Program Verification with Flow-Effect Types

Paritosh Shroff
Johns Hopkins University

(joint work with Chris Skalka and Scott Smith)

Refinements in Type Theory

Operational Awareness

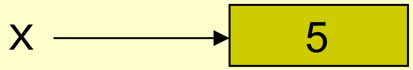
- Simple types *(int, bool, $\tau \rightarrow \tau$)*
- Polymorphic types *('a, 'b, 'a $\rightarrow$ 'b)*
- Subtypes *(τ <: τ)*
- Subtyping constraint types *($t \setminus \{\tau_1 <: \tau_2, \tau_3 <: \tau_4, \ldots\}$)*
- Effect types *($\tau \xrightarrow{\sigma} \tau$)*
- Singleton types *({0}, {true})*
- *Flow-effect types*
  - *incorporate flow-sensitivity*

unordered data-flow
i.e. flow-insensitive

Flow-sensitivity

Temporally Ordered Data-Flow

```
let x = ref 5 in
    x := true;
    !x ∧ false
```



No ML-style value restriction

Flow-sensitivity

Temporally Ordered Data-Flow

```
let x = ref 5 in
      x := true;
      !x ∧ false
```

x → true

No ML-style value restriction

Flow-Effect Types

$$\begin{array}{l} \textbf{let } x_1 \lhd \delta_1 \textbf{ in} \\ \textbf{let } x_2 \lhd \delta_2 \textbf{ in} \\ \ldots \\ \textbf{let } x_n \lhd \delta_n \textbf{ in} \\ x_n \end{array}$$

- Embody all of the computational structure of programs
- *akin to* A-normalized expressions
  - explicitly order atomic computation steps (δ)

$$(1 + 2) + 3 \;\rightarrowtail\; \begin{array}{l} \textbf{let } x_1 \lhd \{1\} + \{2\} \textbf{ in} \\ \textbf{let } x_2 \lhd x_1 + \{3\} \textbf{ in} \\ x_2 \end{array}$$

Flow-Effect "Types"?

$$\begin{array}{l}\textbf{let } x_1 \lhd \delta_1 \textbf{ in} \\ \textbf{let } x_2 \lhd \delta_2 \textbf{ in} \\ \ldots \\ \textbf{let } x_n \lhd \delta_n \textbf{ in} \\ x_n\end{array}$$

- Origins in type theory and shared methodology
- subtyping constraints (data-*flow*) + trace-based *effect*s
- Sequence of data-flows

$$[\delta_1 <: x_1 \;;\; \delta_2 <: x_2 \;;\; \ldots \;;\; \delta_n <: x_n]$$

…as opposed to unordered set

$$\{\delta_1 <: x_1 \,,\; \delta_2 <: x_2 \,,\; \ldots \,,\; \delta_n <: x_n\}$$

$$\lambda x.e \rightarrowtail \lambda x.\begin{array}{l}\textbf{let } x_1 \lhd \delta_1 \textbf{ in} \\ \textbf{let } x_2 \lhd \delta_2 \textbf{ in} \\ \ldots \\ \textbf{let } x_n \lhd \delta_n \textbf{ in} \\ x_n\end{array} \quad \cong \quad x \xrightarrow{[\delta_1 <: x_1 \,;\, \delta_2 <: x_2 \,;\, \ldots \,;\, \delta_n <: x_n]} x_n$$

function body is the *effect*

Flow-Effect Type Closure

- Idealized expression computation
  - abstract interpretation [*Cousot and Cousot*]
    - higher-order
    - trace-based [*Colby and Lee, POPL'96*]
- Naïve Closure: mimics expression computation *i.e. runs* them
  - non-diverging computations $\Rightarrow$ *no problem*
    - `(λid.id(5) + 1; id(true)) (λx.x)`
  - diverging computations $\Rightarrow$ *diverging closures*
    - `(λx.x x) (λx.x x)`

