrm{E}' \vdash P'$, *and* $\langle \eta', \mathrm{E}'\rangle$ *is closed.*

The denotational semantics defines all possible concrete values that an abstract environment could take on.

Lemma 4.8 (Properties of $[\![\mathcal{E}]\!]$). 1. *If* $x \mapsto \langle \eta, \mathrm{E}\rangle \notin \mathrm{E}' \subseteq [\![\mathcal{E}]\!]$ *then* $x \mapsto \langle \eta, \mathrm{E}\rangle \in [\![\mathcal{E}]\!]$.
2. *If* $\mathrm{E} \subseteq [\![\mathcal{E}]\!]$, $\mathrm{E}(x)^{+\lambda} = \langle \mathcal{F}, \mathrm{E}'\rangle$ *and* $\mathrm{E} \vdash P$, *then* $\mathrm{E}' \subseteq [\![\mathcal{E}]\!]$, *and there exists a* P', *such that* $\langle \mathcal{F}, P'\rangle \in \mathcal{E}(x)^{+\lambda}$ *and* $\mathrm{E}' \vdash P'$.

Proof. Follows by induction given Definition 4.7 □

Definition 4.9 (Well-Formed 3-Tuple $(\mathcal{E}, P, p)$). *A 3-tuple* $(\mathcal{E}, P, p)$ *is said to be* well-formed *iff* $\mathcal{E}$ *is the nugget of* $(\mathcal{E}, P, p)$.

Finally, we can prove the soundness of the nuggetizer using a simulation argument. The simulation relation is defined as follows.

Definition 4.10 (Simulation Relation). $\mathrm{E} \sqsubseteq_p (\mathcal{E}, P)$ *iff* (E, p) *and* $(\mathcal{E}, P, p)$ *are each well-formed,* $\mathrm{E} \subseteq [\![\mathcal{E}]\!]$*, and* $\mathrm{E} \vdash P$*.*

Lemma 4.11 (Simulation of COS by AOS). *If* $\mathrm{E} \sqsubseteq_p (\mathcal{E}, P)$ *and* $(\mathrm{E}, p) \longrightarrow^n (\mathrm{E}', p')$ *then* $(\emptyset, \mathcal{E}, P, p) \longrightarrow^n (\emptyset, \mathcal{E}', P', p')$ *and* $\mathrm{E}' \sqsubseteq_{p'} (\mathcal{E}, P')$*.*

The following Lemma essentially states that the nugget, as extracted by the nuggetizer, is a sound abstraction of corresponding program computation.

Main Lemma 4.12 (Soundness of the Nuggetizer). *For a canonical program* p*, if* $nuggetizer(p) = \mathcal{E}$*,* (E', p') *is a node in the derivation tree of* $(\emptyset, p) \longrightarrow^n (\mathrm{E}_n, p_n)$*, and* $x \mapsto \langle \eta, \mathrm{E} \rangle \in \mathrm{E}'$*, then* $x \mapsto \langle \eta, \mathrm{E} \rangle \in [\![\mathcal{E}]\!]$*.*

Proof. Follows by Lemmas 3.4[3], 4.8[1] and 4.11. □

We now show that the nuggetizer is a total computable function. We start by defining the function $pairs\,(P, p)$ which defines a set of subpairs of the pair (P, p).

Definition 4.13 (Pairs of Predicates and Sub-Programs).

1. $pairs\,(P, x) = \{(P, x)\}$*; and,* $pairs\,(P, i) = \{(P, i)\}$*; and,* $pairs\,(P, b) = \{(P, b)\}$*; and,* $pairs\,(P, \lambda x.\, p) = \{(P, \lambda x.\, p)\} \cup pairs\,(P, p) \cup pairs\,(P, \lfloor p \rfloor)$*;* $pairs\,(P, i) = \{(P, i)\}$*; and,* $pairs\,(P, x \oplus x') = \{(P, x \oplus x')\}$*; and,*
2. $pairs\,(P, \mathsf{let}\ x = \eta\ \mathsf{in}\ p) = \{(P, \mathsf{let}\ x = \eta\ \mathsf{in}\ p)\} \cup pairs\,(P, \eta) \cup pairs\,\big(P \wedge (x = \eta), p\big)$*; and*
3. $pairs\,\big(P, \mathsf{let}\ y = (\mathsf{if}\ x\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2)\ \mathsf{in}\ p\big) = \{\big(P, \mathsf{let}\ y = (\mathsf{if}\ x\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2)\ \mathsf{in}\ p\big)\} \cup pairs\,(P_1, p_1) \cup pairs\,(P_2, p_2) \cup pairs\,(P', p)$*, where* $P_1 = P \wedge (x = \mathsf{true})$*,* $P_2 = P \wedge (x = \mathsf{false})$*,* $\lfloor (P_1, p_1) \rfloor = (P_1', y_1')$*,* $\lfloor (P_2, p_2) \rfloor = (P_2', y_2')$*, and* $P' = \big(P_1' \wedge (y = y_1')\big) \vee \big(P_2' \wedge (y = y_2')\big)$*; and*
4. $pairs\,(P, \mathsf{let}\ r = f\ x\ \mathsf{in}\ p) = \{(P, \mathsf{let}\ r = f\ x\ \mathsf{in}\ p)\} \cup pairs\,(P, x) \cup pairs\,(P, p)$*.*

The size of a program is defined as,

Definition 4.14 (Size of a Program: $|\,p\,|$).

1. $|\,x\,| = |\,i\,| = |\,b\,| = |\,x \oplus x'\,| = 1$*;*
2. $|\,\lambda x.\, p\,| = 1 + |\,p\,| + 1$*;*
3. $|\,\mathsf{let}\ x = \eta\ \mathsf{in}\ p\,| = 1 + |\,\eta\,| + |\,p\,|$*;*
4. $|\,\mathsf{let}\ y = (\mathsf{if}\ x\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2)\ \mathsf{in}\ p\,| = 1 + |\,p_1\,| + |\,p_2\,| + |\,p\,|$*;*
5. $|\,\mathsf{let}\ r = f\ x\ \mathsf{in}\ p\,| = 1 + 1 + |\,p\,|$*.*

The function $absClosures(P, p)$ denotes a set of abstract closures in the pair (P, p): $absClosures(P, p) = \big\{\langle \eta', P' \rangle \mid (P', \eta') \in pairs\,(P, p)\big\}$.

Lemma 4.15 (Properties of $|\,p\,|$). *1.* $|\,locals(p)\,| \leq |\,p\,|$*.*
2. $|\,pairs\,(P, p)\,| \leq |\,p\,|$*, for any* P*.*
3. $|\,absClosures(P, p)\,| \leq |\,p\,|$*, for any* P*.*

Proof. Follows by Definition 4.14. □

The following key Lemma states that there exists an upper bound on the abstract environment for all programs.

Lemma 4.16 (Upper Bound on the Abstract Environment). *For a program* p*, if* $\mathcal{E} \subseteq locals(p) \times absClosures(\mathsf{true}, p)$ *and* $(\emptyset, \mathcal{E}, \mathsf{true}, p) \longrightarrow^n (\emptyset, \mathcal{E}', P, r)$ *then* $\mathcal{E}' \subseteq locals(p) \times absClosures(\mathsf{true}, p)$*.*

We write $pairs\,(\mathcal{E}, P, p)$ as a shorthand to denote the set $pairs\,(P, p) \cup \bigcup_{1 \leq i \leq k} pairs\,(P_i, \eta_i)$, where $\mathcal{E} = \{\overline{x_k \mapsto \langle \eta_k, P_k \rangle}\}$. The following lemma states that there is an upper bound on the number of distinct (predicate, redex) pairs in the AOS of any 3-tuple $(\mathcal{E}, P, p)$.

Lemma 4.17 (Upper Bound on (Predicate, Redex) Pairs). *If* $(\mathcal{S}', \mathcal{E}', P', p')$ *is a node in the derivation tree of* $(\mathcal{S}, \mathcal{E}, P, p) \longrightarrow^n (\mathcal{S}'', \mathcal{E}'', P'', p'')$ *then* $(P', p') \in pairs\,(\mathcal{E}, P, p)$*.*

The combination of the termination of the AOS, monotonic growth of the environment, and the existence of an upper bound on the abstract environment means that successive re-runs of the AOS will terminate in a least fixed-point.

Main Lemma 4.18 (Computability of the Nuggetizer). *The function* $nuggetizer(p) = \mathcal{E}$ *is computable.*

Proof. Follows from Lemmas 4.3, 4.2[3b] 4.16 and 4.2[1]. □

We present the complexity of the nuggetizer.

Lemma 4.19 (Complexity of the Nuggetizer). *The runtime complexity of the nuggetizer is* $O(n! \cdot n^3)$*, where* $n = |\,p\,|$*.*

The $n!$ component of the above complexity is due to the possibility of $O(n!)$ distinct configurations. Here is an example realizing this worst case:

$$\begin{array}{l}\mathsf{let}\ x = \mathsf{if}\ y\ \mathsf{then}\ \lambda x_1.\, x_1\ x_1 \\ \qquad\qquad \mathsf{else\ if}\ y'\ \mathsf{then}\ \lambda x_2.\, x_2\ x_2 \\ \qquad\qquad \mathsf{else\ if}\ y''\ \mathsf{then}\ \lambda x_3.\, x_3\ x_3 \\ \qquad\qquad\qquad \mathsf{else}\ \lambda x_4.\, x_4\ x_4 \\ \mathsf{in}\ x\ x\end{array}$$

The environment of the AOS at '$x\ x$' contains mappings $\{x \mapsto \lambda x_1.\, x_1\ x_1, x \mapsto \lambda x_2.\, x_2\ x_2, x \mapsto \lambda x_3.\, x_3\ x_3, x \mapsto \lambda x_4.\, x_4\ x_4\}$ for x (ignoring the predicates for simplicity). As a result the AOS is forced to successively invoke them in all possible orders *i.e.* 16, or $O(4!)$, resulting in $O(4!)$ different configurations. Generalizing the above example to n such functions would result in $O(n!)$ different configurations.

