The Nuggetizer: Abstracting Away Higher-Orderness for Program Verification *

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

[1] The Johns Hopkins University, Baltimore, MD, USA, {pari,scott}@cs.jhu.edu
[2] The University of Vermont, Burlington, VT, USA, skalka@cs.uvm.edu

Abstract. We develop a static analysis which distills first-order computational structure from higher-order functional programs. The analysis condenses higher-order programs into a first-order rule-based system, a *nugget*, that characterizes all value bindings that may arise from program execution. Theorem provers are limited in their ability to automatically reason about higher-order programs; nuggets address this problem, being inductively defined structures that can be simply and directly encoded in a theorem prover. The theorem prover can then prove non-trivial program properties, such as the range of values assignable to particular variables at runtime. Our analysis is flow- and path-sensitive, and incorporates a novel *prune-rerun* analysis technique to approximate higher-order recursive computations.

Keywords program analysis, higher-order, 0CFA, program verification

1 Introduction

Higher-order functional programming is a powerful programming metaphor, but it is also complex from a program analysis standpoint: the actual low-level operations and the order in which they take place are far removed from the source code. It is the simpler first-order view that is easiest for automated verification methods to be applied to. In this paper we focus on defining a new form of program abstraction which distills the first-order computational structure from higher-order functional programs. The analysis is novel in how it condenses higher-order programs into a first-order inductive system, a *nugget*, which characterizes all value bindings that can result from program execution. Nuggets can be extracted automatically from the program source of any untyped functional language, and without any need for programmer annotation.

A major advantage of the nuggets is that they are inductively defined structures which can be directly expressed as inductive definitions in a theorem prover. So in effect, our analysis produces an output, a *nugget*, which is ideally suited as input to a theorem prover. We use Isabelle/HOL [1] to reason about nuggets in

* This paper has been published in the LNCS proceedings of APLAS 2007: The Fifth ASIAN Symposium on Programming Languagues and Systems.

this paper since it has built-in mechanisms to define and reason about inductively defined first-order entities (although other provers with a similar mechanism, e.g. ACL2 [2], could be employed as well). The theorem prover can then be used to automatically prove desirable properties of the corresponding program. Putting these steps together gives a method for automatically proving complex inductive properties of higher-order programs. The alternative approach to formally prove program properties in a theorem prover involves writing an operational or denotational semantics for the programs and proving facts about those definitions, or using an existing axiomatized programming logic. While these approaches are effective, there is a high user overhead due to all of the program features such as higher-order functions that clutter up the semantics or axioms, and a great deal of time and effort is thus required. Nuggets are not complete in that some program information is abstracted out, but enough information remains for a wide class of program properties to be verified. So, we are trading off the completeness of full verification for the speed and simplicity of partial verification. For concreteness, we focus here on solving the *value range* problem for functional programs—deducing the range of values that integer variables can take on at runtime. While this is a narrow problem it is a non-trivial one, and it serves as a testbed for our approach. Our analysis grew out of a type-and-effect constraint type system [3], and is also related to abstract interpretation [4].

2 Informal Overview

In this section we give an informal description of our analysis. We start by showing the form of the nuggets, their expressiveness, and the technique to prove properties they encapsulate. Then we describe the nugget-generation algorithm. Consider the following simple untyped higher-order program which computes the factorial of 5, using recursion encoded by "self-passing",

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

We want to statically analyze the range of values assignable to the variable n during the course of computation of the above program. Obviously this program will recurse for n from 5 down to 0, each time with the condition $(n\ != 0)$ holding, until finally $n = 0$, thus the range of values assignable to n is $[0, \ldots, 5]$. The particular goal of this paper is to define an analysis to automatically infer basic properties such as value ranges. For non-recursive programs it is not hard to completely track such ranges; the challenge is to track ranges in the presence higher-order recursive functions.

2.1 Nuggets

There is a huge array of potential program abstractions to consider: type systems, abstract interpretations, compiler analyses, etc. All of these can be viewed as abstracting away certain properties from program executions. Type systems

tend to abstract away the control-flow, that is, flow- and path-sensitivity, but retain much of the data-flow including infinite datatype domains; abstract interpretations, on the other hand, generally make finitary abstractions on infinite datatype domains, but tend to preserve flow- and path-sensitivity. Our approach is somewhat unique in that we wish to abstract away only the higher-order nature of functional programs, and preserve as much of the other behavior as possible, including flow- and path-sensitivity, infinite datatype domains, and other inductive program structure.

The core of our analysis is the *nuggetizer*, which automatically extracts *nuggets* from source programs. We begin with a description of the nuggets themselves and their role in proving program properties; subsequently, we discuss the nuggetizing process itself.

Nuggets are purely first-order inductive definitions; they may contain higher-order functions but in a nugget they mean nothing, they are just atomic data. All higher-order flows that can occur in the original program execution are reduced to their underlying first-order actions on data by the *nuggetizer*, the algorithm for constructing nuggets described in the next subsection. We illustrate the form and the features of nuggets by considering the nugget produced by the nuggetizer for 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 higher-order functions in program (1) have been expanded. The mapping $n \mapsto 5$ represents the initial value 5 passed in to the function $(\lambda n. \ldots)$ in program (1). The mapping $n \mapsto (n-1)^{n\,!=0}$ additionally contains a *guard*, $n\ != 0$, which is a precondition on the usage of this mapping, analogous to its role in dictating the course of program (1)'s computation. Note the inductive nature of this mapping: n maps to $n-1$ given the guard $n\ != 0$ holds. The mapping can then be read as: during the course of program (1)'s execution, n may be also bound to $(n_i - 1)$, for some value n_i, such that $n \mapsto n_i$ is an already known binding for n, and the guard $n_i\ != 0$ holds. It corresponds to the fact that the recursive invocation of function $(\lambda n. \ldots)$ at call-site '$(\mathit{fact}\ \mathit{fact})\ (n-1)$' during program (1)'s computation results in $(n_i - 1)$ being the new binding for n, given n is currently bound to n_i and the guard $n_i\ != 0$ holds.