➢ ***Goal***: Find a *sound approximation* for *diverging naïve closures*

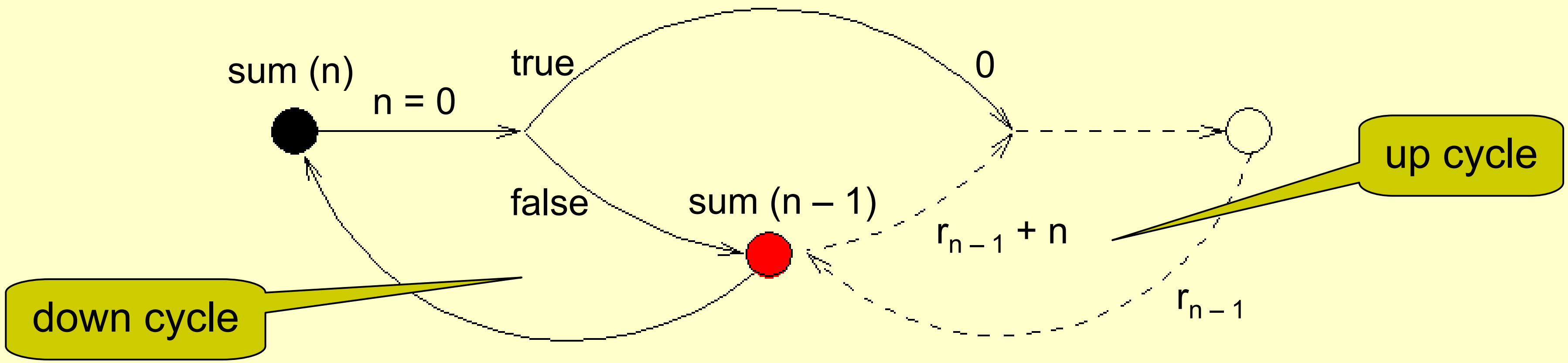
Diverging Computations

- Execute some piece of code *infinitely* often
  - unbounded recursion ⇒ unbounded stack size
  - `(λx.x x) (λx.x x)`

- Crux of a Sound Decidable Closure (Ω-Closure)
  - bounded stack size
    - *fix-point* for recursive computations
  - ***prune-rerun*** technique

Structure of Recursion

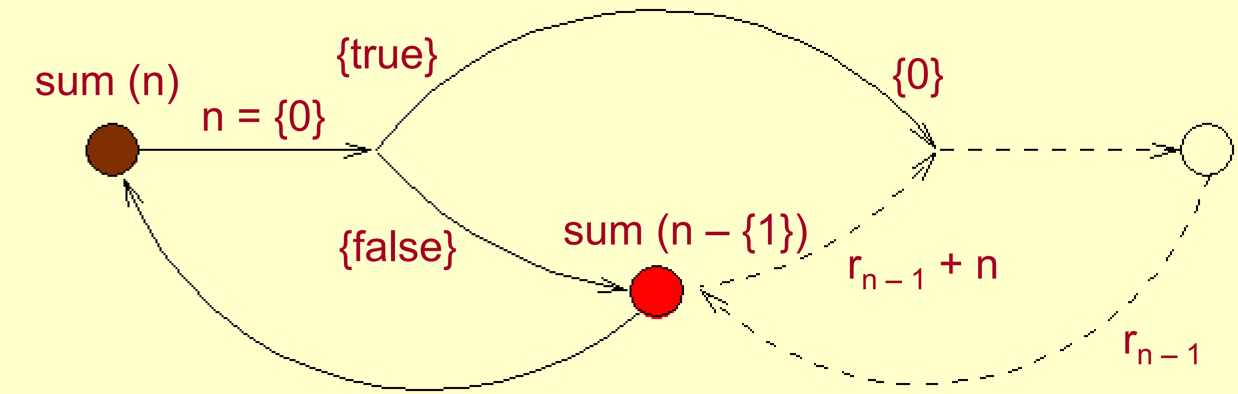
λ_{sum}n. **if** n = 0 **then**
0
else
sum (n – 1) + n