All of the exponential programs we have been able to construct are highly unnatural; it is an open question whether common programming patterns would realize the combinatorial bound.

Note that the non-elementary bound on simply typed λ-reduction (Mairson 1992) does not apply to the nuggetizer 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:

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

(Mairson 1992) encodes primitive recursion in simply typed λ-calculus and shows the complexity of its reduction as non-elementary, while the AOS prunes all recursive invocations including ones such as the above.

4.3 Towards Automated Theorem Proving

In this section we provide "glue" which connects the notation of the formal framework above with the syntax of the Isabelle/HOL theorem prover. The basic idea is we define an equivalent denotational semantics of environments, $[\![\mathcal{E}]\!]$, which grounds programs by inlining the concrete environment.

We need to relax the grammar for lazy values, atomic computations and programs to be used in this subsection as follows: $\eta ::= i \mid b \mid \mathcal{F} \mid \eta \oplus \eta$, $\kappa ::= \eta \mid \mathsf{if}\ \eta\ \mathsf{then}\ p\ \mathsf{else}\ p \mid \eta\ \eta$, and $p ::= \eta \mid \mathsf{let}\ x = \kappa\ \mathsf{in}\ p$, respectively. We write $p[\eta/x]$ to denote

the capture-avoiding substitution of all free occurrences of x in p with η.

The following function reduces a lazy value to its smallest equivalent form, or as we say "grounds" it.

Definition 4.20 (Ground a Lazy Value). $\lfloor\!\lfloor \eta \rfloor\!\rfloor$ *is a function inductively defined as,*

1. $\lfloor\!\lfloor x \rfloor\!\rfloor = x$; $\lfloor\!\lfloor i \rfloor\!\rfloor = i$; $\lfloor\!\lfloor b \rfloor\!\rfloor = b$; $\lfloor\!\lfloor \mathcal{F} \rfloor\!\rfloor = \mathcal{F}$; *and*
2. $\lfloor\!\lfloor \eta_1 \oplus_i \eta_2 \rfloor\!\rfloor = \eta$, *if* $\lfloor\!\lfloor \eta_1 \rfloor\!\rfloor = i_1$, $\lfloor\!\lfloor \eta_2 \rfloor\!\rfloor = i_2$, *and* $i_1 \oplus_i i_2 = \eta$, *where* η *is either integral or boolean; else,* $\lfloor\!\lfloor \eta_1 \oplus_i \eta_2 \rfloor\!\rfloor = \lfloor\!\lfloor \eta_1 \rfloor\!\rfloor \oplus_i \lfloor\!\lfloor \eta_2 \rfloor\!\rfloor$.
3. $\lfloor\!\lfloor \eta_1 \oplus_b \eta_2 \rfloor\!\rfloor = b$, *if* $\lfloor\!\lfloor \eta_1 \rfloor\!\rfloor = b_1$, $\lfloor\!\lfloor \eta_2 \rfloor\!\rfloor = b_2$, *and* $b_1 \oplus_b b_2 = b$; *else* $\lfloor\!\lfloor \eta_1 \oplus_b \eta_2 \rfloor\!\rfloor = \lfloor\!\lfloor \eta_1 \rfloor\!\rfloor \oplus_b \lfloor\!\lfloor \eta_2 \rfloor\!\rfloor$.

We define a new concrete environment denoting the environment in the theorem prover as, $\mathbb{E} ::= \{\overline{x \mapsto \eta}\}$. Further we write $p[\mathbb{E}]$ as shorthand for $p[\overline{\eta_k / x_k}]$, where $\mathbb{E} = \{\overline{x_k \mapsto \eta_k}\}$.

Definition 4.21 (Predicate Satisfaction Relation: $\mathbb{E} \vdash P$).

1. $\mathbb{E} \vdash \mathsf{true}$;
2. $\mathbb{E} \vdash \eta_1 = \eta_2$, *iff* $\lfloor\!\lfloor \eta_1[\mathbb{E}] \rfloor\!\rfloor = \lfloor\!\lfloor \eta_2[\mathbb{E}] \rfloor\!\rfloor$;
3. $\mathbb{E} \vdash P \wedge P'$, *iff* $\mathbb{E} \vdash P$ *and* $\mathbb{E} \vdash P'$;
4. $\mathbb{E} \vdash P \vee P'$, *iff either,* $\mathbb{E} \vdash P$ *or* $\mathbb{E} \vdash P'$.

Definition 4.22 (Denotational Semantics, for Theorem Prover, of $\mathcal{E}$: $[\![\mathcal{E}]\!]$). $[\![\mathcal{E}]\!]$ *is smallest set* $\mathbb{E}$ *such that,*

1. $x \mapsto \eta' \in \mathbb{E}$, *if* $x \mapsto \langle \eta, P \rangle \in \mathcal{E}$, $\emptyset \vdash P$, $\lfloor\!\lfloor \eta \rfloor\!\rfloor = \eta'$, *and* η' *is closed; and,*
2. $x \mapsto \eta' \in \mathbb{E}$, *if* $x \mapsto \langle \eta, P \rangle \in \mathcal{E}$, $\mathbb{E}' \subseteq \mathbb{E}$, $\mathbb{E}' \vdash P$, $\lfloor\!\lfloor \eta[\mathbb{E}'] \rfloor\!\rfloor = \eta'$, *and* η' *is closed.*

Given the relaxed grammar, we redefine the ground of a concrete closure as,

Definition 4.23 (Ground of a Concrete Closure). $\lfloor\!\lfloor \langle \eta, \mathrm{E} \rangle \rfloor\!\rfloor$ *is a function inductively defined as,*

1. $\lfloor\!\lfloor \langle x, \mathrm{E} \rangle \rfloor\!\rfloor = \lfloor\!\lfloor \mathrm{E}(x) \rfloor\!\rfloor$; $\lfloor\!\lfloor \langle i, \mathrm{E} \rangle \rfloor\!\rfloor = i$; $\lfloor\!\lfloor \langle b, \mathrm{E} \rangle \rfloor\!\rfloor = b$; $\lfloor\!\lfloor \langle \mathcal{F}, \mathrm{E} \rangle \rfloor\!\rfloor = \mathcal{F}[\mathbb{E}]$, *where* $free(\mathcal{F}) \cap dom(\mathrm{E}) = \{\overline{x_k}\}$ *and* $\mathbb{E} = \{\overline{x_k \mapsto \lfloor\!\lfloor \langle x_k, \mathrm{E} \rangle \rfloor\!\rfloor}\}$; *and,*
2. $\lfloor\!\lfloor \langle \eta_1 \oplus_i \eta_2, \mathrm{E} \rangle \rfloor\!\rfloor = \eta$, *if* $\lfloor\!\lfloor \langle \eta_1, \mathrm{E} \rangle \rfloor\!\rfloor = i_1$, $\lfloor\!\lfloor \langle \eta_2, \mathrm{E} \rangle \rfloor\!\rfloor = i_2$, *and* $i_1 \oplus_i i_2 = \eta$, *where* η *is either integral or boolean; else,* $\lfloor\!\lfloor \langle \eta_1 \oplus_i \eta_2, \mathrm{E} \rangle \rfloor\!\rfloor = \lfloor\!\lfloor \langle \eta_1, \mathrm{E} \rangle \rfloor\!\rfloor \oplus_i \lfloor\!\lfloor \langle \eta_2, \mathrm{E} \rangle \rfloor\!\rfloor$; *and,*
3. $\lfloor\!\lfloor \langle \eta_1 \oplus_b \eta_2, \mathrm{E} \rangle \rfloor\!\rfloor = b$, *if* $\lfloor\!\lfloor \langle \eta_1, \mathrm{E} \rangle \rfloor\!\rfloor = b_1$, $\lfloor\!\lfloor \langle \eta_2, \mathrm{E} \rangle \rfloor\!\rfloor = b_2$, *and* $b_1 \oplus_b b_2 = b$; *else,* $\lfloor\!\lfloor \langle \eta_1 \oplus_b \eta_2, \mathrm{E} \rangle \rfloor\!\rfloor = \lfloor\!\lfloor \langle \eta_1, \mathrm{E} \rangle \rfloor\!\rfloor \oplus_b \lfloor\!\lfloor \langle \eta_2, \mathrm{E} \rangle \rfloor\!\rfloor$.

We now show the original denotational semantics and the revised form designed for the theorem prover are equivalent.

Lemma 4.24 (Equivalence of $[\![\mathcal{E}]\!]$ and $[\![\mathcal{E}]\!]$).

1. *If* $x \mapsto \langle \eta, \mathrm{E} \rangle \in [\![\mathcal{E}]\!]$ *then* $\lfloor\!\lfloor \langle \eta, \mathrm{E} \rangle \rfloor\!\rfloor = \eta'$ *and* $x \mapsto \eta' \in [\![\mathcal{E}]\!]$.
2. *If* $x \mapsto \eta \in [\![\mathcal{E}]\!]$ *then there exists a* $\langle \eta', \mathrm{E}' \rangle$, *such that* $\lfloor\!\lfloor \langle \eta', \mathrm{E}' \rangle \rfloor\!\rfloor = \eta$ *and* $x \mapsto \langle \eta', \mathrm{E}' \rangle \in [\![\mathcal{E}]\!]$.

Proof. Follows by Definitions 4.7 and 4.22. □

The following theorem then shows that all values arising in variables in program runs will be found in the theorem-prover form of the denotational semantics of the environment, meaning this denotation soundly reflects run-time program behavior.