Denotational semantics of nuggets Nuggets are in fact nothing more than inductive definitions of sets of possible values for the variables—the least set of values implied by the mappings such that their guards hold. So, the denotational semantics of a nugget is nothing more than the values given by this inductive definition. The above nugget has the denotation

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

This is because $n \mapsto 0$ does not satisfy the guard $n\ != 0$, implying it cannot be inlined in the right side of mapping $n \mapsto (n-1)^{n\,!=0}$, to generate the mapping

$n \mapsto (-1)$ for n. Notice that the above nugget precisely denotes the range of values assignable to n during the course of program (1)'s computation.

The key soundness property is: a nugget N for a program p must denote each variable x in p to be mapping to *at least* the values that may occur during the run of p. Thus the nugget (2) above serves to soundly establish the range of n to be $[0, \ldots, 5]$ in program (1), which is also precise in this case: n will take on exactly these values at runtime.

Defining and reasoning about nuggets in Isabelle/HOL Properties of a nugget can be manually computed as we did above, but our goal is to automate proofs of such properties. Since nuggets are inductive definitions, any nugget can be automatically translated into an equivalent inductive definition in a theorem prover. The theorem prover can then be used to directly prove, for example, that $0 \leq n \leq 5$ in program (1). The Isabelle/HOL encoding of the above nugget is presented in Section 5. Theorem proving aligns particularly well with nuggets for two reasons: 1) since arbitrary Diophantine equations can be expressed as nuggets there can be no complete decision procedure; and, 2) theorem provers have built-in mechanisms for writing inductive definitions, and proof strategies thereupon.

Two more complex examples 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 $\mathsf{Z} = \lambda f.\, (\lambda x.\, f\ (\lambda y.\, x\ x\ y))\ (\lambda x.\, f\ (\lambda y.\, x\ x\ y))$ to perform recursion. Z is a version of the Y combinator, given by η-expansion on a part of it, to be used in call-by-value evaluations.

$$\mathsf{let}\ f' = \big(\lambda \mathit{fact}.\lambda n.\, \mathsf{if}\ (n\ \mathsf{!=}\ 0)\ \mathsf{then}\ n * \mathit{fact}\ \ (n-1)\ \mathsf{else}\ 1\big)\ \mathsf{in}\ \mathsf{Z}\ f'\ 5\ . \quad (3)$$

The nugget at n as extracted by the nuggetizer is $\{n \mapsto 5, n \mapsto y, y \mapsto (n-1)^{n\,!=0}\}$ which, by transitive closure, maps n equivalently as in nugget (2). The more complex higher-order structure of the above program proves no more demanding to the nuggetizer.

Now, consider another variation of program (1), but with higher-order mutual recursion,

$$\begin{array}{l} \mathsf{let}\ g = \lambda \mathit{fact}'.\, \lambda m.\, \mathit{fact}'\ \mathit{fact}'\ (m-1)\ \mathsf{in} \\ \mathsf{let}\ f = \lambda \mathit{fact}.\, \lambda n.\, \mathsf{if}\ (n\ \mathsf{!=}\ 0)\ \mathsf{then}\ n * g\ \mathit{fact}\ n\ \mathsf{else}\ 1 \\ \mathsf{in}\ f\ f\ 5\ . \end{array} \quad (4)$$

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

$$\text{Nugget: } \{n \mapsto 5, m \mapsto n^{n\,!=0}, n \mapsto (m-1)\}\ . \quad (5)$$

The mutually recursive computational structure between the functions $(\lambda n.\ \ldots)$ and $(\lambda m.\ \ldots)$ in the above program is reflected as a mutual dependency between the mappings of 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 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 $n\ != 0$ and hence cannot be used to generate the mapping $m \mapsto 0$.

External inputs The above examples assume a concrete value, such as 5, to be flowing into functions. In general the value can come from an 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 a symbolic placeholder, **inp**, corresponding to the input value. Now consider the nugget, $\{n \mapsto \mathbf{inp}^{\mathbf{inp} \geq 0}, n \mapsto (n-1)^{n\ != 0}\}$, extracted from a program which invokes the factorial function, from the above examples, on **inp** under the guard $\mathbf{inp} \geq 0$. The bindings for n in the denotation of this nugget lie in the *symbolic* range $[0, \ldots, \mathbf{inp}]$, which, along with the guard $\mathbf{inp} \geq 0$, establishes that n is never assigned a negative number over any program run.

2.2 The Nuggetizer

We now describe the process for creating nuggets, the *nuggetizer*. It constructs the nugget via a collecting semantics—the nugget is incrementally accumulated over an abstract execution of the program.

The challenge Our goal is to abstract away only the higher-orderness of programs and preserve as much of the other behavior as possible including flow- and path-sensitivity, and infinite datatype domains. In other words, we aim to define an abstract operational semantics (AOS) which structurally aligns very closely with the concrete operational semantics (COS) of programs. This is a non-trivial problem as concrete executions of programs with recursively invoked functions may not terminate; however, abstract executions must always terminate in order to achieve a decidable static analysis. Further, recursive function invocations need not be immediately apparent in the source code of higher-order programs due to the use of Y-combinators, etc., making them hard to detect and even harder to soundly approximate while preserving much of their inductive structure at the same time.

