The Nuggetizer: Abstracting Away Higher-Orderness for Program Verification

Paritosh Shroff

Department of Computer Science
Johns Hopkins University

Joint work with Christian Skalka (University of Vermont) and Scott F. Smith (Johns Hopkins University)

Objective

Prove non-trivial *inductive* properties about *higher-order* programs

- Statically
- Automatically
- Without any programmer annotations

Exemplar: Value range analysis for higher-order functional programs

- Inferring the range of values assignable to integer variables at runtime

Example: Factorial Program

```
let f = λfact. λn. if (n != 0) then
                        n * fact fact (n - 1)
                      else  1
in f f 5
```

Recursion encoded by "self-passing"

Focus of rest of the talk: Verify range of n is [0, 5]

Motivation

Higher-Order Functional Programming

- Powerful programming paradigm
- Complex from automated verification standpoint
  - Actual low-level operations and the order in which they take place are far removed from the source code, especially in presence of recursion, for example, via the Y-combinator

The simpler first-order view is easiest for automated verification methods to be applied to

Our Approach

- Abstract Away the Higher-Orderness
  - Distill the first-order computational structure from higher-order programs into a *nugget*
  - Preserve much of other behavior, including
    - Control-Flow (Flow-Sensitivity + Path-Sensitivity)
    - Infinite Datatype Domains
    - Other Inductive Program Structures
- Feed the nugget to a theorem prover to prove desirable properties of the source program

A Nugget

- Set of purely first-order inductive definitions
- Denotes the underlying computational structure of the higher-order program
  - Characterizes all value bindings that may arise during corresponding program's execution
- Extracted automatically by the *nuggetizer* from any untyped functional program

Example: Factorial Program

```
let f = λfact. λn. if (n != 0) then
                          n * fact fact (n - 1)
                        else  1
in f f 5
```

Property of interest: Range of n is [0, 5]

Nugget at n: $\{ n \mapsto 5, n \mapsto (n - 1)^{n\, != \,0} \}$

Example: Factorial Program

```
let f = λfact. λn. if (n != 0) then
                        n * fact fact (n - 1)
                      else  1
in f f 5
```

Property of interest: Range of n is [0, 5]

Nugget at n: $\{\ n \mapsto 5,\ n \mapsto (n - 1)^{n\ !=\ 0}\ \}$

Example: Factorial Program

```
let f = λfact. λn. if (n != 0) then
                        n * fact fact (n - 1)
                      else  1
in f f 5
```

Property of interest: Range of n is [0, 5]

Nugget at n: $\{\ n \mapsto 5,\ n \mapsto (n - 1)^{n\ !=\ 0}\ \}$

Guard: A precondition on the usage of the mapping

Denotation of a Nugget

The least set of values implied by the mappings such that their guards hold

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

$$\Downarrow$$

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

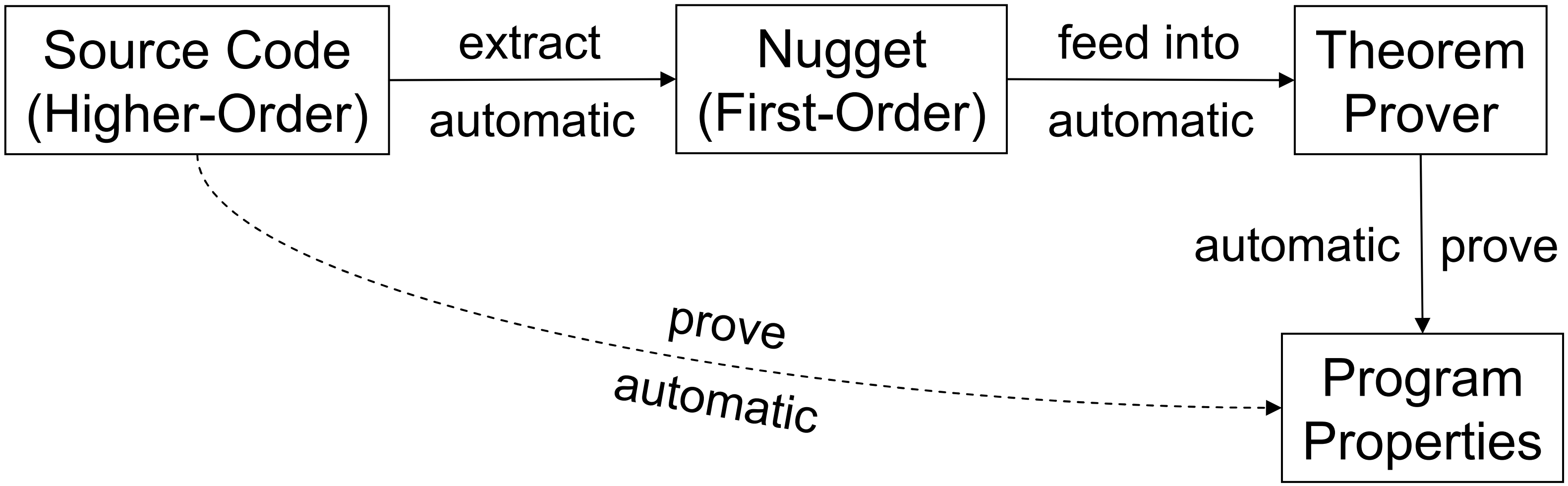
$n \mapsto -1$ is disallowed as $n \mapsto 0$ does not satisfy the guard (n != 0), analogous to the program's computation