Theorem 4.25 (Soundness of the Analysis). *For a canonical program* p, *if* $nuggetizer(p) = \mathcal{E}$, (E', p') *is a node in the derivation tree of* $(\emptyset, p) \longrightarrow^n (\mathrm{E}_n, p_n)$, *and* $x \mapsto \langle \eta, \mathrm{E} \rangle \in \mathrm{E}'$ *then* $\lfloor\!\lfloor \langle \eta, \mathrm{E} \rangle \rfloor\!\rfloor = \eta'$ *and* $x \mapsto \eta' \in [\![\mathcal{E}]\!]$.

Proof. Follows by Lemmas 4.12 and 4.24[1]. □

5. Automated Theorem Proving

The analysis developed thus far is able to produce a nugget, and we have formally defined the denotational semantics of nuggets. In this section we show how to use the Isabelle/HOL proof assistant (Nipkow et al. 2002) to formalize and prove nugget properties. Isabelle/HOL has a rich vocabulary that is well-suited to the encoding of nuggets, and has a number of built-in proof strategies that allow automated proofs to be constructed for properties of interest. Intuitively, our strategy is to translate any given nugget into an inductively defined set in the prover. For any such definition, Isabelle/HOL automatically generates an inductive proof strategy which can be leveraged to prove value range properties.

More formally, we encode variables x via the following Isabelle/HOL datatype, using numbers to distinguish variables in the encoding:

```
datatype var = X of nat
```

Hereafter we will assume for simplicity that all variables in the source language are of the form x_i where the index i is in $\mathbb{N}$, so that encoding of each is straightforward as $X(i)$. Given any set of variables V, their indices $\{i_1, \ldots, i_n\}$ are denoted $indices(V)$. The encoding of any given nugget $\mathcal{E}$, denoted $[\![\mathcal{E}]\!]_{\mathrm{HOL}}$, is defined inductively as a set of (var, nat) pairs called "abstractenv":

```
consts abstractenv :: "(var * nat) set"
inductive abstractenv
```

Each mapping $x_i \mapsto \langle \eta, P \rangle$ in $\mathcal{E}$ then defines a separate clause in the inductive definition, where P defines a set of preconditions. Furthermore any variable x_i referenced in η and P needs to be changed to an Isabelle/HOL variable v_i, and associated with x_i via the precondition $(X(i), v_i)$: abstractenv. In full detail, individual clauses are encoded as introduction rules via the function $[\![\cdot]\!]_{\mathrm{HOL}}$ as follows:

$$
\begin{array}{l}
[\![x_i \mapsto \langle \eta, P \rangle]\!]_{\mathrm{HOL}} = \\
\quad \text{if } free(\eta) = \varnothing \text{ then} \\
\quad\quad \text{“}[|[\![P]\!]_{\mathrm{HOL}}|] \Longrightarrow (X(i), [\![\eta]\!]_{\mathrm{HOL}}) : \text{abstractenv”} \\
\quad \text{else let } \{i_1, \ldots, i_n\} = indices(free(\eta)) \text{ in} \\
\quad\quad \text{“}[|(X(i_1), v_{i_1}) : \text{abstractenv}; \ldots; \\
\quad\quad\quad (X(i_n), v_{i_n}) : \text{abstractenv}; \\
\quad\quad\quad [\![P]\!]_{\mathrm{HOL}}|] \Longrightarrow (X(i), [\![\eta]\!]_{\mathrm{HOL}}) : \text{abstractenv”}
\end{array}
$$

The above definition references our Isabelle/HOL encoding of predicates and abstract values $[\![P]\!]_{\mathrm{HOL}}$ and $[\![\eta]\!]_{\mathrm{HOL}}$ respectively, which are defined as follows:

$$
\begin{array}{rcl}
[\![b]\!]_{\mathrm{HOL}} & = & \text{“}b\text{”} \\
[\![i]\!]_{\mathrm{HOL}} & = & \text{“}i\text{”} \\
[\![x_i]\!]_{\mathrm{HOL}} & = & \text{“}v_i\text{”} \\
[\![\eta \oplus \eta]\!]_{\mathrm{HOL}} & = & \text{“}[\![\eta_1]\!]_{\mathrm{HOL}} \oplus [\![\eta_2]\!]_{\mathrm{HOL}}\text{”} \\
[\![P_1 \wedge P_1]\!]_{\mathrm{HOL}} & = & \text{“}[\![P_1]\!]_{\mathrm{HOL}} \ \&\ [\![P_2]\!]_{\mathrm{HOL}}\text{”} \\
[\![P_1 \vee P_1]\!]_{\mathrm{HOL}} & = & \text{“}[\![P_1]\!]_{\mathrm{HOL}} \ \mid\ [\![P_2]\!]_{\mathrm{HOL}}\text{”}
\end{array}
$$

The main point to notice in this last definition is how variables x_i referenced directly in the nugget mapping are referenced indirectly via the variable v_i associated with $X(i)$ in the encoding. For example, consider the nugget:

$$\{x_0 \mapsto 5, x_0 \mapsto (x_0 - 1)^{x_0 != 0}\}$$

that would be generated by analysis of the factorial function discussed in the introduction if x_0 were substituted for n. The encoding of this nugget will generate the following Isabelle/HOL defini-

tion:

consts abstractenv :: "(var * nat) set"
inductive abstractenv
intros
"[| true |] $\Longrightarrow (X(0), 5)$: abstractenv"
"[|$(X(0), v_0)$: abstractenv; $v_0 \neq 0$|]
$\Longrightarrow (X(0), v_0 - 1)$: abstractenv"

To prove properties of the nugget, we can then leverage the Isabelle/HOL proof assistant. In particular, to prove that x_0 falls in the range [0,5], we state the following theorem:

theorem range :
"$(X(0), v_0)$: abstractenv $\Longrightarrow v_0 \leq 5$ & $v_0 \geq 0$".

Following this, a single application of the elimination rule abstractenv.induct will unroll the theorem according to the inductive definition of abstractenv, and the resulting subgoals can be solved by two applications of the arith strategy. While we have proved this and other more complicated examples in an interactive manner, the strategy in each case is the same: apply the inductive elimination rule, followed by one or more applications of the arith strategy. This suggests a fully automated technique for proof. We note that the nugget encoding itself is fully automated by the preceding definitions.

Observe that the Isabelle/HOL least fixpoint interpretation of the encoding $[\![\mathcal{E}]\!]_{\mathrm{HOL}}$ can be seen to be equivalent to $[\![\mathcal{E}]\!]$ as specified in Definition 4.22, by inspection of the two definitions with the same structure, for any fixed $\mathcal{E}$. Thus, by Theorem 4.25, any property of $[\![\mathcal{E}]\!]_{\mathrm{HOL}}$ verified in Isabelle/HOL for all values of variables is a property of the runtime variables of the program p for which $\mathcal{E}$ is its nugget.

6. The Extended Nuggetizer

The nuggetizer of the previous section does not capture the correlation between argument and return values of function invocations, and hence is unable to precisely track return values of recursive functions, as discussed in Section 2.2. This nuggetizer however can be easily extended to capture the above mentioned dependency; in this section we present such an extended nuggetizer. We do not present proofs of properties of this extension in this paper, but the properties of previous section may all easily be proved, with the exception of runtime complexity which will be worse. We have not yet developed an encoding of extended nuggets into the theorem prover.

The *let* and *if* rules are identical to those in the core AOS. Figure 6 presents the extended AOS *app*$^+$ rule; it is identical to the original *app* except it has added the underlined conjunctions. The predicate $r'_i.x_i = x'$ additionally captures the dependency between the argument and return value of each function invocation. The added predicate $(r = r'_i)$ in the *app*$^+$ rule is analogous to the $(x = \eta)$ predicate in the *let* rule and serves the same purpose. The grammar also must be extended to include this new atomic predicate:

$$P ::= \ldots \mid x.x = \eta$$

Consequently, Definition 4.6, the predicate satisfaction relation, is also extended to include the following case: $\mathrm{E} \vdash r.x = \eta$ iff $[\![\mathrm{E}'(x)]\!] = [\![\langle \eta, \mathrm{E} \rangle]\!]$, where $\mathrm{E}(r) = \langle \eta', \mathrm{E}' \rangle$, for some η'. Note how the interpretation of '$r.x$' in the above definition looks into r's closure environment E' and not the top-level environment E for x; this matches the intuition that $r.x$ means the mapping for x in the environment under which r was generated. The rest of the definitions of the previous section can be re-used in the extended AOS and nuggetizer without change, stopping upon reaching Subsection 4.3 (the theorem prover mapping).

Now consider program (7) from Section 2.2 again. Its nugget under this extended AOS is, after some simplification,

$$\Big\{n \mapsto 5, n \mapsto (n-1)^{n\,!=0}, \quad r \mapsto 1^{n==0}, r \mapsto (n * r)^{n\,!=0 \,\wedge\, \left(r.n = (n-1)\right)}\Big\} \tag{13}$$

The predicate $\big(r.n = (n-1)\big)$ defines a dependency between the values of n and r that can be used in the above mapping to generate new values for r. This dependency allows only the return value mapped to r which corresponds to $(n-1)$ to be multiplied with n in order to create a new return value which then corresponds to n. Now suppose a partial denotation of the above nugget for the return variable r, as per Definition 4.7 when used with the extended Definition 4.6 is:

$$\Big\{r \mapsto \big\langle 1, \{n \mapsto 0\}\big\rangle, \quad r \mapsto \big\langle 1, \{n \mapsto 1, \ldots\}\big\rangle, \quad r \mapsto \big\langle 2, \{n \mapsto 2, \ldots\}\big\rangle, \quad r \mapsto \big\langle 6, \{n \mapsto 3, \ldots\}\big\rangle\Big\}$$

We will now show how new bindings for r will result, assuming known range for n is $[0, 5]$. Observe the correlation between the binding for r and the binding for n under which it was generated – the latter is a factorial of the former, that is, the latter is the return value for function $(\lambda n. \ldots)$ in program (7) when invoked on the former. For $n \mapsto 4$ the predicate $r.n = (n-1)$ allows only the mapping for r which was generated under an environment containing $n \mapsto (4-1)$, that is, $n \mapsto 3$. Note, $r \mapsto \big\langle 6, \{n \mapsto 3, \ldots\}\big\rangle$ is the only mapping satisfying the above predicate which leads to the generation of mapping $r \mapsto \big\langle 24, \{n \mapsto 4, \ldots\}\big\rangle$ for r. The complete denotation of the above nugget is,

$$\Big\{r \mapsto \big\langle 1, \{n \mapsto 0\}\big\rangle, \quad r \mapsto \big\langle 1, \{n \mapsto 1, \ldots\}\big\rangle, \quad r \mapsto \big\langle 2, \{n \mapsto 2, \ldots\}\big\rangle, \quad r \mapsto \big\langle 6, \{n \mapsto 3, \ldots\}\big\rangle, \quad r \mapsto \big\langle 24, \{n \mapsto 4, \ldots\}\big\rangle, \quad r \mapsto \big\langle 120, \{n \mapsto 5, \ldots\}\big\rangle\Big\}$$

which is the precise range of the return variable r.