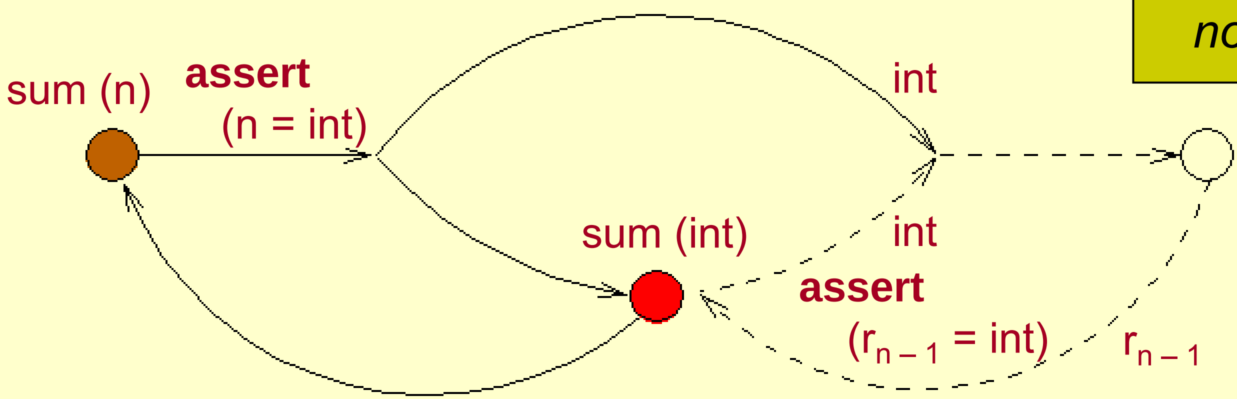


Control-flow graph (CFG) for operational behavior of 'sum'