The AOS Our AOS is a form of environment-based operational semantics, wherein the environment collects abstract mappings such as $n \mapsto (n-1)^{n\ != 0}$; the environment is monotonically increasing, in that, mappings are only added, and never removed from the environment. The AOS closely follows the control-flow of the COS, that is, the AOS is flow-sensitive. Further, the AOS keeps track of all 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; the AOS is path-sensitive. So for example, when analyzing the `then`-branch of the above programs, the AOS tags all mappings with the active guard $n\ != 0$, before adding them to the environment, as for mappings $n \mapsto (n-1)^{n\ != 0}$ and $m \mapsto n^{n\ != 0}$ in nuggets (2) and (5), respectively.

The prune-rerun technique A novel *prune-rerun* technique at the heart of the nuggetizer is pivotal to ensuring its convergence and soundness in presence of a flow-sensitive AOS. All recursive function invocations are *pruned*, possibly at the expense of some soundness, to ensure convergence of the AOS. The AOS is then repeatedly *rerun* on the program, a (provably) finite number of times, while accumulating mapping across subsequent runs, until all soundness lost by way of pruning, if any, is regained.

Finiteness of the abstract environment The domain and range of all mappings added to the environment during abstract execution, e.g. n, y, m, 5, $(n-1)$, $(m-1)$ and $(n\ != 0)$ in the above shown nuggets, are fragments that *directly* appear in corresponding source programs—no new subexpressions are *ever* created in the nuggets, either by substitution or otherwise. For this reason, the maximum number of distinct mappings in the abstract environment of the AOS is finite for any given program. Since the nuggetizer accumulates mappings across subsequent runs of the AOS on a given program, all feasible mappings must eventually appear in the environment after some finite number of reruns. Thus the environment must stop growing and the analysis must terminate, producing the nugget of the program.

An Illustration We now discuss the abstract execution of program (1) placed in an A-normal form [5], for technical convenience, as follows:

$$\begin{array}{l}\mathsf{let}\ f = \lambda \mathit{fact}.\,\lambda n.\,\mathsf{let}\ r = \mathsf{if}\ (n\ != 0)\ \mathsf{then}\ \mathsf{let}\ r' = \mathit{fact}\ \mathit{fact}\ (n-1) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ n * r' \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ 1 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ r \\ \mathsf{in}\ f\ f\ 5\ .\end{array} \tag{6}$$

The abstract execution of the above program closely follows the control-flow of its concrete execution. It is summarized in Fig. 1. The column labeled "Stack" indicates the state of the abstract 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 "Curr. Guard(s)" column , where "Curr." is short for "Current", indicates the guard(s) in force, if any, at the corresponding step. The "Redex" column holds the redex of the abstract execution at the end of the corresponding step. We now highlight the significant steps in Fig. 1.

Setup and forking the branches During step 1, the mapping of f to $(\lambda \mathit{fact}.\,\lambda n \ldots)$ is collected in the environment, and then the function $(\lambda \mathit{fact}.\,\lambda n \ldots)$ is invoked during step 2 by placing it in the abstract stack and collecting the mapping $\mathit{fact} \mapsto f$. Step 3 pops the stack and results in the function application '$(\lambda n \ldots)\ 5$' being the redex at the end of step 3. Step 4 invokes the function $(\lambda n \ldots)$ by placing it in the abstract stack and collecting the mapping $n \mapsto 5$. At step 5

#	*Stack*	*Collected Mappings*	*Curr. 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 f
2	$(\lambda fact.$ $\lambda n\ldots)$	$fact \mapsto f$		$\big((\lambda n\ldots)\big)\ 5$	pop $(\lambda fact.\lambda n\ldots)$
3				$(\lambda n\ldots)\ 5$	invoke $(\lambda n\ldots)$
4	$(\lambda n\ldots)$	$n \mapsto 5$		let $r =$ (if $(n\,!=0)$ $\ldots$) in r	fork execution

	T/F	T	F	T	F	T	F	T	F
5	$(\lambda n\ldots)$			$n\,!=0$	$n==0$	let $r' =$ $fact\ fact\ (n-1)$ in $n * r'$	1	invoke $fact$	nop
6	$(\lambda n\ldots)$	$fact \mapsto fact^{n\,!=0}$		$n\,!=0$	$n==0$	let $r' =$ $(\lambda n\ldots)\ (n-1)$ in $n * r'$	1	prune re-activation of $(\lambda n\ldots)$	nop
7	$(\lambda n\ldots)$	$n \mapsto (n-1)^{n\,!=0}$ $r' \mapsto r$		$n\,!=0$	$n==0$	$n * r'$	1	merge executions	
8	$(\lambda n\ldots)$	$r \mapsto (n * r')^{n\,!=0}$ $r \mapsto 1^{n==0}$				(r)		pop $(\lambda n\ldots)$	
9						r			

Fig. 1. Example: Abstract Execution of Program (6)

the abstract execution is forked into two, such that the then- and 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; since the abstract stack remains unchanged during each of these parallel executions, only one column labeled 'T/F' is used for brevity.

Now, as n is bound to only 5 in the environment, the guard $(n\ != 0)$ is resolvable to only true, and we could have chosen to analyze only the then-branch at step 5; however, that would have required invocation of a decision procedure at the branch site to decide on the branch(es) needing analysis given the current environment. Since the environment can have multiple bindings for the same variable, it is likely that a branching condition will resolve to both true and false in which case both branches would have to be analyzed in any case. So, for efficiency we forgo the decision procedure and always analyze both branches in parallel. Note that this does not lead to a loss in precision as all mappings collected during the abstract execution of each of the branches are predicated on their respective guards, thus preserving the conditional information in the nugget. Step 6 under subcolumn labeled 'T', is similar to step 2, except the collected mapping, $fact \mapsto fact^{n\,!=0}$, is now tagged with the current guard, $n\ != 0$.