The Main Lemma 4.12 stating the soundness of the nuggetizer holds directly in presence of this extended AOS. We plan to extend the theorem prover encoding to process extended nuggets; it will require a more complex encoding which will keep track of the environments under which the values were generated as well.

7. Related Work

This work is an abstract interpretation (Cousot and Cousot 1977a,b) in the sense that an abstraction of an operational semantics is defined. It differs from the abstract interpretation literature in methodology and focus on higher-order functions – nearly all of the abstract interpretation literature focuses on first-order programs. We are also not interested in abstracting away any of the (infinite) structure of the underlying data domains. Some earlier work on higher-order abstract interpretation (Cousot and Cousot 1994; Ashley 1996; Nielson and Nielson 1997; Jones and Rosendahl 1997) does exist. The most related abstract interpretation is the recent LFA (Might 2007), a paper that addresses a similar problem but by different technical means. LFA is over a richer language, but is more a proposal in that it has no formal proofs. We are concerned that it may be infeasible to implement in practice since it fundamentally relies on an initial CPS transformation step, and because of this will never merge results of if-then statements. Since nontriv-

$$\frac{\begin{array}{c}\mathcal{E}(f)^{+\lambda} = \{\overline{\langle \mathcal{F}_k, P_k\rangle}\} \qquad \overline{\mathcal{F}_k = \lambda x_k.\, p_k} \\ \forall 1 \le i \le k \qquad \textit{CALL}\big(\mathcal{S}, \langle \mathcal{F}_i, P_i\rangle\big) = p_i' \qquad \mathcal{S}_i = \mathcal{S} \cup \{\langle \mathcal{F}_i, P_i\rangle\} \qquad \big(\mathcal{S}_i, \mathcal{E} \cup \{x_i \mapsto \langle x', P\rangle\}, P_i, p_i'\big) \longrightarrow^{n_i} \big(\mathcal{S}_i, \mathcal{E}_i, P_i', r_i'\big)\end{array}}{\big(\mathcal{S}, \mathcal{E}, P, \mathsf{let}\ r = f\ x'\ \mathsf{in}\ p_{next}\big) \longrightarrow \left(\mathcal{S}, \mathcal{E} \cup \bigcup_{1\le i\le k} \mathcal{E}_i \cup \{r \mapsto \langle r_i', P_i'\rangle\},\ \bigvee_{1\le i\le k} P \wedge \underline{(r_i'.x_i = x') \wedge (r = r_i')}, p_{next}\right)} \textit{app}^+$$

Figure 6. Extended Abstract Operational Semantics (AOS) app^+ rule with addition underlined

ial programs have significant numbers of conditionals, there will be many paths in the conditionals tree to explore and no reasonable way to prune them since CPS has unfortunately removed the join points. LFA has potentially more precision on function calls by using multiple contours *a la* 1CFA or CPA; we plan to incorporate this extension. LFA also proposes a connection with a theorem-prover as we do but has not formally defined a mapping to a prover, or confirmed that the theorem prover can in fact verify actual examples as we have.

There are also numerous type systems which are methodologically very different than our approach, but do overlap in how they aim to solve some similar problems. In particular, dependent and refinement type systems can assert complex program invariants (Chen and Xi 2005; Jones et al. 2006; Xi and Pfenning 1999, 1998; Freeman and Pfenning 1991; Gronski et al. 2006). Our approach has a good combination of expressiveness and automation in comparison to the type-based approaches in that it gives precise, automatic answers to these problems. In general both type- and abstract-interpretation-based approaches need to be pursued because each has unique strengths.

8. Conclusion

We have defined a value range analysis of functional programs. We believe this work has several novel aspects. The abstract environments include *guards* indicating dependencies. (We have recently shown this dependency information is also useful in a very different setting (Shroff et al. 2007).) The analysis is "fully supportive" of higher-order programs, including the case of recursive functions returning function values. Our main algorithm is the *nuggetizer* which collects a *nugget* from a program: a sound approximation of value bindings resulting from program execution. The nuggets reflect the higher-order flow of the original program, but are first-order objects themselves. The nuggetizer process itself has some unique aspects, in particular the *prune-rerun* technique for soundly interpreting higher-order functions. We show how the meaning of nuggets may easily be formalized in the HOL theorem-prover and program properties thereby proved. The nuggetizer is proved sound, with the proofs appearing in the appendices.

While for this paper we focus on a narrow problem for a pure functional language, our general goal is more broad. We have done initial work on extensions to incorporate flow-sensitive mutable state and context-sensitivity. The nuggetizer is aimed at completely automatic proofs of program properties, but its running time may be long. So, the analysis lies between the level of analysis performed by an optimizing compiler and the time required for full, manual theorem proving.

References

J. Michael 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.

Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 66–77, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-064-7.

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*, 1977a.

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

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.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI'93: Proceedings ACM SIGPLAN conference on Programming Language Design and Implementation*, 1993.

Tim Freeman and Frank Pfenning. Refinement types for ML. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 268–277, New York, NY, USA, 1991. ACM Press. ISBN 0-89791-428-7.

Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. Sage: Hybrid checking for flexible specifications. In *Workshop on Scheme and Functional Programming*, 2006.

Neil D. Jones and Mads Rosendahl. Higher-order minimal function graphs. In *Journal of Functional and Logic Programming*, 1997.

Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *International Conference on Functional Programming (ICFP)*, 2006.

Harry G. Mairson. A simple proof of a theorem of Statman. *Theoretical Computer Science*, 1992.

Matthew Might. Logic-flow analysis of higher-order programs. In *Proceedings of the 34th Annual ACM Symposium on the Principles of Programming Languages (POPL 2007)*, pages 185–198, Nice, France, January 2007.

Hanne Riis Nielson and Flemming 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.

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, vol-

ume 2283 of *LNCS*. Springer, 2002.

Paritosh Shroff, Scott F. Smith, and Mark Thober. Dynamic dependency monitoring to secure information flow. In *CSF 2007: 20th IEEE Computer Security Foundations Symposium*, 2007.

Hongwei Xi and Frank 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. ISBN 0-89791-987-4.

Howgwei Xi and Frank 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. URL `citeseer.ist.psu.edu/xi98dependent.html`.

A. Definitions and Proofs for the COS

The complement operation on a generic set of mappings, $\mathbb{M} := \{\overline{d \mapsto r}\}$, is defined as, $\mathbb{M}\backslash d = \{d' \mapsto r' \mid d' \mapsto r' \in \mathbb{M} \wedge d \neq d'\}$.

Definition A.1 (Free Variables). *1. (Variable).* $free(x) = \{x\}$.
2. (Lazy Value).
(a) $free(i) = free(b) = \emptyset$.
(b) $free(\lambda x.\, p) = free(p) - \{x\}$.
(c) $free(x \oplus x') = \{x, x'\}$.
3. (Atomic Computation).
(a) $free(\mathsf{if}\ x\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2) = \{x\} \cup free(p_1) \cup free(p_2)$.
(b) $free(x\ x') = \{x, x'\}$.
4. (A-normal Program). $free(\mathsf{let}\ x = \kappa\ \mathsf{in}\ p) = free(\kappa) \cup (free(p) - \{x\})$.

Definition A.2 (Local Variables). *1. (Variable).* $locals(x) = \emptyset$.
2. (Lazy Value).
(a) $locals(i) = locals(b) = locals(x \oplus x') = \emptyset$.
(b) $locals(\lambda x.\, p) = \{x\} \cup locals(p)$.
3. (Atomic Computation).
(a) $locals(\mathsf{if}\ x\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2) = locals(p_1) \cup locals(p_2)$.
(b) $locals(x\ x') = \emptyset$.
4. (A-normal Program). $locals(\mathsf{let}\ x = \kappa\ \mathsf{in}\ p) = \{x\} \cup locals(\kappa) \cup locals(p)$.

Lemma A.3 (Properties of $\|\langle \eta, \mathrm{E}\rangle\|$). *1. If* $\mathrm{E}(x)$ *is defined then* $\|\langle x, \mathrm{E}\rangle\| = \|\mathrm{E}(x)\|$.
2. If $x \notin free(\eta)$ *then* $\|\langle \eta, \mathrm{E}\rangle\| = \|\langle \eta, \mathrm{E}\backslash x\rangle\|$.

