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 = \text{fact. } \lambda n. \text{if } (n \neq 0) \text{ then } n \times \text{fact fact}(n - 1) \text{ else } 1 \text{ 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

\[
\text{let } f = \lambda \text{fact. } \lambda n. \text{ if } (n \neq 0) \text{ then } n \ast \text{fact fact (n - 1)} \text{ else 1 }
\]

\[
in f f 5
\]

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

Nugget at \(n\): \(\{ n \mapsto 5, n \mapsto (n - 1)^{n \neq 0} \}\)
Example: Factorial Program

\[
\text{let } f = \lambda \text{fact. } \lambda n. \text{ if } (n \neq 0) \text{ then } n \ast \text{fact fact } (n - 1) \text{ else } 1 \text{ in } f \ f \ 5
\]

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

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

\[
\{ n \mapsto 5, n \mapsto (n - 1)^n \neq 0 \} \downarrow
\]

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

\( n \mapsto -1 \) is disallowed as \( n \mapsto 0 \) does not satisfy the guard \( (n \neq 0) \), 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

Source Code (Higher-Order) \[\xrightarrow{\text{extract automatic}}\] Nugget (First-Order) \[\xrightarrow{\text{feed into automatic}}\] Theorem Prover

Program Properties

prove automatic
The Nuggetizer

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

\[
\text{let } f = \lambda \text{fact. } \lambda n. \ \text{let } r = \text{if } (n \neq 0) \text{ then } \\
\quad \text{let } \text{fact}' = \text{fact} \text{ fact in } \\
\quad \text{let } r' = \text{fact'} (n - 1) \text{ in } \\
\quad \quad n * r' \\
\quad \text{else } 1 \\
\quad \text{in } r \\
\text{in } \text{let } f' = f \ f \ \text{in } \\
\quad \text{in } \text{let } z = f' \ 5 \ \text{in } \\
\quad z
\]

Abstract Call Stack:
- empty

Abstract Environment:

A-normal form – each program point has an associated variable
Illustration of the Nuggetizer

let f = \text{fact}. \lambda n. let r = if (n \neq 0) then
    let fact' = \text{fact} \\text{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

Collect the let-binding in the abstract environment
Illustration of the Nuggetizer

let f = \( \lambda \text{fact}. \lambda n. \) let r = if (n != 0) then
let fact’ = \( \lambda \text{fact}. \lambda n. \) 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

Invoke (\( \lambda \text{fact}. \lambda n. \ldots \)) on f, and place it in the call stack

Abstract Call Stack
(\( \lambda \text{fact}. \lambda n. \ldots \))

Abstract Environment
f \mapsto (\( \lambda \text{fact}. \lambda n. \ldots \)), \text{fact} \mapsto f

Invoke (\( \lambda \text{fact}. \lambda n. \ldots \)) on f, and place it in the call stack
Illustration of the Nuggetizer

let f = \(\lambda\) fact. \(\lambda\) n. 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 Environment

Pop (\(\lambda\) fact. \(\lambda\) n. ...), and return (\(\lambda\) n. ...) to f'

Abstract Call Stack

empty
Illustration of the Nuggetizer

let f = \( \lambda \text{fact. } \lambda n. \right) \text{if } (n \neq 0) \text{ then}
\text{let fact}' = \text{fact} \text{ fact in}
\text{let r}' = \text{fact}' (n - 1) \text{ in}
\text{n} \ast \text{r}'
\text{else 1 in r}
in r

in let f' = f f in
in let z = f' 5 in
\text{z}

Invoke (\( \lambda n. \ldots \)) on 5, and place it in the call stack
Illustration of the Nuggetizer

let f = \( \text{fact. } \lambda n. \) let r = if \( (n \neq 0) \) 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
(\( \lambda n. \ldots \))

Abstract Environment
f \mapsto (\lambda \text{fact. } \lambda n. \ldots), \text{fact} \mapsto f, f' \mapsto (\lambda n. \ldots),

n \mapsto 5

Analyze the then and else branches in parallel
Illustration of the Nuggetizer

Let $f = \lambda \text{fact. } \lambda n. \text{ let } r = \text{ if } (n \neq 0) \text{ then }$

let fact' = fact fact in
let $r' = \text{ fact'} (n - 1)$ in
$n * r'$
else 1
in r
in let $f' = f f$ in
in let $z = f' 5$ in
$z$

Abstract Environment

Invoke ($\lambda \text{fact. } \lambda n. \text{ ...}$) on fact under the guard $n \neq 0$
Illustration of the Nuggetizer

let f = \fact. \n. 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

Pop (\fact. \n. ...), and return (\n. ...) to fact’

Abstract Call Stack
(\n. ...)

