Flow-Sensitive Type Recovery in Linear-Log Time

Michael D. Adams*, Andrew W. Keep*, Jan Midtgaard†, Matthew Might‡, Arun Chauhan*, and R. Kent Dybvig*

*Indiana University       †Aarhus University       ‡University of Utah

To appear at SPLASH/OOPSLA’11
Fast type recovery for dynamically typed languages

- Historic goal: to make Lisp compilers competitive (e.g., Steenkiste:91 survey)

- Present relevance:
  - JIT compiler performance
  - Programmer feedback in IDEs
  - Dynamic languages galore: JavaScript, Python, Ruby, Scheme, ...
Examples

... (car x) ...
... (cdr x) ...

(if (pair? x)
   (begin ... (car x) ...)
   (begin ... (cdr x) ...))

(if (... (my-pair? x) ...)
   (begin ... (car x) ...)
   (begin ... (cdr x) ...))
Related work: tagging

Dynamically typed languages *tag* values with their type, in order to test it at run time.

Steenkiste:Topics91 surveyed tagging techniques. He classified tagging operations into four primitives:

- *tag insertion*,

- *tag extraction*,

- *tag removal*, and

- *tag checking*.

Steenkiste’s conclusion: the latter is the most expensive (adding 11–24% to run time)
Related work: tagging optimization

Henglein (LFP’92): an $O(n^{\alpha(n)})$ tagging optimization algorithm.

Goal: eliminate *tag insertion* and *tag removal*

Means: unification

Performance:

- eliminates $\sim 40\%$ of dynamic tag insertions
- and $\sim 55\%$ of dynamic tag removals

Closely related, albeit a slightly different problem
Context and contribution

- Type recovery for Scheme
- Optimization angle: eliminate redundant *tag checks*
Context and contribution

- Type recovery for Scheme
- Optimization angle: eliminate redundant *tag checks*
- Result: An $O(n \log n)$ time flow analysis algorithm.
Context and contribution

- Type recovery for Scheme
- Optimization angle: eliminate redundant *tag checks*
- Result: An $O(n \log n)$ time flow analysis algorithm.
- Chez Scheme implementation:
  - Optimization enabled by default
  - Large input programs
  - Computer generated code (e.g., Scheme macros)
Outline

- From data-flow analysis to 0CFA
- From 0CFA to sub-0CFA
- From flow-insensitive to flow-sensitive CFA
- Efficient flow-sensitive CFA
- Implementation and benchmarks
Data flow analysis and complexity

Work-list based data-flow analysis algorithm:

\[ W := \{ e | e \in Prog \} \]

\[ \text{while } W \neq \emptyset \]

\[ e := \text{dequeue}(W) \]

\[ \text{compute } e.\text{out}(e.\text{in}) \]

\[ \text{if } e.\text{out} \text{ changed} \]

\[ \text{for outgoing edges } e \rightarrow e' \]

\[ \text{compute } e'.\text{in}(e.\text{out}) \]

\[ \text{enqueue}(e') \]

Time complexity: \( O(|L|(N + E)) \)

(under the usual finite lattice and monotonicity assumptions)
0CFA can also be cast as a work-list based algorithm:

\[ W := \{ e \mid e \in \text{Prog} \} \]

\[ \text{while } W \neq \emptyset \]

\[ e := \text{dequeue}(W) \]

\[ \text{compute } e.\text{out}(e.\text{in}) \]

\[ \text{if } e.\text{out} \text{ changed} \]

\[ \text{for outgoing edges } e \rightarrow e' \]

\[ \text{compute } e'.\text{in}(e.\text{out}) \]

\[ \text{enqueue}(e') \]

with dynamic addition of call and return edges

and where \( \hat{r} \in \{\bot, \top\} \) and \( \hat{v} \in \{\lambda x_1.e_1, \lambda x_2.e_2, \ldots\} \).

Time complexity: \( O(|L|(N + E)) = O(n(n + n^2)) = O(n^3) \)
Sub-0CFA and complexity

Sub-0CFA is a faster and more approximate CFA variant (Ashley-Dybvig:98).

It works over a flat lattice of singleton lambdas:

\[
\lambda x_1.e_1 \quad \lambda x_2.e_2 \quad \lambda x_3.e_3 \quad \cdots \quad \lambda x_n.e_n
\]

where \( \top \) denotes “any lambda”

Time complexity: \( O(|L|(N + E)) = O(3(n + n)) = O(n) \)
Technical question: how do you apply $\top$?

Answer: (modularity trick due to Shivers)

track keep of 'escaping lambdas'.