Proof. Follows by Definition 3.1. □

Lemma A.4 (Property of a Canonical Closure). *If* $\langle \eta, \mathrm{E}\rangle$ *is canonical and* $\mathrm{E}(\eta)^{+_\lambda} = \langle \mathcal{F}, \mathrm{E}'\rangle$ *then* $\langle \mathcal{F}, \mathrm{E}'\rangle$ *is canonical.*

Proof. Directly by Definition 3.2[2]. □

B. Proofs for the AOS and Nuggetizer

Lemma B.1 (Properties of Predicate Satisfaction Relation). *1. If* $\mathrm{E} \vdash P$, $\mathrm{E} \subseteq \mathrm{E}'$ *and* E' *is a set of single-valued mappings, then* $\mathrm{E}' \vdash P$.
2. If $\mathrm{E} \vdash P$, $x \notin dom(\mathrm{E})$ *and* $x \notin free(\eta)$ *then* $\mathrm{E} \cup \{x \mapsto \langle \eta, \mathrm{E}\rangle\} \vdash P \wedge (x = \eta)$.
3. If $\mathrm{E} \vdash P$ *and* $\|\langle \eta, \mathrm{E}\rangle\| = \langle b, \emptyset\rangle$ *then* $\mathrm{E} \vdash P \wedge (\eta = b)$.

Proof. Follows by Definition 4.6. □

Lemma B.2 (Properties of AOS under Fixed-Point of Environment).

1. (Stack Expansion). If $(\mathcal{S}, \mathcal{E}, P, p) \longrightarrow (\mathcal{S}, \mathcal{E}, P', p_{next})$ *then for all* $\mathcal{S}'$, $(\mathcal{S} \cup \mathcal{S}', \mathcal{E}, P, p) \longrightarrow (\mathcal{S} \cup \mathcal{S}', \mathcal{E}, P', p_{next})$.
2. (Stack Shrinkage). If $(\mathcal{S}, \mathcal{E}, P, p) \longrightarrow (\mathcal{S}, \mathcal{E}, P', p_{next})$, *and* $(\mathcal{S}, \mathcal{E}, P_{clo}, p_{body}) \longrightarrow^n (\mathcal{S}, \mathcal{E}, P'_{clo}, r)$, *then* $(\mathcal{S}', \mathcal{E}, P, p) \longrightarrow (\mathcal{S}', \mathcal{E}, P', p_{next})$, *where* $\mathcal{S}' = \mathcal{S} - \{\langle \lambda x.\, p_{body}, P_{clo}\rangle\}$, *for any* x.

Proof. Follows by induction on the respective derivations. □

Lemma B.3 (Stack Shrinkage). *If* $(\mathcal{S}, \mathcal{E}, P, p) \longrightarrow^n (\mathcal{S}, \mathcal{E}, P', r)$ *then* $(\mathcal{S}', \mathcal{E}, P, p) \longrightarrow^n (\mathcal{S}', \mathcal{E}, P', r)$, *where* $\mathcal{S}' = \mathcal{S} - \{\langle \lambda x.\, p, P\rangle\}$, *for any* x.

Proof. Follows directly by Lemma B.2[2]. □

Lemma B.4 (Simulation: 1-step). *If* $\mathrm{E} \sqsubseteq_p (\mathcal{E}, P)$ *and* $(\mathrm{E}, p) \longrightarrow (\mathrm{E}', p')$ *then* $(\emptyset, \mathcal{E}, P, p) \longrightarrow (\emptyset, \mathcal{E}, P', p')$ *and* $\mathrm{E}' \sqsubseteq_{p'} (\mathcal{E}, P')$.

Proof. The proof proceeds by induction on the derivation of $(\mathrm{E}, p) \longrightarrow (\mathrm{E}', p')$. Following are the possible semantics rules at the root of its derivation:

1. `let`. Hence, $p = (\mathsf{let}\ x = \eta\ \mathsf{in}\ p_{next})$, for some x, η and p_{next}, and $\mathrm{E}' = \mathrm{E} \cup \{x \mapsto \langle \eta, \mathrm{E}\rangle\}$ and $p' = p_{next}$.
By hypothesis, Definition 4.4 and *let* rule, $(\emptyset, \mathcal{E}, P, p) \longrightarrow (\emptyset, \mathcal{E}, P', p')$ such that $x \mapsto \langle \eta, P\rangle \in \mathcal{E}$ and $P' = P \wedge (x = \eta)$.
By Lemma 3.4[2] (E', p') is well-formed. By hypothesis and Definition 4.4, $(\mathcal{E}, P', p')$ is well-formed.
By hypothesis, we know $\mathrm{E} \subseteq [\![\mathcal{E}]\!]$ and $\mathrm{E} \vdash P$. Also, we know (E, p) is well-formed, implying $\langle \eta, \mathrm{E}\rangle$ is closed. Then by Definition 4.7[2], $x \mapsto \langle \eta, \mathrm{E}\rangle \in [\![\mathcal{E}]\!]$. Hence $\mathrm{E}' \subseteq [\![\mathcal{E}]\!]$.
Again knowing (E, p) is well-formed, implies $x \notin dom(\mathrm{E})$ and $x \notin free(\eta)$. Then by Lemma B.1[2], $\mathrm{E}' \vdash P'$.
Then by Definition 4.10, $\mathrm{E}' \sqsubseteq_{p'} (\mathcal{E}, P')$.
2. `if`. Hence $p = \big(\mathsf{let}\ y = (\mathsf{if}\ x\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2)\ \mathsf{in}\ p_{next}\big)$, for some y, x, p_1, p_2 and p_{next}. Without loss in generality, we assume $\|\langle x, \mathrm{E}\rangle\| = \langle \mathsf{true}, \emptyset\rangle$. Hence, $(\mathrm{E}, p_1) \longrightarrow^{n_1} (\mathrm{E}_1, y_1)$, $\mathrm{E}' = \mathrm{E}_1 \cup \{y \mapsto \langle y_1, \mathrm{E}_1\rangle\}$, $p' = p_{next}$, and $n_1 = len(p_1)$.
By hypothesis, Definition 4.4 and *if*, $(\emptyset, \mathcal{E}, P, p) \longrightarrow (\emptyset, \mathcal{E}, P', p')$ such that $P_1 = P \wedge (x = \mathsf{true})$, $P_2 = P \wedge (x = \mathsf{false})$, $(\emptyset, \mathcal{E}, P_1, p_1) \longrightarrow^{n_1} (\emptyset, \mathcal{E}, P'_1, y_1)$, $(\emptyset, \mathcal{E}, P_2, p_2) \longrightarrow^{n_2} (\emptyset, \mathcal{E}, P'_2, y_2)$, $n_2 = len(p_2)$, $P' = \big(P'_1 \wedge (y = y_1)\big) \vee \big(P'_2 \wedge (y = y_2)\big)$, and $\{y \mapsto \langle y_1, P'_1\rangle, y \mapsto \langle y_2, P'_2\rangle\} \subseteq \mathcal{E}$.
By premise, (E, p) is well-formed, implying (E, p_1) is well-formed as well. By Definition 4.4, $(\mathcal{E}, P_1, p_1)$ is well-formed.
By assumption and Lemma B.1[3], $\mathrm{E} \vdash P_1$. Hence, by Definition 4.10, $\mathrm{E} \sqsubseteq_{p_1} (\mathcal{E}, P_1)$. Then applying the induction hypothesis n_1 times, we get $\mathrm{E}_1 \sqsubseteq_{y_1} (\mathcal{E}, P'_1)$.
By Lemma 3.4[2], (E', p') is well-formed. By hypothesis and Definition 4.4, $(\mathcal{E}, P', p')$ is also well-formed.
By Definition 4.7[2], $y \mapsto \langle y_1, \mathrm{E}_1\rangle \in [\![\mathcal{E}]\!]$. Hence $\mathrm{E}' \subseteq [\![\mathcal{E}]\!]$.
Since, E' is canonical, that is, $\mathrm{E}_1 \cup \{y \mapsto \langle y_1, \mathrm{E}_1\rangle\}$ is a set of single-valued mappings, $y \notin dom(\mathrm{E}_1)$; and, since $\langle y_1, \mathrm{E}_1\rangle$ is closed, $y \notin free(y_1)$. Then by Lemma B.1[2], $\mathrm{E}' \vdash P'_1 \wedge (y = y_1)$, that is, $\mathrm{E}' \vdash P'$.
Hence by Definition 4.10, $\mathrm{E}' \sqsubseteq_{p'} (\mathcal{E}, P')$.
3. `app`. Hence, $p = (\mathsf{let}\ r = f\ x'\ \mathsf{in}\ p_{next})$, for some r, f, x and p_{next}. As per `app`, $\mathrm{E}(f)^{+_\lambda} = \langle \mathcal{F}, \mathrm{E}_f\rangle$, $\mathcal{F} = \lambda x.\, p_{body}$, $\mathrm{E}'_f = \mathrm{E}_f \cup \{x \mapsto \langle x', \mathrm{E}\rangle\}$, $(\mathrm{E}'_f, p_{body}) \longrightarrow^n (\mathrm{E}''_f, r')$, $r' = \lfloor p_{body} \rfloor$, $n = len(p_{body})$. $\mathrm{E}' = \mathrm{E} \cup \{r \mapsto \langle r', \mathrm{E}''_f\rangle\}$, and $p' = p_{next}$.
By hypothesis and Lemma 4.8[2], $\mathrm{E}_f \subseteq [\![\mathcal{E}]\!]$, and there exists P_f, such that $\langle \mathcal{F}, P_f\rangle \in \mathcal{E}(f)^{+_\lambda}$ and $\mathrm{E}_f \vdash P_f$. Then by hypothesis, Definition 4.4, and *app*, $\big(\{\langle \mathcal{F}, P_f\rangle\}, \mathcal{E}, P_f, p_{body}\big) \longrightarrow^n \big(\{\langle \mathcal{F}, P_f\rangle\}, \mathcal{E}, P''_f, r'\big)$ such that $\big\{x \mapsto \langle x', P\rangle, r \mapsto \langle r', P''_f\rangle\big\} \subseteq \mathcal{E}$; and, $P' = P$ and $p' = p_{next}$.
By Corollary B.3, $(\emptyset, \mathcal{E}, P_f, p_{body}) \longrightarrow^n (\emptyset, \mathcal{E}, P''_f, r')$. Hence by Definition 4.9, $(\mathcal{E}, P_f, p_{body})$ is well-formed. By premise and Lemma A.4, $(\mathrm{E}'_f, p_{body})$ is well-formed as well. Also by Definition 4.7[2], $x \mapsto \langle x', \mathrm{E}\rangle \in [\![\mathcal{E}]\!]$, implying $\mathrm{E}'_f \subseteq [\![\mathcal{E}]\!]$. By Lemma B.1[1] $\mathrm{E}'_f \vdash P_f$. Hence $\mathrm{E}'_f \sqsubseteq_{p_{body}} (\mathcal{E}, P_f)$.
Then applying the induction hypothesis n times, we get $\mathrm{E}''_f \sqsubseteq_{r'} (\mathcal{E}, P''_f)$.
By Lemma 3.4[2], (E', p') is well-formed. By hypothesis and Definition 4.4, $(\mathcal{E}, P', p')$ is also well-formed.
By Definition 4.7[2], $r \mapsto \langle r', \mathrm{E}''_f\rangle \in [\![\mathcal{E}]\!]$, implying $\mathrm{E}' \subseteq [\![\mathcal{E}]\!]$.
By Lemma B.1[1], $\mathrm{E}' \vdash P$, that is, $\mathrm{E}' \vdash P'$.
Hence by Definition 4.10, $\mathrm{E}' \sqsubseteq_{p'} (\mathcal{E}, P')$.