Abstract Environment

f ↦ (\fact. \n. ...), fact ↦ f, f’ ↦ (\n. ...),
fact ↦ fact^n ! = 0, fact’ ↦ (\n. ...),
n ↦ 5
Illustration of the Nuggetizer

let f = \(\text{fact. } \lambda n. \) let r = if \(n \neq 0\) then
let fact' = fact fact in
let r' = fact' \((n - 1)\) in
\(n \times r'\) else 1
in r

in let f' = f f in
in let z = f' 5 in
z

Abstract Call Stack
\((\lambda n. \ldots)\)

Abstract Environment
\(f \mapsto (\lambda \text{fact. } \lambda n. \ldots), \text{fact} \mapsto f, f' \mapsto (\lambda n. \ldots),\)
\(\text{fact} \mapsto \text{fact}^{n \neq 0}, \text{fact'} \mapsto (\lambda n. \ldots),\)
\(n \mapsto 5\)
Illustration of the Nuggetizer

\[
\text{let } f = \lambda \text{fact. } \lambda n. \text{ let } r = \text{ if } (n \neq 0) \text{ then }
\]
\[
\text{let fact}' = \text{fact fact in }
\]
\[
\text{let } r' = \text{fact}' (n - 1) \text{ in }
\]
\[
n * r' \text{ else } 1
\]
\[
in r
\]
\[
\text{in let } f' = f f \text{ in }
\]
\[
\text{in let } z = f' 5 \text{ in }
\]
\[
z
\]

Prune (ignore) the recursive invocation of (\(\lambda n. \ldots\))
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

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

Abstract Call Stack
(lambdan. ...)

Abstract Environment
f ↦ (λfact. λn. ...), fact ↦ f, f' ↦ (lambdan. ...),
    fact ↦ fact^n != 0, fact' ↦ (lambdan. ...),
    n ↦ 5, n ↦ (n - 1)^n != 0,
r' ↦ r
Illustration of the Nuggetizer

\[
\begin{align*}
&\text{let } f = \lambda \text{fact. } \lambda n. \text{ let } r = \text{ if } (n \neq 0) \text{ then } \\
&\quad \text{let } \text{fact'} = \text{fact} \text{ fact in} \\
&\quad \text{let } r' = \text{fact'} (n - 1) \text{ in} \\
&\quad n \ast r' \\
&\text{else } 1 \\
\text{in } r \\
\text{in let } f' = f \circ f \text{ in} \\
\text{in let } z = f' 5 \text{ in} \\
\text{z}
\end{align*}
\]

\[r \text{ and, transitivity, } r' \text{ now have concrete bindings}\]

\[\text{Merge the results of the two branches, tagged with appropriate guards}\]
Illustration of the Nuggetizer

Let $f = \lambda n. \lambda n. \text{let } r = \text{if } (n \neq 0) \text{ then }$

- Let $\text{fact'} = \text{fact } \text{fact } \text{in}$
- Let $r' = \text{fact'} (n - 1) \text{ in}$
- $n \ast r'$
- else 1

in $r$

Let $f' = f f$ in

Let $z = f' 5$ in

$z$

Pop ($\lambda n. \text{...}$), and return $r$ to $z$

Abstract Call Stack

empty

Abstract Environment

$$
\begin{align*}
  f & \mapsto (\lambda \text{fact. } \lambda n. \text{...}), \quad \text{fact} \mapsto f, \quad f' \mapsto (\lambda n. \text{...}), \\
  \text{fact} & \mapsto \text{fact}^{n \neq 0}, \quad \text{fact'} \mapsto (\lambda n. \text{...}), \\
  n & \mapsto 5, \quad n \mapsto (n - 1)^{n \neq 0}, \\
  r' & \mapsto r, \quad r \mapsto (n \ast r')^{n \neq 0}, \quad r \mapsto 1^{n = 0}, \quad z \mapsto r
\end{align*}
$$
Illustration of the Nuggetizer

let f = \(\lambda\) fact. \(\lambda\) n. let r = if (n \neq 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),\) \(\text{fact} \mapsto f,\) \(\text{f'} \mapsto (\lambda n. \ldots),\)
\(\text{fact} \mapsto \text{fact}^\text{n \neq 0},\) \(\text{fact'} \mapsto (\lambda n. \ldots),\)
\(n \mapsto 5,\) \(n \mapsto (n - 1)^\text{n \neq 0},\)
\(r' \mapsto r,\) \(r \mapsto (n * r')^\text{n \neq 0},\) \(r \mapsto 1^n = 0,\) \(z \mapsto r\)

The abstract execution terminates
Illustration of the Nuggetizer

let f = \lambda fact. \lambda n. let r = if (n \not= 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