Pruning recursion The redex $(\lambda n \ldots)\ (n-1)$ at the end of step 6 entails a recursive invocation of the function $(\lambda n \ldots)$, which is already on the stack. The abstract execution has two options at this point: i) follow the recursive invocation, as the concrete execution would, opening the possibility of divergence if later recursive invocations are followed likewise, or ii) *prune*, that is, ignore, the recursive invocation in order to achieve convergence, while (possibly) losing soundness. The first option is obviously infeasible to obtain a convergent analysis, hence we choose the second option. We show later that soundness can be achieved by simply rerunning the abstract execution. 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, and (b) skipping over the abstract execution of the body of $(\lambda n \ldots)$ by collecting the mapping $r' \mapsto r$, simulating the immediate return from the recursive invocation. Now, since the function was *not* in fact recursively invoked, the abstract execution is yet to collect any binding for r, hence, at this point in the abstract execution, r only serves as a *placeholder* for the return value of the recursive call, to be filled in by later analysis. We say return variables r and transitively, r', are *inchoate* as of the end of step 7 since they have no mappings in the environment. Consequently, later invocations of r and r', if any, would be skipped over as well until they are choate (this example, however, has no such invocations). (Note, the mapping $r' \mapsto r$ is not tagged with any guard so as to allow any binding for r, that may appear later, to be transitively bound to r' as well.)

This pruning technique was in fact inspired by the type closure rule for function application in type constraint systems (which is itself isomorphic [6] to 0CFA's [7] handling of function application): $\tau_1 \to \tau_2 <: \tau_1' \to \tau_2'$ implies $\tau_1' <: \tau_1$ and $\tau_2 <: \tau_2'$. The recursive invocation at step 7 can be thought of as generating a type constraint $n \to r <: (n-1) \to r'$ (punning by using program point expressions as type variables) which by the above function type closure rule would give $(n-1) <: n$ and $r <: r'$, which are in turn isomorphic in structure to the mappings collected in step 7, minus the guard. So, this work can be viewed as a method of extending type constraints or 0CFA to incorporate flow- and path-sensitivity while preserving infinite datatype domains. The close alignment of the AOS with the COS imparts flow-sensitivity to our analysis, while the guards on the mappings furnish path-sensitivity.

Merging branches and completing Step 8 merges the completed executions of the two branches by collecting the resulting values tagged with their corresponding guards, that is, adding mappings $r \mapsto (n * r')^{n\,!=0}$ and $r \mapsto 1^{n==0}$, respectively, depicting the flow of each of the tagged resulting values into the outer let-binding $\big(\mathsf{let}\ r = \ (\mathsf{if}\ \ldots)\ \mathsf{in}\ r\big)$. Now r and, by transitivity, r' are no longer inchoate. The redex at the end of step 8 is (r). Step 9 pops the stack, and the abstract execution terminates.

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

$$\begin{array}{l}\{f \mapsto (\lambda fact.\, \lambda n \ldots), fact \mapsto f, fact \mapsto fact^{n\,!=0},\\ \quad n \mapsto 5, n \mapsto (n-1)^{n\,!=0}, r' \mapsto r, r \mapsto (n * r')^{n\,!=0}, r \mapsto 1^{n==0}\}\ ,\end{array} \tag{7}$$

which is, in fact, the nugget for program (6). It is identical to (2) but with the mappings elided there now shown. In general, the nugget is the least fixed-point of the symbolic mappings collectable by the AOS for a given program. A *rerun* of the AOS on the program (6), but this time using the above environment as its initial environment, will yield the same environment at its end *i.e.*, it is a fixed-point. In general, however, the initial run need not result in a fixed-point of the environment.

The need for rerunning The above example does not need to be rerun other than to observe that a fixed-point has been reached. To show the need for rerunning, consider the following variation of program (6) where the return value of the function $(\lambda n \ldots)$ is changed to be a function,

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

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. Hence, when the redex is '$r'()$', the environment of the abstract execution has no known function mapping to r'. So the abstract execution simply skips over the call site '$r'()$' and proceeds without adding any mapping for r'', either. At the merging of the branches the abstract execution adds the mappings $r \mapsto (\lambda y.\, n * r'')^{n\,!=0}$ and $r \mapsto (\lambda x.\, 1)^{n==0}$ to the environment, finally giving mappings to r and r'. Since the AOS is flow-sensitive it must *not* now jump back, out of context, to the skipped-over call-site '$r'()$' and reanalyze it with the now-known bindings for r'—if it were to do so it would lose flow-sensitive information. Although the pruning step was inspired by flow-insensitive type constraint systems, as discussed above, the closure process in a constraint system is flow-*in*-sensitive and can ignore the order of steps; we cannot follow that lead here and must instead align the closure step order with the computation itself. The way we achieve this alignment is by continuing with and finishing the current run, and then *rerunning* the AOS on the same program, but with an initial environment of the one at the end of the just concluded run. This rerun will collect the new bindings for the call $r'()$ in proper execution order, and the environment at the end of the rerun will be

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

This is in fact the least fixed-point of the mappings collected for program (8), that is, the *nugget*; the AOS is run one last time to verify a fixed-point has indeed been reached. As pointed out earlier, the maximum size of the environment is

strongly bound by the number of program subexpressions, and the environment itself is monotonically increasing in size during the course of nuggetizing, thus it must always converge at a fixed-point nugget. The number of reruns required by the nuggetizer depends on the level of nesting of higher-order recursive functions which themselves return functions; we believe it will be small in practice.