□

Lemma 4.11 (Simulation). *If* $\mathrm{E} \sqsubseteq_p (\mathcal{E}, P)$ *and* $(\mathrm{E}, p) \longrightarrow^n (\mathrm{E}', p')$ *then* $(\emptyset, \mathcal{E}, P, p) \longrightarrow^n (\emptyset, \mathcal{E}, P', p')$ *and* $\mathrm{E}' \sqsubseteq_{p'} (\mathcal{E}, P')$.

Proof. Follows from Lemma B.4. □

Lemma B.5 (Existence of Derivation: 1-step). *For all 4-tuples* $(\mathcal{S}, \mathcal{E}, P, p)$*, where* $p \neq x$*, for any* x*, there exists* $\mathcal{S}'$*,* $\mathcal{E}'$*,* P' *and* p'*, such that* $(\mathcal{S}, \mathcal{E}, P, p) \longrightarrow (\mathcal{S}', \mathcal{E}', P', p')$.

Proof. By induction on elements of the set, $absClosures(\mathcal{E}, P, p) - \mathcal{S}$, we assume the lemma holds for all its subsets. By premise $p \neq x$, for any x; hence, following are possible outermost structures for p:

1. *let*. $p = (\mathsf{let}\ x = \eta\ \mathsf{in}\ p_{next})$, for some x, η and p_{next}. Directly by *let*, $(\mathcal{S}, \mathcal{E}, P, p) \longrightarrow \big(\mathcal{S}, \mathcal{E} \cup \{x \mapsto \langle \eta, P\rangle\}, P \wedge (x = \eta), p_{next}\big)$.
2. *if*. $p = \big(\mathsf{let}\ y = (\mathsf{if}\ x\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2)\ \mathsf{in}\ p_{next}\big)$, for some y, x, p_1, p_2 and p_{next}.
   Let $P_1 = P \wedge (x = \mathsf{true})$ and $P_2 = P \wedge (x = \mathsf{false})$. Then $absClosures(\mathcal{E}, P_1, p_1) - \mathcal{S} \subseteq absClosures(\mathcal{E}, P, p) - \mathcal{S}$ and $absClosures(\mathcal{E}, P_2, p_2) - \mathcal{S} \subseteq absClosures(\mathcal{E}, P, p) - \mathcal{S}$. By induction hypothesis there exists $\mathcal{E}_1$, P_1', y_1 and n_1 such that $(\mathcal{S}, \mathcal{E}, P_1, p_1) \longrightarrow^{n_1} (\mathcal{S}, \mathcal{E}_1, P_1', \eta_1)$. Analogously, there exists $\mathcal{E}_2$, P_2', y_2 and n_2 such that $(\mathcal{S}, \mathcal{E}, P_2, p_2) \longrightarrow^{n_2} (\mathcal{S}, \mathcal{E}_2, P_2', y_2)$.
   Hence, by *if*, $(\mathcal{S}, \mathcal{E}, P, p) \longrightarrow \big(\mathcal{S}, \mathcal{E}_1 \cup \{y \mapsto \langle y_1, P_1'\rangle\} \cup \mathcal{E}_2 \cup \{y \mapsto \langle y_2, P_2'\rangle\}, P', p_{next}\big)$, where $P' = \big(P_1' \wedge (y = y_1)\big) \vee \big(P_2' \wedge (y = y_2)\big)$.
3. *app*. $p = (\mathsf{let}\ r = f\ x'\ \mathsf{in}\ p_{next})$, for some r, f, x' and p_{next}.
   Let $\overline{\mathcal{E}(f)^{+\lambda} = \{\langle \mathcal{F}_k, P_k\rangle\}}$, $\overline{\mathcal{F}_k = \lambda x_k.\, p_k}$, $\overline{\mathcal{S}_k = \mathcal{S} \cup \{\langle \mathcal{F}_k, P_k\rangle\}}$, and $\overline{\mathcal{E}_k = \mathcal{E} \cup \{x_k \mapsto \langle x', P\rangle\}}$. Now for each $i \in [1, k]$ there are two possible cases:
   (a) $\langle \mathcal{F}_i, P_i\rangle \in \mathcal{S}$. Then, $\textsc{call}\big(\mathcal{S}, \langle \mathcal{F}_i, P_i\rangle\big) = \lfloor p_i \rfloor = r_i$, for some r_i'. By reflexivity, $(\mathcal{S}_i, \mathcal{E}_i, P_i, r_i') \longrightarrow^0 (\mathcal{S}_i, \mathcal{E}_i', P_i', r_i')$ such that $\mathcal{E}_i' = \mathcal{E}_i$ and $P_i' = P_i$.
   (b) $\langle \mathcal{F}_i, P_i\rangle \notin \mathcal{S}$. Then, $\textsc{call}\big(\mathcal{S}, \langle \mathcal{F}_i, P_i\rangle\big) = p_i$. Note, $absClosures(\mathcal{E}_i, P_i, p_i) - \mathcal{S}_i \subseteq absClosures(\mathcal{E}, P, p) - \mathcal{S}$. Then by induction hypothesis, there exists $\mathcal{E}_i'$, P_i', r_i' and n_i such that $(\mathcal{S}_i, \mathcal{E}_i, P_i, p_i) \longrightarrow^{n_i} (\mathcal{S}_i, \mathcal{E}_i', P_i', r_i')$.
   Then by *app*, $\big(\mathcal{S}, \mathcal{E}, P, p\big) \longrightarrow \Big(\mathcal{S}, \mathcal{E} \cup \bigcup_{1 \le i \le k} \mathcal{E}_i' \cup \big\{r \mapsto \langle r_i', P_i'\rangle\big\}, P, p_{next}\Big)$.

□

Lemma 4.3 (Existence of Derivation). *For all 4-tuples* $(\mathcal{S}, \mathcal{E}, P, p)$ *there exists* $\mathcal{S}'$*,* $\mathcal{E}'$*,* P'*,* r' *and* n*, such that* $(\mathcal{S}, \mathcal{E}, P, p) \longrightarrow^n (\mathcal{S}', \mathcal{E}', P', r')$.

Proof. Follows from Lemma B.5. □

We write $pairs\ (\mathcal{S}, \mathcal{E}, P, p)$ as a shorthand for $pairs\ (\mathcal{E}, P, p) \cup \big\{(P', \mathcal{F}') \mid \langle \mathcal{F}', P'\rangle \in \mathcal{S}\big\}$.

Lemma B.6 (Upper Bound on Pairs of Predicates and Sub-Programs: 1-step). *If* $(\mathcal{S}', \mathcal{E}', P', p')$ *is a node in the derivation tree of* $(\mathcal{S}, \mathcal{E}, P, p) \longrightarrow (\mathcal{S}'', \mathcal{E}'', P'', p'')$ *then* $pairs\ (\mathcal{S}, \mathcal{E}, P, p) \supseteq pairs\ (\mathcal{S}', \mathcal{E}', P', p')$.