…will tweak this CFG to bound the stack

Abstract Int/Bool *(…for now)*

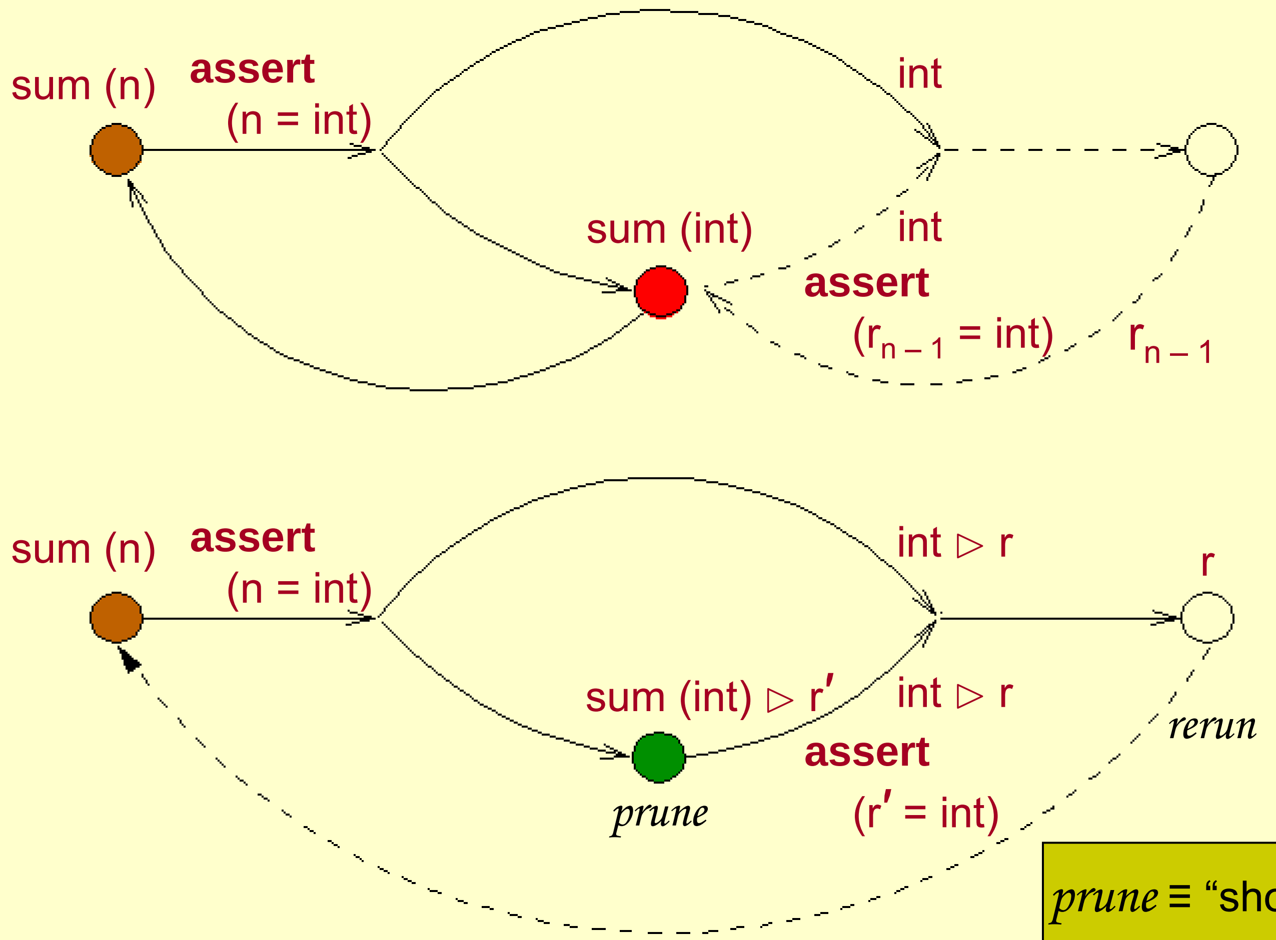




only data is abstracted
not the control-flow

CFG for operational behavior of *abstracted* ‘sum’