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

$$\{n \mapsto 5, n \mapsto (n-1)^{n\,!=0}, r \mapsto 1^{n==0}, r \mapsto (n*r)^{n\,!=0}\} \ . \tag{9}$$

Note the mapping $r' \mapsto r$ is inlined into mapping $r \mapsto (n*r)^{n\,!=0}$ for simplicity. Observe that r in the range of the mapping $r \mapsto (n*r)^{n\,!=0}$ is not guarded—in effect allowing *any* known value of r to be multiplied with *any* known non-zero value of n in order to generate a new value for r. The denotational semantics at n and r of the above nugget is,

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

$\{r \mapsto 1, r \mapsto 2, r \mapsto 6, r \mapsto 24, r \mapsto 120, r \mapsto 5, r \mapsto 8, r \mapsto 18, r \mapsto 48, \ldots\}$

which is sound but not precise at r: r maps to 5 because $n \mapsto 5$ and $r \mapsto 1$ are present, but 5 is not in the range of runtime values assignable to r. The correlation between the argument and return values of recursive function invocations is not captured by the nuggetizer while pruning re-activations of a function, as shown in step 7 of Fig. 1 for $(\lambda n \ldots)$; hence, precision for the analyzed return value is lost. The nuggetizer can, however, be extended to capture the above mentioned correlation and thus perform a precise analysis on the range of return values as well; this extension is presented in [8].

Incompleteness To better show the scope of the analysis, we give an example of an incomplete nugget, the handling of which is beyond the scope of this paper. The following program is inspired by a bidirectional bubble sort.

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

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

$$\left\{x \mapsto 0, x \mapsto (x+1)^{x\,<\,limit}, limit \mapsto 9, limit \mapsto (limit-1)^{x\,<\,limit}\right\}$$

and their corresponding denotational semantics are,

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

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,

then the above nugget could have been used to prove that all accesses to such an array would be in-bounds.

3 Language Model and Concrete Operational Semantics

Our programming language model is an untyped pure higher-order functional language with variables x, integers i, booleans $b \in \{\mathsf{true}, \mathsf{false}\}$, and

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

The grammar assumes expressions are already in an A-normal form [5], so that each program point has an associated program variable. $\langle \eta, \mathrm{E} \rangle$ represents a closure, for a lazy (discussed below) value η, and an environment E. The overbar notation indicates zero or more comma separated repetitions and $\{\cdot\}$ denotes a set, so for example '$\{\overline{x \mapsto \langle \eta, \mathrm{E} \rangle}\}$' is shorthand for the set '$\{x \mapsto \langle \eta, \mathrm{E} \rangle, x \mapsto \langle \eta, \mathrm{E} \rangle, \ldots\}$'; while the subscripted overbar notation denotes a fixed number of repetitions, such that, for example, '$\{\overline{x_k \mapsto \langle \eta_k, \mathrm{E}_k \rangle}\}$' where $k \geq 0$, is shorthand for the set '$\{x_1 \mapsto \langle \eta_1, \mathrm{E}_1 \rangle, x_2 \mapsto \langle \eta_2, \mathrm{E}_2 \rangle, \ldots, x_k \mapsto \langle \eta_k, \mathrm{E}_k \rangle\}$'. Fig. 2 gives the COS for our language. The semantics is *mixed-step*, that is, a combination of both small- and big-step reductions; it allows for an elegant alignment with the AOS. The COS is otherwise standard. 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 environment 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 $\mathrm{E}(F)^{+\lambda} = \langle F, \mathrm{E} \rangle$, respectively. The binary operations $(x \oplus x)$ are evaluated in a maximally lazy fashion; hence the term *lazy values*. 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 absolutely essential to do so for the computation to proceed, that is, the branching condition of the `if` rule needs to be resolved. Again, lazy values allow for a closer alignment between the COS and the AOS. The following function reduces a closure to its smallest equivalent form, or as we say "grounds" it.

Definition 1 (Ground of a Concrete Closure). *The function* $[\![\cdot]\!] : \{\overline{\langle \eta, \mathrm{E} \rangle}\} \to \{\overline{\langle \eta, \mathrm{E} \rangle}\}$ *is inductively defined as,*

1. $[\![\langle x, \mathrm{E} \rangle]\!] = [\![\mathrm{E}(x)]\!]$; $[\![\langle i, \mathrm{E} \rangle]\!] = \langle i, \emptyset \rangle$; $[\![\langle b, \mathrm{E} \rangle]\!] = \langle b, \emptyset \rangle$; $[\![\langle F, \mathrm{E} \rangle]\!] = \langle F, \mathrm{E}' \rangle$, *where* $\mathit{free}(F) \cap \mathit{dom}(\mathrm{E}) = \{\overline{x_k}\}$ *and* $\mathrm{E}' = \{\overline{x_k \mapsto [\![\langle x_k, \mathrm{E} \rangle]\!]}\}$; *and,*
2. $[\![\langle x_1 \oplus x_2, \mathrm{E} \rangle]\!] = \langle \eta, \emptyset \rangle$, *if* $[\![\langle x_1, \mathrm{E} \rangle]\!] = \langle i_1, \emptyset \rangle$, $[\![\langle x_2, \mathrm{E} \rangle]\!] = \langle i_2, \emptyset \rangle$, *and* $i_1 \oplus i_2 = \eta$; *else,* $[\![\langle x_1 \oplus x_2, \mathrm{E} \rangle]\!] = \langle x_1 \oplus x_2, \mathrm{E}' \rangle$, *where* $\mathrm{E}' = \{x_1 \mapsto [\![\langle x_1, \mathrm{E} \rangle]\!], x_2 \mapsto [\![\langle x_2, \mathrm{E} \rangle]\!]\}$.