Proof. Proceeding by induction on the derviation of $(\mathcal{S}, \mathcal{E}, P, p) \longrightarrow (\mathcal{S}'', \mathcal{E}'', P'', p'')$, following are the possible semantics rules at its root:

1. *let*. $p = (\mathsf{let}\ x = \eta\ \mathsf{in}\ p_{next})$, for some x, η and p_{next}. Hence, $\mathcal{S}'' = \mathcal{S}$, $\mathcal{E}'' = \mathcal{E} \cup \{x \mapsto \langle \eta, P\rangle\}$, $P'' = P \wedge (x = \eta)$, and $p'' = p_{next}$.
   Directly, $pairs\ (\mathcal{S}, \mathcal{E}, P, p) \supseteq pairs\ (\mathcal{S}'', \mathcal{E}'', P'', p'')$.
2. *if*. $p = \big(\mathsf{let}\ y = (\mathsf{if}\ x\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2)\ \mathsf{in}\ p_{next}\big)$, for some y, x, p_1, p_2 and p_{next}. As per *if*, $\mathcal{S}'' = \mathcal{S}$, $\mathcal{E}'' = \mathcal{E}_1 \cup \{y \mapsto \langle y_1, P_1'\rangle\} \cup \mathcal{E}_2 \cup \{y \mapsto \langle y_2, P_2'\rangle\}$, $p'' = p_{next}$ such that $P_1 = P \wedge (x = \mathsf{true})$, $P_2 = P \wedge (x = \mathsf{false})$, $(\mathcal{S}, \mathcal{E}, P_1, p_1) \longrightarrow^{n_1} (\mathcal{S}, \mathcal{E}_1, P_1', y_1)$, $(\mathcal{S}, \mathcal{E}, P_2, p_2) \longrightarrow^{n_2} (\mathcal{S}, \mathcal{E}_2, P_2', y_2)$, and $P'' = \big(P_1' \wedge (y = y_1)\big) \vee \big(P_2' \wedge (y = y_2)\big)$.
   Directly, $pairs\ (\mathcal{S}, \mathcal{E}, P, p) \supseteq pairs\ (\mathcal{S}, \mathcal{E}, P_1, p_1)$ and $pairs\ (\mathcal{S}, \mathcal{E}, P, p) \supseteq pairs\ (\mathcal{S}, \mathcal{E}, P_2, p_2)$. The result then holds by induction hypothesis.
3. *app*. $p = (\mathsf{let}\ r = f\ x'\ \mathsf{in}\ p_{next})$, for some r, f, x' and p_{next}. As per *app*, $\mathcal{S}'' = \mathcal{S}$, $\mathcal{E}'' = \mathcal{E} \cup \bigcup_{1 \le i \le k} \mathcal{E}_i' \cup \big\{r \mapsto \langle r_i, P_i''\rangle\big\}$, $\overline{P'' = P}$, and $p'' = p_{next}$ such that $\overline{\mathcal{E}(f)^{+\lambda} = \{\langle \mathcal{F}_k, P_k\rangle\}}$, $\overline{\mathcal{F}_k = \lambda x_k.\, p_k}$, and for all $i \in [1, k]$, $\mathcal{S}_i = \mathcal{S} \cup \{\langle \mathcal{F}_i, P_i\rangle\}$, $\textsc{call}\big(\mathcal{S}, \langle \mathcal{F}_i, P_i\rangle\big) = p_i'$, $\mathcal{E}_i = \mathcal{E} \cup \big\{x_i \mapsto \langle x', P\rangle\big\}$, $(\mathcal{S}_i, \mathcal{E}_i, P_i, p_i') \longrightarrow^{n_i} (\mathcal{S}_i, \mathcal{E}_i', P_i', r_i)$.
   Directly, for all $i \in [1, k]$, $pairs\ (\mathcal{S}, \mathcal{E}, P, p) \supseteq pairs\ (\mathcal{S}_i, \mathcal{E}_i, P_i, p_i')$. The result then holds by induction hypothesis.

□

Corollary B.7 (Upper Bound on Pairs of Predicates and Sub-Programs: n-step). *If* $(\mathcal{S}', \mathcal{E}', P', p')$ *is a node in the derivation tree of* $(\mathcal{S}, \mathcal{E}, P, p) \longrightarrow^n (\mathcal{S}'', \mathcal{E}'', P'', p'')$ *then* $pairs\ (\mathcal{S}, \mathcal{E}, P, p) \supseteq pairs\ (\mathcal{S}', \mathcal{E}', P', p')$.

Proof. Follows directly by Lemma B.6. □

$absClosures(\mathcal{E}, P, p) = \big\{\langle \eta', P'\rangle \mid (P', \eta') \in pairs\ (\mathcal{E}, P, p)\big\}$.
We write $absClosures(\mathcal{S}, \mathcal{E}, P, p)$ as a shorthand for $\mathcal{S} \cup absClosures(\mathcal{E}, P, p)$.

Lemma B.8 (Bounds on Abstract Stack). *If* $(\mathcal{S}', \mathcal{E}', P', p')$ *is a node in the derivation tree of* $(\mathcal{S}, \mathcal{E}, P, p) \longrightarrow^n (\mathcal{S}'', \mathcal{E}'', P'', p'')$ *then* $\mathcal{S} \subseteq \mathcal{S}'$ *and* $\mathcal{S}' \subseteq absClosures(\mathcal{S}, \mathcal{E}, P, p)$.

Proof. Follows by induction on the derivation of $(\mathcal{S}, \mathcal{E}, P, p) \longrightarrow^n (\mathcal{S}'', \mathcal{E}'', P'', p'')$. □

We write $locals(\mathcal{E}, p)$ as shorthand for $\{\overline{x_k}\} \cup \bigcup_{1 \le i \le k} locals(\eta_i) \cup locals(p)$, where $\mathcal{E} = \{\overline{x_k \mapsto \langle \eta_k, P_k\rangle}\}$, for some $\overline{P_k}$.

Lemma B.9 (Bounds on Abstract Environment). *If* $(\mathcal{S}', \mathcal{E}', P', p')$ *is a node in the derivation tree of* $(\mathcal{S}, \mathcal{E}, P, p) \longrightarrow^n (\mathcal{S}'', \mathcal{E}'', P'', p'')$ *then* $\mathcal{E} \subseteq \mathcal{E}' \subseteq \mathcal{E}''$ *and* $\mathcal{E}' \subseteq locals(\mathcal{E}, p) \times absClosures(\mathcal{E}, P, p)$.

Proof. Follows by induction on the derivation of $(\mathcal{S}, \mathcal{E}, P, p) \longrightarrow^n (\mathcal{S}'', \mathcal{E}'', P'', p'')$. □

Lemma 4.16 (Upper Bound on the Abstract Environment). *For a program* p*, if* $\mathcal{E} \subseteq locals(p) \times absClosures(\mathsf{true}, p)$ *and* $(\emptyset, \mathcal{E}, \mathsf{true}, p) \longrightarrow^n (\emptyset, \mathcal{E}', P, r)$ *then* $\mathcal{E}' \subseteq locals(p) \times absClosures(\mathsf{true}, p)$.

Proof. Follows from Lemmas B.9. □

Lemma 4.17 (Upper Bound on (Predicate, Redex) Pairs). *If* $(\mathcal{S}', \mathcal{E}', P', p')$ *is a node in the derivation tree of* $(\mathcal{S}, \mathcal{E}, P, p) \longrightarrow^n (\mathcal{S}'', \mathcal{E}'', P'', p'')$ *then* $(P', p') \in pairs\ (\mathcal{E}, P, p)$.

Proof. Follows by Lemma B.7. □

Lemma B.10 (Upper Bound on Distinct Nodes). *The number of distinct nodes in the computation of* $nuggetizer(p)$ *is* $O(n! \cdot n^3)$*, where* $n = |p|$.

$\eta ::= \ldots \mid loc$	*lazy concrete value*
$\kappa ::= \ldots \mid \mathsf{ref}\ x \mid !x \mid x := x$	*atomic computation*
$\mathrm{H} ::= \{\overline{loc \mapsto \langle \eta, \mathrm{E} \rangle}\}$	*heap*
$\mathcal{H} ::= \{\overline{loc \mapsto \langle \eta, P \rangle}\}$	*abstract heap*
$\mathcal{S} ::= \{\overline{\langle \mathcal{F}, P \rangle^{\mathcal{H}}_{\mathcal{H}}}\}$	*abstract stack*

Figure 7. Syntax Grammar with Mutable State

Proof. Let $(\mathcal{S}, \mathcal{E}, P, p)$ be a node in the computation of $nuggetizer(p)$. By Lemma B.8 and Lemma 4.15[2], number of distinct possibilities for $\mathcal{S}$, by simple combinatorics, is $O(n!)$. By Lemma B.9 and Lemma 4.15[1,2], number of distinct possibilities for $\mathcal{E}$, again by simple combinatorics, is n^2. By Lemma 4.17 and Lemma 4.15[2], number of distinct possibilities for tuples (P, p) is n. Hence the number of distinct possibilities for the node $(\mathcal{S}, \mathcal{E}, P, p)$, once again by simple combinatorics, is $O(n! \cdot n^2 \cdot n)$, that is, $O(n! \cdot n^3)$. □

Lemma 4.19 (Complexity of the Nuggetizer). *The runtime complexity of the nuggetizer is $O(n! \cdot n^3)$, where $n = |\, p \,|$.*

Proof. Follows by Lemma B.10. □

$$\frac{}{(\mathrm{E},\mathrm{H},\mathsf{let}\ x=\eta\ \mathsf{in}\ p)\longrightarrow\big(\mathrm{E}\cup\{x\mapsto\langle\eta,\mathrm{E}\rangle\},\mathrm{H},p\big)}\,\mathtt{let}^{\mathrm{H}}$$

$$\frac{[\![\langle x,\mathrm{E}\rangle]\!]=\langle b_i,\emptyset\rangle\qquad(b_1,b_2)=(\mathsf{true},\mathsf{false})\qquad(\mathrm{E},\mathrm{H},p_i)\longrightarrow^{n}(\mathrm{E}',\mathrm{H}',y')}{\big(\mathrm{E},\mathrm{H},\mathsf{let}\ y=(\mathsf{if}\ x\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2)\ \mathsf{in}\ p\big)\longrightarrow\big(\mathrm{E}'\cup\{y\mapsto\langle y',\mathrm{E}'\rangle\},\mathrm{H}',p\big)}\,\mathtt{if}^{\mathrm{H}}$$

$$\frac{\mathrm{E}(f)^{+_\lambda}=\langle\lambda x.\,p,\mathrm{E}_{clo}\rangle\qquad\big(\mathrm{E}_{clo}\cup\{x\mapsto\langle x',\mathrm{E}\rangle\},\mathrm{H},p\big)\longrightarrow^{n}(\mathrm{E}',\mathrm{H}',r')}{(\mathrm{E},\mathrm{H},\mathsf{let}\ r=f\ x'\ \mathsf{in}\ p_{next})\longrightarrow\big(\mathrm{E}\cup\{r\mapsto\langle r',\mathrm{E}'\rangle\},\mathrm{H}',p_{next}\big)}\,\mathtt{app}^{\mathrm{H}}$$