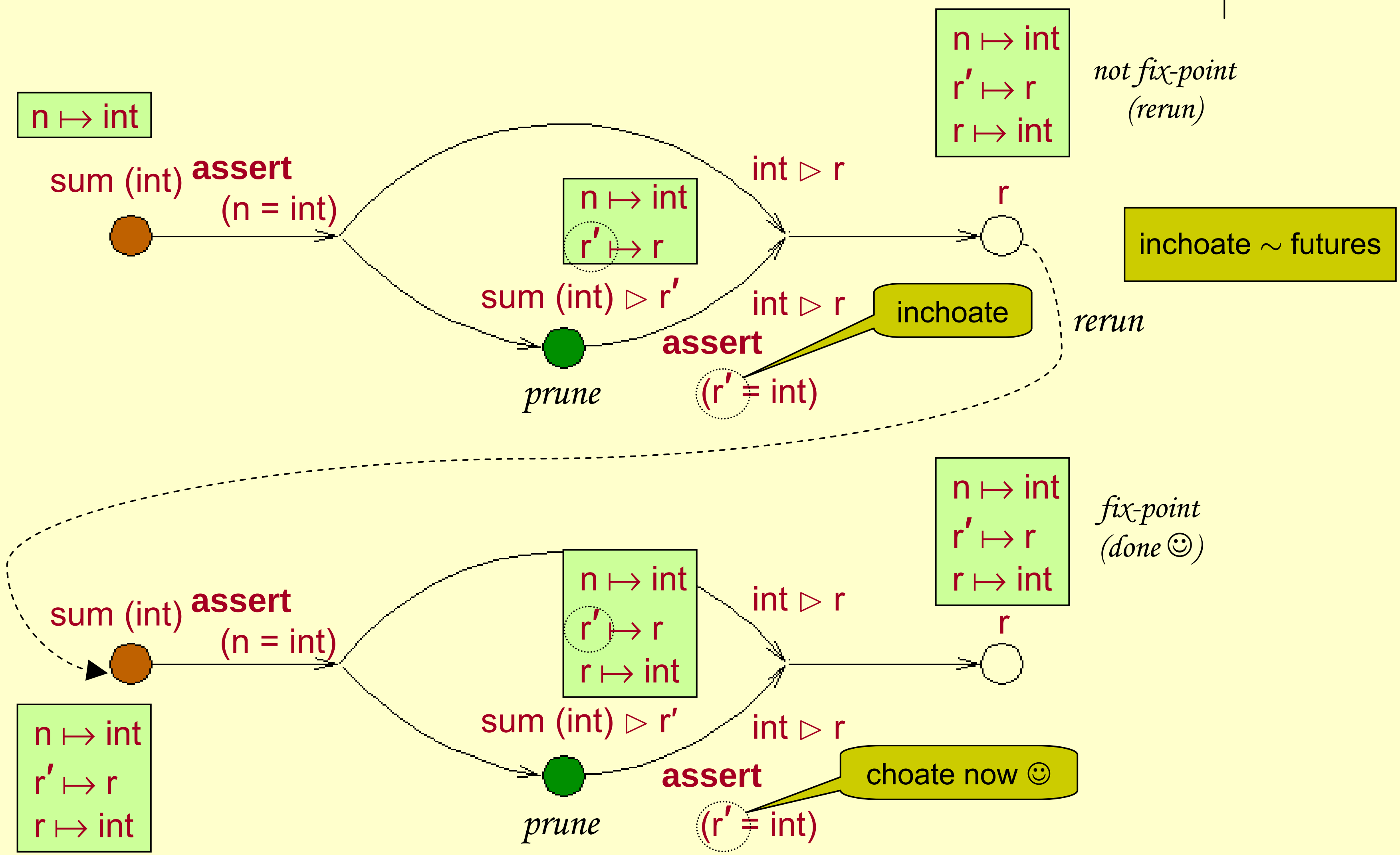
prune-rerun Technique



prune ≡ "short-circuit" function call

CFG for type closure of 'sum' via *prune-rerun* technique

Ω–Closure: sum (int)