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

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

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

Fig. 2. Concrete Operational Semantics (COS) Rules

A program p is said to be canonical iff all its local variables are distinct. The canonicality of programs allows new mappings to be simply appended to the environment in the semantics rules, as opposed to overwriting any previous bindings—by keeping local variables distinct we reduce the amount of renaming needed in the COS to zero. The function $\lfloor\cdot\rfloor : \{\overline{p}\} \to \{\overline{x}\}$ returns the variable x serving as a placeholder for the result value of a program p; it is inductively defined as $\lfloor \mathsf{let}\ x = \kappa\ \mathsf{in}\ p\rfloor = \lfloor p\rfloor$, and $\lfloor x\rfloor = x$.

4 Abstract Operational Semantics and Nuggetizer

The additional syntax needed for the AOS is as follows:

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

The abstract environment $\mathcal{E}$ is a set of mappings from variables to abstract closures; it may have multiple mappings for the same variable. Unlike concrete closures, abstract closures $\langle \eta, \mathcal{P}\rangle$ do not come with full environments but with their abstracted forms, the predicates $\mathcal{P}$, which are simple propositional formulae. The predicate $\mathcal{P}$ was informally called a "guard" in Section 2, and notated slightly differently: for example, $n \mapsto (n-1)^{n\,!=0}$ in Section 2 is formally $n \mapsto \langle (n-1), n\ != 0\rangle$. 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 to detect recursive calls for pruning. Fig. 3 presents the AOS rules; observe how the AOS rules structurally align with the COS rules of Fig. 2. The AOS reduction $\longrightarrow$ is defined over configurations which are 4-tuples, $(\mathcal{S}, \mathcal{E}, \mathcal{P}, p)$. The predicate $\mathcal{P}$ in abstract configurations indicates the constraints in force *right now* in the the current function activation. The transitively closed abstract environment λ-lookup function on variables, $\mathcal{E}(x)^{+\lambda}$, is inductively defined to be the smallest set $\{\overline{\langle F_k, \mathcal{P}_k\rangle}\}$, such that $\forall x \mapsto \langle y, \mathcal{P}\rangle \in \mathcal{E}.\, \mathcal{E}(y)^{+\lambda} \subseteq \{\overline{\langle F_k, \mathcal{P}_k\rangle}\}$, and $\forall x \mapsto \langle F, \mathcal{P}\rangle \in \mathcal{E}.\, \langle F, \mathcal{P}\rangle \in \{\overline{\langle F_k, \mathcal{P}_k\rangle}\}$.

The *let* rule collects the let-binding as an abstract closure $x \mapsto \langle \eta, \mathcal{P}\rangle$, analogous to the `let` rule collecting it as a concrete closure. The current predicate

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

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

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

Fig. 3. Abstract Operational Semantics (AOS) Rules

is then updated to reflect the just-executed let-assignment by conjoining the equality condition $(x = \eta)$, which is the new constraint in force, hereafter, in the current function activation. The equality predicates were ignored in Section 2 for simplicity of presentation.

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 then merges their resulting environments and predicates.

The *app* rule performs abstract execution of all possible function invocations at the corresponding call-site in parallel under their respective predicates (recall that $\mathcal{E}$ may map a variable multiply), and then merges their resulting environments and values. Observe various analogies between the app and *app* rules—for example, the app rule pulls the concrete environment E_f from the concrete closure of the corresponding function being invoked, while the *app* rule pulls its abstracted form, that is, the predicate $\mathcal{P}_i$, from the corresponding abstract closure. The function $CALL : \{\overline{\mathcal{S}}\} \times \{\overline{\langle F, \mathcal{P}\rangle}\} \to \{\overline{p}\}$ returns the redex p to be executed when an abstract function closure $\langle F, \mathcal{P}\rangle$ is invoked given an abstract stack $\mathcal{S}$—if it is not a recursive call, the body of F is returned, while if it is a recursive call, it should be pruned, and only the return variable of F is returned, as discussed in Section 2.2. Formally, for $F = \lambda x.\, p$, $CALL\big(\mathcal{S}, \langle F, \mathcal{P}\rangle\big) = p$ if $\langle F, \mathcal{P}\rangle \notin \mathcal{S}$, and $CALL\big(\mathcal{S}, \langle F, \mathcal{P}\rangle\big) = \lfloor p \rfloor$ if $\langle F, \mathcal{P}\rangle \in \mathcal{S}$. If $\mathcal{E}(f)^{+\lambda} = \emptyset$, that is, f is *inchoate* in $\mathcal{E}$, 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 used for analysis in later rerun(s).

We now formally define the nugget and state that it is computable. The formal proofs can be found in [8].

Definition 2 (Nugget). *The nugget of a 3-tuple* $(\mathcal{E}, \mathcal{P}, p)$ *is the the smallest set* $\mathcal{E}'$ *such that* $(\emptyset, \mathcal{E}, \mathcal{P}, p) \longrightarrow^n (\emptyset, \mathcal{E}_n, \mathcal{P}_n, r)$*, for some* n*,* $\mathcal{P}_n$ *and* r*, and either* $\mathcal{E} = \mathcal{E}_n = \mathcal{E}'$*, or inductively,* $\mathcal{E}'$ *is the nugget of 3-tuple* $(\mathcal{E}_n, \mathcal{P}, p)$*.*

The nuggetizer is then defined as the function that builds a nugget starting from an empty environment.

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

As discussed in Section 2.2, the combination of guaranteed termination of the AOS, monotonic growth of the abstract environment during nuggetizing, and existence of a finite upper bound on the abstract environment, implies the abstract environment of the nuggetizer is guaranteed to reach a fixed-point after a finite number of reruns.

Lemma 4 (Computability of the Nugget). *The function nuggetizer* : $\{\overline{p}\} \rightarrow \{\overline{\mathcal{E}}\}$ *is computable.*

In theory, the worst-case runtime complexity of the nuggetizer is $O(n! \cdot n^3)$, where n is the size of a program; we expect it to be significantly less in practice.