Range of n is denoted to be *precisely* [0, 5]

Nuggets in Theorem Provers

- Nuggets are automatically translatable to equivalent definitions in a theorem prover
  - Theorem provers provide built-in mechanisms for writing inductive definitions, and automatically generating proof strategies thereupon
- We provide an automatic translation scheme for Isabelle/HOL
  - We have proved $0 \leq n \leq 5$ and similar properties for other programs

Summary of Our Approach



The Nuggetizer

- Extracts nuggets from higher-order programs via a collecting semantics
  - Incrementally accumulates the nugget over an abstract execution of the program
- = 0CFA + flow-sensitivity + path-sensitivity
  - Abstract execution closely mimics concrete execution
  - Novel *prune-rerun* technique ensures convergence and soundness in presence of flow-sensitivity and recursion

Illustration of the Nuggetizer

```
let f = λfact. λn. let r = if (n != 0) then
                               let fact′ = fact fact in
                               let r′ = fact′ (n - 1) in
                                  n * r′
                             else  1
                        in r
in let f′ = f f in
in let z = f′ 5 in
   z
```

Abstract Call Stack

empty

Abstract Environment

A-normal form – each program point has an associated variable

Illustration of the Nuggetizer

```
let f = λfact. λn. let r = if (n != 0) then
                              let fact′ = fact fact in
                              let r′ = fact′ (n - 1) in
                                n * r′
                            else  1
                       in r
in let f′ = f f in
in let z = f′ 5 in
   z
```

redex

Abstract Call Stack

empty

Abstract Environment

$f \mapsto (\lambda fact.\ \lambda n.\ \ldots)$

Collect the let-binding in the abstract environment

Illustration of the Nuggetizer

```
let f = λfact. λn. let r = if (n != 0) then
                                 let fact′ = fact fact in
                                 let r′ = fact′ (n - 1) in
                                    n * r′
                               else  1
                         in r
in let f′ = f f in
in let z = f′ 5 in
   z
```

redex

Abstract Call Stack

(λfact. λn. …)

Abstract Environment

f ↦ (λfact. λn. …), fact ↦ f

Invoke (λfact. λn. …) on f, and place it in the call stack

Illustration of the Nuggetizer

let f = λfact. λn. let r = if (n != 0) then
let fact′ = fact fact in
let r′ = fact′ (n - 1) in
n * r′
else 1
in r
in let f′ = f f in
in let z = f′ 5 in
z

redex

Abstract Call Stack

empty

Abstract Environment

f ↦ (λfact. λn. …), fact ↦ f, f′ ↦ (λn. …)

Pop (λfact. λn. …), and return (λn. …) to f′

Illustration of the Nuggetizer

```
let f = λfact. λn. let r = if (n != 0) then
                                let fact′ = fact fact in
                                let r′ = fact′ (n - 1) in
                                   n * r′
                              else  1
                        in r
in let f′ = f f in
in let z = f′ 5 in
   z
```

redex

Abstract Call Stack

(λn. …)