Summary of Ω-Closure

- Environment-based
  - monotonic
- Non Recursive Computations
  - simply *run*
- Recursive Computations
  - *prune* recursive calls
  - *rerun* until fix-point (or an *error*) is found

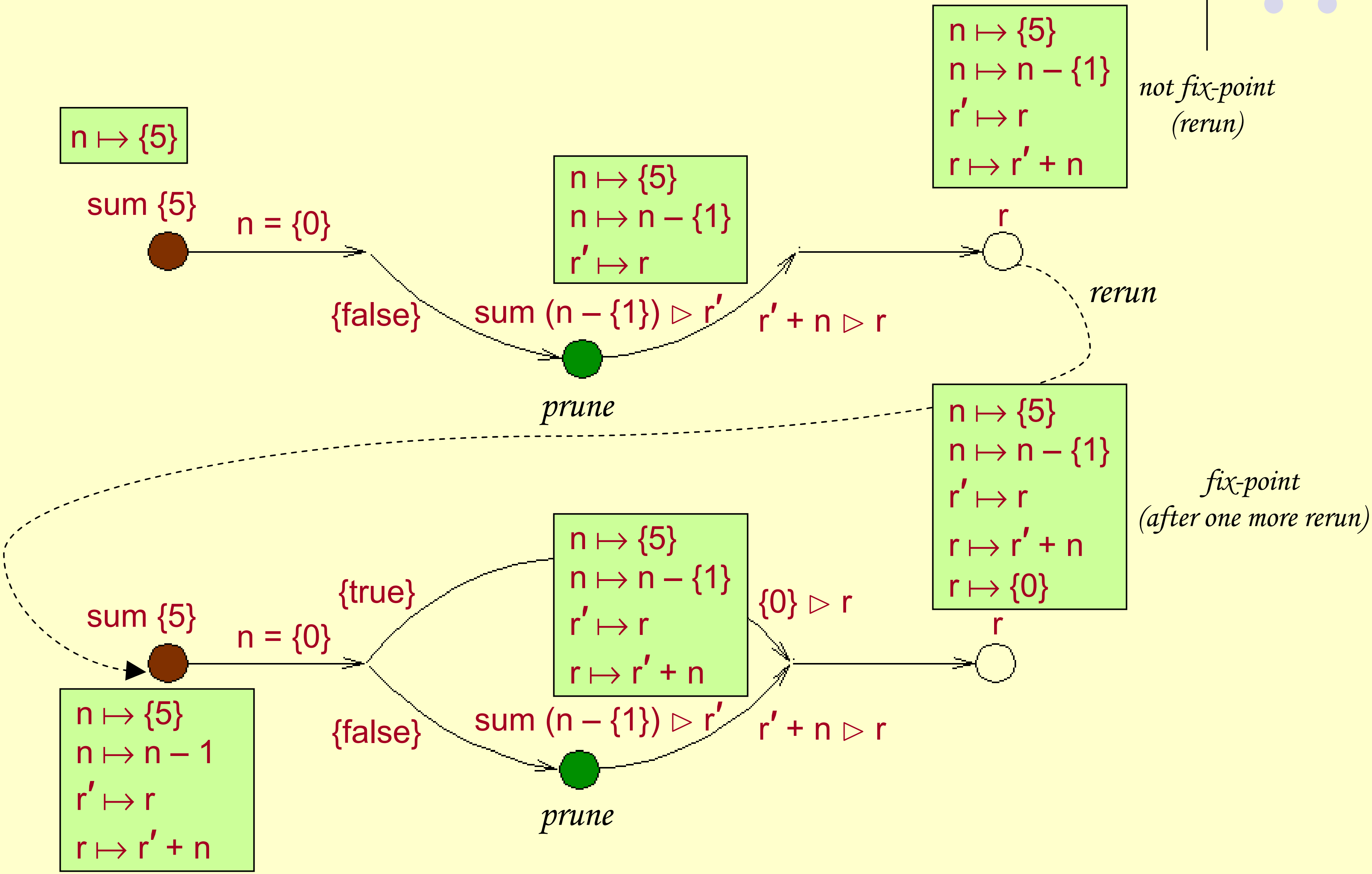
Extensions to Ω-Closure

A. Singleton Types via *Type Operators*
   - {true}, {false}, {1}, {2}, …
   - ∧, ∨, =, +, –, *, /, …
     - {1} + {2}, {true} ∧ {false}

B. Higher-Order Parametric Polymorphism via *Argument Tagging*
   - `(λid.id (5) + 1; id (true) ∧ false) (λx.x)`
   - CPA-style [*Ole Agesen, ECOOP'95*]

C. Mutable State via *Abstract Heap*

Singleton Types via *Type Operators*



Higher-Order Parametric Polymorphism via *Argument Tagging*

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

$$id\ \{5\} + \{1\};\ id\ \{true\} \wedge \{false\}$$

$id \mapsto \lambda x.x$

$$x + \{1\};\ id\ \{true\} \wedge \{false\}$$

$id \mapsto \lambda x.x$
$x \mapsto \{5\}$

$$id\ \{true\} \wedge \{false\}$$

$id \mapsto \lambda x.x$
$x \mapsto \{5\}$

$$x \wedge \{false\}$$

$id \mapsto \lambda x.x$
$\mathbf{x \mapsto \{5\}}$
$\mathbf{x \mapsto \{true\}}$

Type Error

Higher-Order Parametric Polymorphism via *Argument Tagging*

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

$\downarrow$

$id\ \{5\} + \{1\};\ id\ \{true\} \wedge \{false\}$

$id \mapsto \lambda x.x$

$\downarrow$

$\mathbf{x}^{\{5\}} + \{1\};\ id\ \{true\} \wedge \{false\}$

$id \mapsto \lambda x.x$
$\mathbf{x}^{\{5\}} \mapsto \{5\}$

$\downarrow$

$id\ \{true\} \wedge \{false\}$

$id \mapsto \lambda x.x$
$\mathbf{x}^{\{5\}} \mapsto \{5\}$

$\downarrow$

$\mathbf{x}^{\{true\}} \wedge \{false\}$

$id \mapsto \lambda x.x$
$\mathbf{x}^{\{5\}} \mapsto \mathbf{\{5\}}$
$\mathbf{x}^{\{true\}} \mapsto \mathbf{\{true\}}$

$\downarrow$

$\{false\}$

Mutable State via *Abstract Heap*

- Mutable abstract heap
- **ref**, **get**, **set** mimic run-time heap operations
- Recursive data structures
  - collapsed to single abstract heap locations
    - E.g. linked-list
- No need for ML-style value restriction
  - heap operations are flow-sensitive
  - memory-based fixed points
- ➢ Verifying Temporal Heap Properties [Yahav *et al,* ESOP'03]

Properties of Ω-Closure

- **Soundness** If *e* has a type closure then it either diverges or computes to a value.
  - finite automaton simulates the execution of *e*
- **Computability** Type closure is computable for any *e*.
  - bounded stack depth, to number of functions in *e*
  - monotonic environment
  - *no* new type creation $\Rightarrow$ bounded environment

Conjectures about Ω-Closure

- **Completeness** Ω-closure based flow-effect type system is HM-complete.
  - *ordered* subtyping constraint closure
- **Complexity** Ω-closure is computable in exponential time.

Applications

- Model Checking
  - finite automaton of program execution
  - control-flow + data-flow
- Automated Verification of Programs
  - higher-order
    - *akin to* ESP, ARCHER, SLAM for first-order
  - **assert** (x = y): verify program equivalences
- Program Analysis in Compilers

Future Work

- Apply to Java and ML-like languages
  - object-oriented $\Rightarrow$ higher-order features
- Path-sensitivity
  - tag branches before merging
  - split branches based on tag when needed
- Inductive Assertions
  - **assert** $(n \geq 0)$
- Static array bounds check

Download

Ω-Closure based type system has been implemented

www.cs.jhu.edu/~pari/floweffecttypes