4.1 Towards Automated Theorem Proving

In this subsection we provide "glue" which connects the notation of the formal framework above with the syntax of the Isabelle/HOL theorem prover, and then we prove the soundness of the nuggetizer. We relax the grammar for lazy values, atomic computations and programs to be used in this subsection as follows: $\eta ::= x \mid i \mid b \mid F \mid \eta \oplus \eta$, $\kappa ::= \eta \mid \mathsf{if}\ \eta\ \mathsf{then}\ p\ \mathsf{else}\ p \mid \eta\ \eta$, 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 then reduces a lazy value to its smallest equivalent form, or as we say grounds it.

Definition 5 (Ground a Lazy Value). *The function* $[\![\cdot]\!] : \{\overline{\eta}\} \rightarrow \{\overline{\eta}\}$ *is inductively defined as,* $[\![x]\!] = x$*;* $[\![i]\!] = i$*;* $[\![b]\!] = b$*;* $[\![F]\!] = F$*; and,* $[\![\eta_1 \oplus \eta_2]\!] = \eta$*, if* $[\![\eta_1]\!] = i_1$*,* $[\![\eta_2]\!] = i_2$*, and* $i_1 \oplus i_2 = \eta$*; else,* $[\![\eta_1 \oplus \eta_2]\!] = [\![\eta_1]\!] \oplus [\![\eta_2]\!]$*.*

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

Definition 6 (Predicate Satisfaction Relation: $\mathbb{E} \vdash \mathcal{P}$**).** $\mathbb{E} \vdash \mathsf{true}$*;* $\mathbb{E} \vdash \eta_1 = \eta_2$*, iff* $[\![\eta_1[\mathbb{E}]]\!] = [\![\eta_2[\mathbb{E}]]\!]$*;* $\mathbb{E} \vdash \mathcal{P} \wedge \mathcal{P}'$*, iff* $\mathbb{E} \vdash \mathcal{P}$ *and* $\mathbb{E} \vdash \mathcal{P}'$*; and* $\mathbb{E} \vdash \mathcal{P} \vee \mathcal{P}'$*, iff either,* $\mathbb{E} \vdash \mathcal{P}$ *or* $\mathbb{E} \vdash \mathcal{P}'$*.*

Definition 7 (Denotational Semantics of $\mathcal{E}$**:** $[\![\mathcal{E}]\!]$**).** $[\![\mathcal{E}]\!]$ *is smallest set* $\mathbb{E}$ *such that,*

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

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

Definition 8 (Ground of a Concrete Closure). *The function* $[\![\cdot]\!] : \{\overline{\langle \eta, \mathrm{E} \rangle}\} \to \{\overline{\eta}\}$ *is inductively defined as,* $[\![\langle x, \mathrm{E} \rangle]\!] = [\![\mathrm{E}(x)]\!]$; $[\![\langle i, \mathrm{E} \rangle]\!] = i$; $[\![\langle b, \mathrm{E} \rangle]\!] = b$; $[\![\langle F, \mathrm{E} \rangle]\!] = F[\mathbb{E}]$, *where* $free(F) \cap dom(\mathrm{E}) = \{\overline{x_k}\}$ *and* $\mathbb{E} = \{\overline{x_k \mapsto [\![\langle x_k, \mathrm{E} \rangle]\!]}\}$; *and,* $[\![\langle \eta_1 \oplus \eta_2, \mathrm{E} \rangle]\!] = \eta$, *if* $[\![\langle \eta_1, \mathrm{E} \rangle]\!] = i_1$, $[\![\langle \eta_2, \mathrm{E} \rangle]\!] = i_2$, *and* $i_1 \oplus i_2 = \eta$; *else,* $[\![\langle \eta_1 \oplus \eta_2, \mathrm{E} \rangle]\!] = [\![\langle \eta_1, \mathrm{E} \rangle]\!] \oplus [\![\langle \eta_2, \mathrm{E} \rangle]\!]$.

The following theorem then shows that all values arising in variables at runtime will be found in the denotation of the nugget, meaning the latter is a sound reflection of the runtime program behavior.

Theorem 9 (Soundness of the Nuggetizer). *For a closed 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* $[\![\langle \eta, \mathrm{E} \rangle]\!] = \eta'$, *for some* η', *such that* $x \mapsto \eta' \in [\![\mathcal{E}]\!]$.

5 Automated Theorem Proving

In this section we discuss how we use the Isabelle/HOL proof assistant [1] to formalize and prove properties of the nugget. Isabelle/HOL has a rich vocabulary that is well-suited to the encoding of nuggets, and has a number of powerful built-in proof strategies. We translate each 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 properties of programs.

For brevity we elide formal details of the encoding here, but they can be found in [8]. In summary, the encoding of a given nugget $\mathcal{E}$, denoted $[\![\mathcal{E}]\!]_{\mathrm{HOL}}$, is defined inductively as a set of (var, nat) pairs called "abstractenv", where the elements of var are of the form $X(n)$ with $n \in$ nat, representing variables x_n from a given nugget domain. Each mapping $x_i \mapsto \langle \eta, \mathcal{P} \rangle$ in $\mathcal{E}$ defines a separate clause in the inductive definition, where $\mathcal{P}$ defines a set of preconditions. The encoding is straightforward, the main trick being that any variable x_j referenced in η and $\mathcal{P}$ needs to be changed to an Isabelle/HOL variable v_j, and associated with x_j via the precondition $(X(j), v_j) \in$ abstractenv. If η is variable-free, then it is a basic clause, otherwise the clause is inductive.