Abstract Environment

f ↦ (λfact. λn. …), fact ↦ f, f′ ↦ (λn. …),

n ↦ 5

Invoke (λn. …) on 5, and place it in the call stack

Illustration of the Nuggetizer

let f = λfact. λn. let r = if (n != 0) then
let fact′ = fact fact in
let r′ = fact′ (n - 1) in
n * r′
else 1
in r
in let f′ = f f in
in let z = f′ 5 in
z

redex

Abstract Call Stack

(λn. …)

Abstract Environment

f ↦ (λfact. λn. …), fact ↦ f, f′ ↦ (λn. …),

n ↦ 5

Analyze the *then* and *else* branches in parallel

Illustration of the Nuggetizer

redex

```
let f = λfact. λn. let r = if (n != 0) then
                               let fact′ = fact fact in
                               let r′ = fact′ (n - 1) in
                                 n * r′
                             else  1
                       in r
in let f′ = f f in
in let z = f′ 5 in
   z
```

Abstract Call Stack

(λn. …)

(λfact. λn. …)

Abstract Environment

f $\mapsto$ (λfact. λn. …), fact $\mapsto$ f, f′ $\mapsto$ (λn. …),

fact $\mapsto$ $fact^{n\ !=\ 0}$

n $\mapsto$ 5

Invoke (λfact. λn. …) on fact under the guard n != 0

Illustration of the Nuggetizer

let f = λfact. λn. let r = if (n != 0) then
let fact′ = fact fact in
let r′ = fact′ (n - 1) in
n * r′
else 1
in r
in let f′ = f f in
in let z = f′ 5 in
z

redex

Abstract Call Stack

(λn. …)

Abstract Environment

f ↦ (λfact. λn. …), fact ↦ f, f′ ↦ (λn. …),
fact ↦ $\text{fact}^{n\ !=\ 0}$, fact′ ↦ (λn. …),
n ↦ 5

Pop (λfact. λn. …), and return (λn. …) to fact′

Illustration of the Nuggetizer

```
let f = λfact. λn. let r = if (n != 0) then
                               let fact′ = fact fact in
                               let r′ = fact′ (n - 1) in
                                 n * r′
                             else  1
                         in r
in let f′ = f f in
in let z = f′ 5 in
   z
```

redex

Abstract Call Stack

(λn. …)

Abstract Environment

$f \mapsto (\lambda fact.\ \lambda n.\ \ldots),\ fact \mapsto f,\ f' \mapsto (\lambda n.\ \ldots),$
$fact \mapsto fact^{n\,!=\,0},\ fact' \mapsto (\lambda n.\ \ldots),$
$n \mapsto 5$

Illustration of the Nuggetizer

```
let f = λfact. λn. let r = if (n != 0) then
                                let fact′ = fact fact in
                                let r′ = fact′ (n - 1) in
                                   n * r′
                              else  1
                       in r
in let f′ = f f in
in let z = f′ 5 in
   z
```

redex

Abstract Call Stack

(λn. …)

Abstract Environment

f ↦ (λfact. λn. …), fact ↦ f, f′ ↦ (λn. …),
fact ↦ $\text{fact}^{n\,!=\,0}$, fact′ ↦ (λn. …),
n ↦ 5, n ↦ $(n - 1)^{n\,!=\,0}$,
r′ ↦ r

Prune (ignore) the recursive invocation of (λn. …)

Illustration of the Nuggetizer

let f = λfact. λn. let r = if (n != 0) then
let fact′ = fact fact in
let r′ = fact′ (n - 1) in
n * r′
else 1
in r
in let f′ = f f in
in let z = f′ 5 in
z

redex

Abstract Call Stack

(λn. …)

Abstract Environment

f ↦ (λfact. λn. …), fact ↦ f, f′ ↦ (λn. …),
fact ↦ $fact^{n\ != 0}$, fact′ ↦ (λn. …),
n ↦ 5, n ↦ $(n - 1)^{n\ != 0}$,
r′ ↦ r

r and, transitively, r′ ***have no*** concrete bindings, as of now

r only serves as a placeholder for the return value of the recursive call

Illustration of the Nuggetizer

let f = λfact. λn. let r = if (n != 0) then
let fact′ = fact fact in
let r′ = fact′ (n - 1) in
n * r′
else 1
in r
in let f′ = f f in
in let z = f′ 5 in
z

redex

r and, transitively, r′ ***now have*** concrete bindings

Abstract Call Stack

(λn. …)