$$\frac{\textit{loc}\text{ is a fresh heap location}}{(\mathrm{E},\mathrm{H},\mathsf{let}\ y=\mathsf{ref}\ x\ \mathsf{in}\ p)\longrightarrow\big(\mathrm{E}\cup\{y\mapsto\langle loc,\mathrm{E}\rangle\},\mathrm{H}\cup\{loc\mapsto\langle x,\mathrm{E}\rangle\},p\big)}\,\mathtt{ref}^{\mathrm{H}}$$

$$\frac{[\![\langle y,\mathrm{E}\rangle]\!]=\langle loc,\emptyset\rangle}{(\mathrm{E},\mathrm{H},\mathsf{let}\ z=!y\ \mathsf{in}\ p)\longrightarrow\big(\mathrm{E}\cup\{z\mapsto\mathrm{H}(loc)\},\mathrm{H},p\big)}\,\mathtt{deref}^{\mathrm{H}}$$

$$\frac{[\![\langle y,\mathrm{E}\rangle]\!]=\langle loc,\emptyset\rangle}{\big(\mathrm{E},\mathrm{H},\mathsf{let}\ z=(y:=x)\ \mathsf{in}\ p\big)\longrightarrow\big(\mathrm{E},\mathrm{H}[loc\mapsto\langle x,\mathrm{E}\rangle],\mathsf{let}\ z=x\ \mathsf{in}\ p\big)}\,\mathtt{update}^{\mathrm{H}}$$

Figure 8. Operational Semantics Rules with Mutable State

$$\frac{}{\big(\mathcal{S},\mathcal{E},\mathcal{H},P,\mathsf{let}\ x=\eta\ \mathsf{in}\ p\big)\longrightarrow\big(\mathcal{S},\mathcal{E}\cup\{x\mapsto\langle\eta,P\rangle\},\mathcal{H},P\wedge(x=\eta),p\big)}\,\textit{let}^{\mathcal{H}}$$

$$\frac{\begin{array}{c}P_1=P\wedge(x=\mathsf{true})\qquad(\mathcal{S},\mathcal{E},\mathcal{H}_1,P_1,p_1)\longrightarrow^{n_1}(\mathcal{S}_1,\mathcal{E}_1,\mathcal{H}_1',P_1',y_1)\\ P_2=P\wedge(x=\mathsf{false})\qquad(\mathcal{S},\mathcal{E},\mathcal{H}_2,P_2,p_2)\longrightarrow^{n_2}(\mathcal{S}_2,\mathcal{E}_2,\mathcal{H}_2',P_2',y_2)\qquad P'=\big(P_1'\wedge(y=y_1)\big)\vee\big(P_2'\wedge(y=y_2)\big)\end{array}}{\big(\mathcal{S},\mathcal{E},\mathcal{H},P,\mathsf{let}\ y=(\mathsf{if}\ x\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2)\ \mathsf{in}\ p\big)\longrightarrow\big(\mathcal{S}_1\uplus\mathcal{S}_2,\mathcal{E}_1\cup\{y\mapsto\langle y_1,P_1'\rangle\}\cup\mathcal{E}_2\cup\{y\mapsto\langle y_2,P_2'\rangle\},\mathcal{H}_1'\cup\mathcal{H}_2',P',p\big)}\,\textit{if}^{\mathcal{H}}$$

$$\frac{\begin{array}{c}\mathcal{E}(f)^{+_\lambda}=\{\overline{\langle\mathcal{F}_k,P_k\rangle}\}\qquad\overline{\mathcal{F}_k=\lambda x_k.\,p_k}\qquad k>0\\ \forall 1\le i\le k\\ \textit{CALL}\big(\mathcal{S},\mathcal{E},\mathcal{H},P,\langle\mathcal{F}_i,P_i\rangle\ x'\big)\longrightarrow^{n_i}\big(\mathcal{S}_i,\mathcal{E}_i,\mathcal{H}_i,P_i',r_i'\big)\qquad \textit{FIX}_{\textit{heap}}\big(\mathcal{S}_i,\mathcal{E}_i,\mathcal{H}_i,\langle\mathcal{F}_i,P_i\rangle\big)=\big(\mathcal{S}_i',\mathcal{E}_i',\mathcal{H}_i'\big)\qquad \textit{POP}_{\mathcal{S}}\big(\mathcal{S}_i',\langle\mathcal{F}_i,P_i\rangle\big)=\mathcal{S}_i''\end{array}}{\big(\mathcal{S},\mathcal{E},\mathcal{H},P,\mathsf{let}\ r=f\ x'\ \mathsf{in}\ p_{next}\big)\longrightarrow\left(\mathcal{S}\cup\bigoplus_{1\le i\le k}\mathcal{S}_i'',\mathcal{E}\cup\bigcup_{1\le i\le k}\mathcal{E}_i'\cup\{r\mapsto\langle r_i',P_i'\rangle\},\bigcup_{1\le i\le k}\mathcal{H}_i',P,p_{next}\right)}\,\textit{app}^{\mathcal{H}}$$

$$\frac{\langle\mathcal{F},P\rangle^{\mathcal{H}_a}_{\mathcal{H}_r}\in\mathcal{S}\qquad\mathcal{H}_r=\mathcal{H}}{\textit{FIX}_{\textit{heap}}\big(\mathcal{S},\mathcal{E},\mathcal{H},\langle\mathcal{F},P\rangle\big)=(\mathcal{S},\mathcal{E},\mathcal{H})}\qquad\qquad\frac{\begin{array}{c}\langle\mathcal{F},P\rangle^{\mathcal{H}_a}_{\mathcal{H}_r}\in\mathcal{S}\qquad\mathcal{S}'=\mathcal{S}\uplus\big\{\langle\mathcal{F},P\rangle^{\mathcal{H}_a}_{\mathcal{H}_r\cup\mathcal{H}}\big\}\\ \mathcal{H}_r\neq\mathcal{H}\qquad(\mathcal{S}',\mathcal{E},\mathcal{H}_a,P,p)\longrightarrow^{n}(\mathcal{S}'',\mathcal{E}',\mathcal{H}',P',r')\end{array}}{\textit{FIX}_{\textit{heap}}\big(\mathcal{S},\mathcal{E},\mathcal{H},\langle\mathcal{F},P\rangle\big)=\textit{FIX}_{\textit{heap}}\big(\mathcal{S}'',\mathcal{E}',\mathcal{H}',\langle\mathcal{F},P\rangle\big)}$$

$$\frac{\neg\exists loc.\langle loc,P\rangle\in\mathcal{E}(y)^{+_{loc}}\qquad\textit{loc}\text{ is a fresh heap location}}{\big(\mathcal{S},\mathcal{E},\mathcal{H},P,\mathsf{let}\ y=\mathsf{ref}\ x\ \mathsf{in}\ p\big)\longrightarrow\big(\mathcal{S},\mathcal{E}\cup\{y\mapsto\langle loc,P\rangle\},\mathcal{H}\cup\{loc\mapsto\langle x,P\rangle\},P,p\big)}\,\textit{ref-fresh}^{\mathcal{H}}$$

$$\frac{\{\overline{loc_k}\}=\{loc\mid\langle loc,P\rangle\in\mathcal{E}(y)^{+_{loc}}\}\qquad k>0}{\big(\mathcal{S},\mathcal{E},\mathcal{H},P,\mathsf{let}\ y=\mathsf{ref}\ x\ \mathsf{in}\ p\big)\longrightarrow\big(\mathcal{S},\mathcal{E},\mathcal{H}\cup\{\overline{loc_k}\mapsto\langle x,P\rangle\},P,p\big)}\,\textit{ref-reuse}^{\mathcal{H}}$$

$$\frac{\mathcal{E}(y)^{+_{loc}}=\{\overline{\langle loc_j,P_j\rangle}\}\qquad\mathcal{H}\big(\overline{loc_j}\big)^{+}=\{\overline{\langle\eta_k,P_k\rangle}\}}{\big(\mathcal{S},\mathcal{E},\mathcal{H},P,\mathsf{let}\ z=!y\ \mathsf{in}\ p\big)\longrightarrow\big(\mathcal{S},\mathcal{E}\cup\{z\mapsto\overline{\langle\eta_k,P_k\rangle}\},\mathcal{H},P,p\big)}\,\textit{deref}^{\mathcal{H}}$$

$$\frac{\mathcal{E}(y)^{+_{loc}}=\{\langle loc,P'\rangle\}\qquad\mathcal{H}'=\mathcal{H}[loc\mapsto\langle x,P\rangle]}{\big(\mathcal{S},\mathcal{E},\mathcal{H},P,\mathsf{let}\ z=(y:=x)\ \mathsf{in}\ p\big)\longrightarrow\big(\mathcal{S},\mathcal{E},\mathcal{H}',P,\mathsf{let}\ z=x\ \mathsf{in}\ p\big)}\,\textit{strong-update}^{\mathcal{H}}$$

$$\frac{\mathcal{E}(y)^{+_{loc}}=\{\overline{\langle loc_k,P_k\rangle}\}\qquad k\neq 1\qquad\mathcal{H}'=\mathcal{H}\cup\{\overline{loc_k}\mapsto\langle x,P\rangle\}}{\big(\mathcal{S},\mathcal{E},\mathcal{H},P,\mathsf{let}\ z=(y:=x)\ \mathsf{in}\ p\big)\longrightarrow\big(\mathcal{S},\mathcal{E},\mathcal{H}',P,\mathsf{let}\ z=x\ \mathsf{in}\ p\big)}\,\textit{weak-update}^{\mathcal{H}}$$

Figure 9. Abstract Operational Semantics Rules with Flow-Sensitive Mutable State