Our main result for the encoding is that the Isabelle/HOL least fixpoint interpretation of $[\![\mathcal{E}]\!]_{\mathrm{HOL}}$ is provably equivalent to the interpretation of $\mathcal{E}$, i.e. $[\![\mathcal{E}]\!]$. Thus, by Theorem 9, 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 corresponding program p whose nugget is $\mathcal{E}$. For example, consider the nugget $\{x_0 \mapsto 5, x_0 \mapsto (x_0 - 1)^{x_0\,!=0}\}$ for program (1) from Section 2, assuming x_0 in place of n. The encoding of this nugget will generate the following Isabelle/HOL definition:

inductive abstractenv intros
"$(X(0), 5) \in$ abstractenv"
"$((X(0), v_0) \in$ abstractenv $\wedge\, v_0 \neq 0) \Longrightarrow (X(0), v_0 - 1) \in$ abstractenv"

To prove that x_0 falls in the range $[0, 5]$, we state the following theorem in Isabelle/HOL: "$(X(0), v_0) \in \text{abstractenv} \Longrightarrow (v_0 \leq 5 \ \wedge \ v_0 \geq 0)$". 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. In a deployed system we could imagine writing statements such as `assert`$(x_0 \geq 0)$ in the source code of the function, and such asserts would then be compiled to theorems and automatically proved over the automatically generated nugget. This yields a general, powerful, end-to-end programming logic.

6 Related Work

We know of no direct precedent for an automated algorithm that abstracts arbitrary higher-order programs as inductive definitions; however, our work is both related to other verification efforts and to previous techniques in program analysis. We address these two topics in turn.

There is a wide class of research also aimed at partial, more automated verification of program properties than that obtained by full formal verification with a theorem prover. Examples that we would consider more close to our work include systems with dependent and refinement types [9–12]. Our approach has a good combination of expressiveness and automation in comparison to the aforementioned works in that it gives precise, automatic answers to verification questions. Several projects also similarly aim to combine a program analysis with a theorem prover in a single tool, *e.g.* [9, 12, 13]; we believe this general approach has much promise in the future.

This work is an abstract interpretation [4] in the sense that an abstraction of an operational semantics is defined. It differs from abstract interpretation in that we are not interested in abstracting away any of the (infinite) structure of the underlying data domains, and that we wish to derive an inductive structure. The most related abstract interpretation is LFA [13], which addresses a similar problem but by a different technical means. LFA is more a proposal in that it has no formal proofs. Further, it does not generate inductive definitions (like our nuggets) to be fed into a theorem prover at the end of the analysis; rather it relies on invoking a theorem prover on-the-fly to verify first-order logical propositions about the program. We are concerned about the feasibility of implementing LFA in practice, as it fundamentally relies on an initial CPS transformation step which removes the join points of conditional branching statements; hence LFA must explore nearly all paths of the conditional tree in parallel. Our work evolved from attempts to incorporate flow- and path-sensitivity into a type constraint system [3]. Since simple type constraint systems are closely related to 0CFA [7], our work is also a logical descendant of that work.

7 Conclusion

We have defined a static analysis which distills the first-order computational structure from untyped higher-order functional programs, producing a *nugget*. We believe this work has several novel aspects. Most importantly, the analysis produces nuggets which are simple inductive definitions. Inductive definitions provide the best abstraction level for modern theorem-provers—modern provers do their best when reasoning directly over inductively defined structures since that gives a natural induction principle. There are several other features of our approach which make it appealing. The nuggets include guards indicating dependencies. The analysis is fully supportive of higher-order programs—nuggets reflect the higher-order flow of the original program, but expressed as a first-order entity. The *nuggetizer* algorithm which collects a nugget is completely automated and always terminates. The *prune-rerun* technique, a synthesis of existing ideas in type constraint systems and abstract interpretation, provides a new method for soundly interpreting higher-order functions in presence of flow- and path-sensitivity. We show how the meaning of nuggets can be easily formalized in the HOL theorem-prover.

While in this paper we focus on value range analysis for a pure functional language, our general goal is much broader. We have done initial work on extensions to incorporate flow-sensitive mutable state and context-sensitivity.

References

1. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL – A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
2. Kaufmann, M., Moore, J.S.: ACL2. University of Texas at Austin (April 2007)
3. Skalka, C., Smith, S.: History effects and verification. In: ASIAN Symposium on Programming Languages and Systems (APLAS). (2004)
4. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Symposium on Principles of Programming Languages (POPL). (1977)
5. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. In: Conference on Programming Language Design and Implementation (PLDI). (1993)
6. Palsberg, J., Smith, S.: Constrained types and their expressiveness. ACM Transactions on Programming Languages and Systems (TOPLAS) (1996)
7. Shivers, O.G.: Control-flow analysis of higher-order languages. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA (1991)
8. Shroff, P., Skalka, C., Smith, S.F.: The nuggetizer: Abstracting away higher-orderness for program verification. Technical report, Johns Hopkins University (2007) `http://www.cs.jhu.edu/~scott/pll/papers/nuggetizer-TR.pdf`.
9. Chen, C., Xi, H.: Combining programming with theorem proving. In: International Conference on Functional programming (ICFP). (2005)
10. Jones, S.L.P., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for GADTs. In: International Conference on Functional Programming (ICFP). (2006)

11. Xi, H., Pfenning, F.: Eliminating array bound checking through dependent types. In: Conference on Programming language design and implementation (PLDI). (1998)
12. Gronski, J., Knowles, K., Tomb, A., Freund, S.N., Flanagan, C.: Sage: Hybrid checking for flexible specifications. In: Workshop on Scheme and Functional Programming. (2006)
13. Might, M.: Logic-flow analysis of higher-order programs. In: Symposium on the Principles of Programming Languages (POPL). (2007)