Nugget: The least fixed-point of the abstract environment

Abstract Call Stack
empty

Nugget

f \mapsto (\lambda fact. \lambda n. \ldots), fact \mapsto f, f' \mapsto (\lambda n. \ldots),
fact \mapsto fact^n \not= 0, fact' \mapsto (\lambda n. \ldots),
n \mapsto 5, n \mapsto (n - 1)^n \not= 0,
r' \mapsto r, r \mapsto (n * r')^n \not= 0, r \mapsto 1^n = 0, z \mapsto r
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 = \text{fact} \cdot \lambda n. \text{let } r = \text{if } (n \neq 0) \text{ then }
 \text{let } \text{fact}' = \text{fact} \text{ fact} \text{ in }
 \text{let } r' = \text{fact}' (n - 1) \text{ in }
 \text{let } r'' = r' () \text{ in }
 \lambda x. (n * r'')
 \text{else } \lambda y. 1
\text{in } r
\text{in } \text{let } f' = f f \text{ in }
\text{in } \text{let } z = f' 5 \text{ in }
\text{in } \text{let } z' = z () \text{ in } z'

Abstract Call Stack
empty

Abstract Environment
Illustration of Rerunning for Convergence

let f = \( \lambda \text{fact.} \, \lambda n. \) let r = if \( n \neq 0 \) then
  let fact' = fact fact in
  let r' = fact' (n - 1) in
  let r'' = r' () in
  \( \lambda x. (n * r'') \)
else \( \lambda y. 1 \) in r

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

Prune the recursive invocation of (\( \lambda n. \ldots \)), as before

**Abstract Call Stack**
(\( \lambda n. \ldots \))

**Abstract Environment**
f \( \mapsto (\lambda \text{fact.} \, \lambda n. \ldots), \) fact \( \mapsto f, \) f' \( \mapsto (\lambda n. \ldots), \)
fact \( \mapsto \text{fact}^{n \neq 0}, \) fact' \( \mapsto (\lambda n. \ldots), \)
n \( \mapsto 5, \) n \( \mapsto (n - 1)^{n \neq 0}, \)
r' \( \mapsto r \)
Illustration of Rerunning for Convergence

let f = \text{fact}. \lambda n. let r = if (n \neq 0) then
let fact' = fact fact in
let r' = fact' (n - 1) in
let r'' = r' () in
\lambda x. (n * r'')
else \lambda y. 1
in r''
in let f' = f f in

Abstract Call Stack

Abstract Environment

No concrete binding for r', the analysis simply skips over the redex '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

r’ now has concrete bindings, but no binding for r”

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

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

Abstract Call Stack

Abstract Environment

r’ = r, r = (λx. n * r”)n != 0, r = (λy. 1)n == 0

f = (λfact. λn. ...), fact = f, f’ = (λn. ...),
fact = factn != 0, fact’ = (λn. ...),
n = 5, n = (n - 1)n != 0,
Illustration of Rerunning for Convergence

let \( f = \lambda \text{fact. } \lambda n. \) let \( r = \) if \( (n \neq 0) \) then
let \( \text{fact}' = \text{fact} \) fact in
let \( r' = \text{fact}' (n - 1) \) in
let \( r'' = r' () \) in
\( \lambda x. (n \ast r'') \) else \( \lambda y. 1 \)
in \( r \)
in let \( f' = f f \) in
in let \( z = f' 5 \) in
in let \( z' = z () \) in

End of the initial run

Abstract Call Stack
empty

Abstract Environment

\[ f \mapsto (\lambda \text{fact. } \lambda n. \ldots), \ \text{fact} \mapsto f, \ f' \mapsto (\lambda n. \ldots), \]
\[ \text{fact} \mapsto \text{fact}^n \neq 0, \ f' \mapsto (\lambda n. \ldots), \]
\[ n \mapsto 5, \ n \mapsto (n - 1)^n \neq 0, \]
\[ r' \mapsto r, \ r \mapsto (\lambda x. n \ast r'')^n \neq 0, \ r \mapsto (\lambda y. 1)^n = 0, \]
\[ z \mapsto r, \ x \mapsto (), \ y \mapsto (), \ z' \mapsto (n \ast r'')^n \neq 0, \ z' \mapsto 1^n = 0 \]
Illustration of Rerunning for Convergence

let f = λ fact. λ n. let r = if (n != 0) then
    let fact' = fact fact in
    let r' = fact' (n - 1) in
    let r'' = r' () in
    λ x. (n * r'')
else λ y. 1
in r

During the rerun

r' has concrete bindings

Abstract Call Stack

Abstract Environment

29 Nov 2007, APLAS
Abstracting Away Higher-Orderness for Program Verification
Illustration of Rerunning for Convergence

let f = \lambda \text{fact. } \lambda n. \text{ let } r = \text{ if } (n \neq 0) \text{ then }
\text{let fact'} = \text{ fact fact in }
\text{let } r' = \text{ fact'} (n - 1) \text{ in }
\text{let } r'' = r' () \text{ in }
\lambda x. (n * r'')
\text{else } \lambda y. 1
\text{in } r