Abstract Environment

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

Merge the results of the two branches, tagged with appropriate guards

Illustration of the Nuggetizer

let f = λfact. λn. let r = if (n != 0) then
let fact′ = fact fact in
let r′ = fact′ (n - 1) in
n * r′
else 1
in r

in let f′ = f f in
in let z = f′ 5 in
z

redex

Abstract Call Stack

empty

Abstract Environment

f ↦ (λfact. λn. …), fact ↦ f, f′ ↦ (λn. …),
fact ↦ fact$^{n\ != 0}$, fact′ ↦ (λn. …),
n ↦ 5, n ↦ (n - 1)$^{n\ != 0}$,
r′ ↦ r, r ↦ (n * r′)$^{n\ != 0}$, r ↦ 1$^{n\ == 0}$, z ↦ r

Pop (λn. …), and return r to z

Illustration of the Nuggetizer

```
let f = λfact. λn. let r = if (n != 0) then
                              let fact′ = fact fact in
                              let r′ = fact′ (n - 1) in
                                n * r′
                            else  1
                        in r
in let f′ = f f in
in let z = f′ 5 in
   z
```

Abstract Call Stack

empty

Abstract Environment

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

The abstract execution terminates

Illustration of the Nuggetizer

```
let f = λfact. λn. let r = if (n != 0) then
                               let fact′ = fact fact in
                               let r′ = fact′ (n - 1) in
                                 n * r′
                             else  1
                   in r
in let f′ = f f in
in let z = f′ 5 in
   z
```

Fixed-point of the abstract environment -- observable by *rerunning* abstract execution

Abstract Call Stack

empty

Nugget

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

Nugget: The least fixed-point of the abstract environment

Rerunning Abstract Execution

- Can also contribute new mappings
  - Especially in presence of higher-order recursive functions which themselves return functions

Illustration of Rerunning for Convergence

```
let f = λfact. λn. let r = if (n != 0) then
                           let fact′ = fact fact in
                           let r′ = fact′ (n - 1) in
                           let r″ = r′ () in
                             λx. (n * r″)
                         else λy. 1
                      in r
in let f′ = f f in
in let z = f′ 5 in
in let z′ = z () in
   z′
```

Higher-order recursive function itself returning functions

Abstract Call Stack

empty

Abstract Environment

Illustration of Rerunning for Convergence

let f = λfact. λn. let r = if (n != 0) then
let fact′ = fact fact in
let r′ = fact′ (n - 1) in
let r″ = r′ () in
λx. (n * r″)
else λy. 1
in r
in let f′ = f f in
in let z = f′ 5 in
in let z′ = z () in
z′

redex

Abstract Call Stack

(λn. …)

Abstract Environment

f ↦ (λfact. λn. …), fact ↦ f, f′ ↦ (λn. …),
fact ↦ $\text{fact}^{n\ !=\ 0}$, fact′ ↦ (λn. …),
n ↦ 5, n ↦ $(n - 1)^{n\ !=\ 0}$,
r′ ↦ r

Prune the recursive invocation of (λn. …), as before

Illustration of Rerunning for Convergence

let f = λfact. λn. let r = if (n != 0) then
  let fact′ = fact fact in
  let r′ = fact′ (n - 1) in
  let r″ = r′ () in
    λx. (n * r″)
  else λy. 1

redex

No concrete binding for r′, the analysis simply skips over the redex ‘r′ ()’

in let z′ = z () in
  z′

Abstract Call Stack

(λn. …)

Abstract Environment

f ↦ (λfact. λn. …), fact ↦ f, f′ ↦ (λn. …),
fact ↦ $fact^{n\,!=\,0}$, fact′ ↦ (λn. …),
n ↦ 5, n ↦ $(n - 1)^{n\,!=\,0}$,
r′ ↦ r

Skip over the call-site r′ ()

Illustration of Rerunning for Convergence

let f = λfact. λn. let r = if (n != 0) then
let fact′ = fact fact in
let r′ = fact′ (n - 1) in
let r″ = r′ () in
λx. (n * r″)
else λy. 1

in let z = f′ 5 in
in let z′ = z () in
z′

r′ now has concrete bindings, but no binding for r″

Abstract Call Stack

(λn. …)

