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

Temporally Ordered Data-Flow

```ocaml
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
```

No ML-style value restriction
Flow-Effect Types

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

\[
\begin{align*}
(1 + 2) + 3 & \mapsto \\
\text{let } x_1 \triangleleft \delta_1 \text{ in } & \\
\text{let } x_2 \triangleleft \delta_2 \text{ in } & \\
\vdots & \\
\text{let } x_n \triangleleft \delta_n \text{ in } & \\
x_n
\end{align*}
\]
Flow-Effect “Types”?  

- Origins in type theory and shared methodology  
- Subtyping constraints (data-flow) + trace-based effects  
- Sequence of data-flows  

\[
[\delta_1 <: x_1 ; \delta_2 <: x_2 ; \ldots ; \delta_n <: x_n] \quad \text{as opposed to unordered set} \quad \{\delta_1 <: x_1 , \delta_2 <: x_2 , \ldots , \delta_n <: x_n\}
\]

\[
\lambda x.e \mapsto \lambda x. \begin{array}{l}
\text{let } x_1 \triangleleft \delta_1 \text{ in } x_1 \\
\text{let } x_2 \triangleleft \delta_2 \text{ in } x_2 \\
\ldots \\
\text{let } x_n \triangleleft \delta_n \text{ in } x_n
\end{array}
\]

\[
\xrightarrow{\equiv} \quad x \quad \begin{array}{l}
[\delta_1 <: x_1 ; \delta_2 <: x_2 ; \ldots ; \delta_n <: x_n] \\
\rightarrow x_n
\end{array}
\]

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
      - \((\lambda \text{id} . \text{id}(5) + 1; \text{id}(\text{true})) (\lambda x . x)\)
    - diverging computations \(\Rightarrow\) diverging closures
      - \((\lambda x . x x) (\lambda x . x x)\)

- **Goal:** Find a *sound approximation* for diverging naïve closures
Diverging Computations

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

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

\[ \lambda_{\text{sum}} n. \begin{cases} 0 & \text{if } n = 0 \\ \text{else} & \text{sum} (n - 1) + n \end{cases} \]

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

…will tweak this CFG to bound the stack
Abstract Int/Bool (…for now)

CFG for operational behavior of abstracted ‘sum’

only data is abstracted not the control-flow
**prune-rerun** Technique

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

*prune* $\equiv$ “short-circuit” function call
Ω–Closure: sum (int)

[n ↦ int] assert (n = int)

assert (r' = int)

prune

sum (int) ▷ r'

int ▷ r

[r' ↦ r]

[r ↦ int]

[n ↦ int]

r

not fix-point (rerun)

inchoate ~ futures

fix-point (done 😊)

choate now 😊

[r ↦ int]

[r' ↦ r]

[r ↦ int]

n ↦ int

r' ↦ r

n ↦ int

r' ↦ r

r ↦ int

n ↦ int

[r' ↦ r]

[r ↦ int]

[n ↦ int]

[r ↦ int]

[r' ↦ r]

[r ↦ int]
Summary of $\Omega$-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 $\Omega$-Closure

A. Singleton Types via *Type Operators*
   - $\{\text{true}\}$, $\{\text{false}\}$, $\{1\}$, $\{2\}$, …
   - $\land$, $\lor$, $=$, $+$, $-$, $\ast$, $/$, …
     - $\{1\} + \{2\}$, $\text{true} \land \text{false}$

B. Higher-Order Parametric Polymorphism via *Argument Tagging*
   - $(\lambda \text{id}. \text{id} (5) + 1; \text{id} (\text{true}) \land \text{false}) \ (\lambda x.x)$
   - CPA-style [*Ole Agesen, ECOOP’95*]

C. Mutable State via *Abstract Heap*
Singleton Types via

**Type Operators**

**Not fix-point**
(rerun)

fix-point
(after one more rerun)

---

\[
\begin{align*}
\text{sum: } &\{5\} \\
n &\rightarrow \{5\} \\
n &\rightarrow n - \{1\} \\
r' &\rightarrow r \\
\text{prune: } &\{false\} \\
\text{sum: } &\{5\} \\
n &\rightarrow \{5\} \\
n &\rightarrow n - \{1\} \\
r' &\rightarrow r \\
\text{rerun: } &\{false\} \\
r' &\rightarrow r' + n \\
\end{align*}
\]
Higher-Order Parametric Polymorphism via *Argument Tagging*

\[ (\lambda \text{id}. \text{id} \{5\} + \{1\}; \text{id} \{\text{true}\} \land \{\text{false}\}) (\lambda x.x) \]

\[ \text{id} \{5\} + \{1\}; \text{id} \{\text{true}\} \land \{\text{false}\} \]

\[ x + \{1\}; \text{id} \{\text{true}\} \land \{\text{false}\} \]

\[ \text{id} \{\text{true}\} \land \{\text{false}\} \]

\[ x \land \{\text{false}\} \]

*Type Error*
Higher-Order Parametric Polymorphism via *Argument Tagging*

\[(\lambda \text{id}. \text{id} \{5\} + \{1\}; \text{id} \{\text{true}\} \land \{\text{false}\}) (\lambda x. x)\]

\[\downarrow\]

\[\text{id} \{5\} + \{1\}; \text{id} \{\text{true}\} \land \{\text{false}\}\]

\[\downarrow\]

\[x^{\{5\}} + \{1\}; \text{id} \{\text{true}\} \land \{\text{false}\}\]

\[\downarrow\]

\[\text{id} \{\text{true}\} \land \{\text{false}\}\]

\[\downarrow\]

\[x^{\{\text{true}\}} \land \{\text{false}\}\]

\[\downarrow\]

\[\{\text{false}\} \]
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 $\Omega$-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 $\Omega$-Closure

- **Completeness** $\Omega$-closure based flow-effect type system is HM-complete.
  - *ordered* subtyping constraint closure

- **Complexity** $\Omega$-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
Ω-Closure based type system has been implemented

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