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

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

Abstract Call Stack
empty

End of the rerun

<table>
<thead>
<tr>
<th>Nugget</th>
</tr>
</thead>
<tbody>
<tr>
<td>f ↦ (\lambda \text{fact. } \lambda n. ...), fact ↦ f, f' ↦ (\lambda n. ...), fact' ↦ fact^n \neq 0, fact'' ↦ (\lambda n. ...), n ↦ 5, n ↦ (n - 1)^n \neq 0, r' ↦ r, r ↦ (\lambda x. n * r'')^n \neq 0, r ↦ (\lambda y. 1)^n = 0, z ↦ r, x ↦ (), y ↦ (), z' ↦ (n * r'')^n \neq 0, z' ↦ 1^n = 0, x ↦ ()^n \neq 0, y ↦ ()^n \neq 0, r'' ↦ (n * r'')^n \neq 0, r'' ↦ 1^n = 0</td>
</tr>
</tbody>
</table>
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 = \( \lambda \text{sort. } \lambda x. \lambda \text{limit. if } (x < \text{limit}) \) then

\[ \text{sort sort (x + 1) (limit - 1)} \]

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:

\( \{ x \mapsto 0, x \mapsto (x + 1) \times \text{limit}, \text{limit} \mapsto 9, \text{limit} \mapsto (\text{limit} - 1) \times \text{limit} \} \)

⇓

\( \{ x \mapsto 0, \ldots, x \mapsto 9, \text{limit} \mapsto 9, \ldots, \text{limit} \mapsto 0 \} \)

Correlation between order of assignments to x and limit is lost
External Inputs

```
let f = \fact. \n. 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, ..., \text{inp}]

Nugget at n: \{ n \mapsto \text{inp} \geq 0, n \mapsto (n - 1)^n \neq 0 \}

\[\downarrow\]

\{ n \mapsto \text{inp}, n \mapsto \text{inp} - 1, ..., 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 \( f' = \lambda \text{fact}. \lambda n. \text{if } (n \neq 0) \text{ then } n \times \text{fact} (n - 1) \text{ else } 1 \)

in \( Z f'\) 5

Nugget at \( n \):
\[
\{ n \mapsto 5, \ n \mapsto y, \ y \mapsto (n - 1)^n \neq 0 \} \equiv \{ \ n \mapsto 5, \ n \mapsto (n - 1)^n \neq 0 \}
\]
Another complex example

\[
\begin{align*}
\text{let } g &= \lambda \text{fact}'. \lambda m. \text{fact'} \text{ fact'} (m - 1) \text{ in} \\
\text{let } f &= \lambda \text{fact}. \lambda n. \text{if } (n \neq 0) \text{ then} \\
&\quad n * g \text{ fact } n \\
&\quad \text{else } 1 \\
\text{in } f \ f \ 5
\end{align*}
\]

Nugget at \text{n} and \text{m}: \{ \text{n} \mapsto 5, \text{m} \mapsto \text{n}^{n \neq 0}, \text{n} \mapsto (m - 1) \} \\
\Downarrow \\
\{ \text{n} \mapsto 5, \text{n} \mapsto 4, \text{n} \mapsto 3, \text{n} \mapsto 2, \text{n} \mapsto 1, \text{n} \mapsto 0 \} \\
\{ \text{m} \mapsto 5, \text{m} \mapsto 4, \text{m} \mapsto 3, \text{m} \mapsto 2, \text{m} \mapsto 1 \}
General, End-to-End Programming Logic

\[
\text{let } f = \lambda \text{fact. } \lambda n. \text{ assert } (n \geq 0); \\
\quad \text{if } (n \neq 0) \text{ then } \\
\quad \quad n \times \text{fact fact } (n - 1) \\
\quad \text{else } 1
\]

\[
in \ f \ f \ 5
\]

\text{assert } (n \geq 0) \text{ 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

<table>
<thead>
<tr>
<th></th>
<th>Program Model Checking</th>
<th>Our Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Abstraction Model</strong></td>
<td>Finite Automaton</td>
<td>First-Order Inductive Definitions (Nugget)</td>
</tr>
<tr>
<td><strong>Verification Method</strong></td>
<td>Model Checking</td>
<td>Theorem Proving</td>
</tr>
<tr>
<td><strong>Pros</strong></td>
<td>Faster</td>
<td>Higher-Order Programs, Inductive Properties</td>
</tr>
<tr>
<td><strong>Cons</strong></td>
<td>First-Order Programs, Non-Inductive Properties</td>
<td>Slower</td>
</tr>
</tbody>
</table>