Abstract Environment

f ↦ (λfact. λn. …), fact ↦ f, f′ ↦ (λn. …),
fact ↦ $\text{fact}^{n\,!=\,0}$, fact′ ↦ (λn. …),
n ↦ 5, n ↦ $(n - 1)^{n\,!=\,0}$,
r′ ↦ r, r ↦ $(\lambda x.\ n * r'')^{n\,!=\,0}$, r ↦ $(\lambda y.\ 1)^{n\,==\,0}$

Merge the results of the two branches, tagged with appropriate guards

Illustration of Rerunning for Convergence

let f = λfact. λn. let r = if (n != 0) then
let fact′ = fact fact in
let r′ = fact′ (n - 1) in
let r″ = r′ () in
λx. (n * r″)
else λy. 1
in r
in let f′ = f f in
in let z = f′ 5 in
in let z′ = z () in
z′

Abstract Call Stack

empty

Abstract Environment

f $\mapsto$ (λfact. λn. …), fact $\mapsto$ f, f′ $\mapsto$ (λn. …),
fact $\mapsto$ $\text{fact}^{n\ !=\ 0}$, fact′ $\mapsto$ (λn. …),
n $\mapsto$ 5, n $\mapsto$ $(n - 1)^{n\ !=\ 0}$,
r′ $\mapsto$ r, r $\mapsto$ $(\lambda x.\ n * r'')^{n\ !=\ 0}$, r $\mapsto$ $(\lambda y.\ 1)^{n\ ==\ 0}$,
z $\mapsto$ r, x $\mapsto$ (), y $\mapsto$ (), z′ $\mapsto$ $(n * r'')^{n\ !=\ 0}$, z′ $\mapsto$ $1^{n\ ==\ 0}$

End of the initial run

Illustration of Rerunning for Convergence

let f = λfact. λn. let r = if (n != 0) then
    let fact′ = fact fact in
    let r′ = fact′ (n - 1) in
    let r″ = r′ () in
      λx. (n * r″)
  else λy. 1
in r
in let f′ = f f in
in let z = f′ 5 in
in let z′ = z () in
  z′

redex

r′ has concrete bindings

Abstract Call Stack

(λn. …)

Abstract Environment

f ↦ (λfact. λn. …), fact ↦ f, f′ ↦ (λn. …),
fact ↦ $\text{fact}^{n\ !=\ 0}$, fact′ ↦ (λn. …),
n ↦ 5, n ↦ $(n - 1)^{n\ !=\ 0}$,
r′ ↦ r, r ↦ $(\lambda x.\ n * r'')^{n\ !=\ 0}$, r ↦ $(\lambda y.\ 1)^{n\ ==\ 0}$,
z ↦ r, x ↦ (), y ↦ (), z′ ↦ $(n * r'')^{n\ !=\ 0}$, z′ ↦ $1^{n\ ==\ 0}$,
x ↦ $()^{n\ !=\ 0}$, y ↦ $()^{n\ !=\ 0}$, r″ ↦ $(n * r'')^{n\ !=\ 0}$, r″ ↦ $1^{n\ ==\ 0}$

During the rerun

Illustration of Rerunning for Convergence

let f = λfact. λn. let r = if (n != 0) then
    let fact′ = fact fact in
    let r′ = fact′ (n - 1) in
    let r″ = r′ () in
      λx. (n * r″)
  else λy. 1
in r

in let f′ = f f in
in let z = f′ 5 in
in let z′ = z () in
  z′

Now a fixed-point of the abstract environment -- observable by *rerunning* abstract execution

Abstract Call Stack

empty

Nugget

f ↦ (λfact. λn. …), fact ↦ f, f′ ↦ (λn. …),
fact ↦ $\text{fact}^{n\,!=\,0}$, fact′ ↦ (λn. …),
n ↦ 5, n ↦ $(n - 1)^{n\,!=\,0}$,
r′ ↦ r, r ↦ $(\lambda x.\ n * r'')^{n\,!=\,0}$, r ↦ $(\lambda y.\ 1)^{n\,==\,0}$,
z ↦ r, x ↦ (), y ↦ (), z′ ↦ $(n * r'')^{n\,!=\,0}$, z′ ↦ $1^{n\,==\,0}$,
x ↦ $()^{n\,!=\,0}$, y ↦ $()^{n\,!=\,0}$, r″ ↦ $(n * r'')^{n\,!=\,0}$, r″ ↦ $1^{n\,==\,0}$

End of the rerun

However…

Number of reruns required to reach a fixed-point is always (*provably*) finite

- Abstract environment is monotonically increasing across runs
- Size of abstract environment is strongly bound
  - Domain, range and guards of all mappings are fragments of the source program

All feasible mappings will eventually be collected after some finite number of reruns, and a fixed-point reached

Properties of the Nuggetizer

Soundness Nugget denotes all values that may arise in variables at runtime

Termination Nuggetizer computes a nugget for all programs

Runtime Complexity 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

Related Work

- No direct precedent to our work
  - *An automated algorithm for abstracting arbitrary higher-order programs as first-order inductive definitions*
- A logical descendent of 0CFA [Shivers'91]
- Dependent, Refinement Types [Xi+'05, Flanagan+'06]
  - Require programmer annotations
    - Our approach: No programmer annotations
- Logic Flow Analysis [Might'07]
  - Does not generate inductive definitions
  - Invokes theorem prover many times, and on-the-fly
    - Our approach: only once, at the end

Currently working towards

- Completeness
  - *A lossless* translation of higher-order programs to first-order inductive definitions

(The current analysis is sound but not complete)

- Incorporating Flow-Sensitive Mutable State
  - Shape-analysis of heap data structures
- Prototype Implementation

Thank You

Example of Incompleteness

Inspired by bidirectional bubble sort

```
let f = λsort. λx. λlimit. if (x < limit) then
                              sort sort (x + 1) (limit - 1)
                           else  1
in f f 0 9
```

Range of x is [0, 5] and range of limit is [4, 9]

Nugget at x and limit:

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

$$\Downarrow$$

$$\{ x \mapsto 0, \ldots, x \mapsto 9, limit \mapsto 9, \ldots, limit \mapsto 0 \}$$

Correlation between order of assignments to x and limit is lost

External Inputs

let f = λfact. λn. if (n != 0) then
    n * fact fact (n - 1)
  else 1
in if (***inp*** $\geq$ 0) then
  f f ***inp***

Property of interest: Symbolic range of n is [0, …, ***inp***]

Nugget at n: $\{ n \mapsto \boldsymbol{inp}^{\boldsymbol{inp} \geq 0}, n \mapsto (n - 1)^{n \,!= 0} \}$

$$\Downarrow$$

$$\{ n \mapsto \boldsymbol{inp}, n \mapsto \boldsymbol{inp} - 1, \ldots, n \mapsto 0 \}$$

A more complex example

$Z = \lambda f.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))$

```
let f′ = λfact. λn. if (n != 0) then
                          n * fact (n - 1)
                        else  1
in Z f′ 5
```

Nugget at n:

$\{\ n \mapsto 5,\ n \mapsto y,\ y \mapsto (n - 1)^{n\ != 0}\ \} \equiv \{\ n \mapsto 5,\ n \mapsto (n - 1)^{n\ != 0}\ \}$

Another complex example

```
let g = λfact′. λm. fact′ fact′ (m - 1) in
let f = λfact. λn. if (n != 0) then
                        n * g fact n
                      else  1
in f f 5
```

Nugget at n and m: $\{ n \mapsto 5,\ m \mapsto n^{n\,!=\,0},\ n \mapsto (m - 1) \}$

$$\Downarrow$$

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

$$\{ m \mapsto 5,\ m \mapsto 4,\ m \mapsto 3,\ m \mapsto 2,\ m \mapsto 1 \}$$

General, End-to-End Programming Logic

```
let f = λfact. λn. assert (n ≥ 0);
                   if (n != 0) then
                     n * fact fact (n - 1)
                   else  1
in f f 5
```

assert (n ≥ 0) would be compiled down to a theorem, and automatically proved by the theorem prover over the automatically generated nugget

Many asserts are implicit

- Array bounds and null pointer checks

Methodology by Analogy

	Program Model Checking	Our Approach
Abstraction Model	Finite Automaton	First-Order Inductive Definitions (Nugget)
Verification Method	Model Checking	Theorem Proving
Pros	Faster	Higher-Order Programs, Inductive Properties
Cons	First-Order Programs, Non-Inductive Properties	Slower