$\top$ and escape trigger each other:

- If $e_0$ of a call site $e_0 \ e_1$ is $\top$, then operand $e_1$ escapes
- If a function $\lambda x. e$ escapes, then $x$ is $\top$
Flow-Sensitive vs. Flow-Insensitive Analysis

\[
\lambda x \\
\text{car } x \\
\text{cdr } x
\]
Standard 0CFA is **flow-insensitive**: Different occurrences of the same variable are not distinguished.
Flow-Sensitive vs. Flow-Insensitive Analysis

\[
\lambda x \\
\quad x: T \\
\text{car } x \\
\quad x: PAIR \\
\text{cdr } x \\
\quad x: PAIR
\]

Hence we need a **flow-sensitive** CFA to distinguish different occurrences of the same variable

(Mogensen:HOSEC00)
Flow-sensitive sub-0CFA as a work-list based algorithm:

\[
W := \{ e \mid e \in Prog \}
\]

while \( W \neq \emptyset \)

\[
e := \text{dequeue}(W)
\]

compute \( e.\text{out}(e.\text{in}) \)

if \( e.\text{out} \) changed

\[
\text{for outgoing edges } e \to e' \\
\text{compute } e'.\text{in}(e.\text{out}) \\
\text{enqueue}(e')
\]

where \( \hat{r} \in \{ \bot, \top \} \), \( \hat{\rho} \in \text{Var} \to \hat{\text{Val}} \) and

\( \hat{v} \in \hat{\text{Val}} = \{ \text{TRUE}, \text{FALSE}, \text{PAIR}, \ldots, \lambda x_1.e_1, \lambda x_2.e_2, \ldots \} \)

Time complexity: \( O(|L|(N + E)) = O(n(n + n)) = O(n^2) \)
Idea: move information more efficiently between occurrences of the same variable. This is the job of the skipping function.

Given an expression \( e \) in context \( C \), the information about \( x \)'s type after \( C \) is a function of the information about \( x \)'s type after \( e \) and \( x \)'s type before \( C \) (assuming \( C \) does not mention \( x \)).
Assuming $e_1$ and $e_2$ do not mention $x$:

$$ (\text{if (pair? x) e}_1 e_2) $$

If, e.g, $e_1$ can only give true and $e_2$ can only give false
Assuming $e_1$ and $e_2$ do not mention $x$:

$\text{(if (pair? x) } e_1 e_2)\text{)}$

If, e.g., $e_1$ can only give false and $e_2$ can only give true
Skipping function, examples

\((\lambda\ x.e)\)
Skipping functions in general

Assuming C does not mention x:

In general: a combination of knowledge about x that depends on the flow into C and e
With only a (small) finite number of combinations a context skipping function has a canonical form $\mathcal{V}_{C,e}$

Combining context skips:

$$\mathcal{V}_{C_2C_1,e} = \mathcal{V}_{C_2,C_1[e]} \circ \mathcal{V}_{C_1,e}$$

Idea: cache context skipping functions

A cache entry $\mathcal{V}_{C,e}$ is shared for all variables not mentioned in $C$.

Once more flow information is learned about $C$ or $e$ we update the cache.
With the cache we can quickly move information across $C$ in $O(1)$ time

Intuitively, there are $n$ choices for $C$ and $e$, hence the cache size would be $O(n^2)$.

Idea: cache less, but enough information:

- $O(n \log n)$ entries
- any entry can be recomputed in $O(\log n)$ time
- entry updates only require $O(\log n)$ time
Cache, diagrammatically

\[\begin{align*}
\mathcal{V}_{C_1 \ldots C_4, e_5} & \quad \mathcal{V}_{C_1, e_1} \quad \text{if}_0 \\
\mathcal{V}_{C_1 C_2, e_3} & \quad \mathcal{V}_{C_2, e_2} \quad \text{if}_1 \quad e_1^t \quad e_1^f \\
\mathcal{V}_{C_1 \ldots C_4, e_5} & \quad \mathcal{V}_{C_3, e_3} \quad \text{if}_2 \quad e_2^t \quad e_2^f \\
\mathcal{V}_{C_3 C_4, e_5} & \quad \mathcal{V}_{C_4, e_4} \quad \text{if}_3 \quad e_3^t \quad e_3^f \\
\mathcal{V}_{C_1 \ldots C_8, e_9} & \quad \mathcal{V}_{C_5, e_5} \quad \text{if}_4 \quad e_4^t \quad e_4^f \\
\mathcal{V}_{C_5 \ldots C_8, e_9} & \quad \mathcal{V}_{C_6, e_6} \quad \text{if}_5 \quad e_5^t \quad e_5^f \\
\mathcal{V}_{C_5 \ldots C_8, e_9} & \quad \mathcal{V}_{C_7, e_7} \quad \text{if}_6 \quad e_6^t \quad e_6^f \\
\mathcal{V}_{C_7 C_8, e_9} & \quad \mathcal{V}_{C_8, e_8} \quad \text{if}_7 \quad e_7^t \quad e_7^f \\
\mathcal{V}_{C_7 \ldots C_8, e_9} & \quad \mathcal{V}_{C_8, e_8} \quad \text{if}_8 \quad e_8^t \quad e_8^f
\end{align*}\]
What to skip?

The cache tells us *how* to skip a context.

Now, *what* should we skip?
What to skip?

The cache tells us how to skip a context.

Now, what should we skip?

A: skip the longest context not containing the variable in question

This can be answered with a lowest common ancestor algorithm (Aho-al:73, Alstrup-al:04)

It takes linear time to build, and then answers queries in $O(1)$ time.
Flow-sensitive sub-0CFA, revised complexity

$W := \{ e \mid e \in Prog \}$

while $W \neq \emptyset$

    e := dequeue($W$)
    compute $e$.out($e$.in)
    if $e$.out changed
        update skipping function cache
    for outgoing edges $e \rightarrow e'$ //with and w/o skipping
        compute $e'$.in($e$.out)
        enqueue($e'$)

Time complexity:

$O(|L|(N + E)) = O(1(n \log n + n \log n)) = O(n \log n)$
Implemented for full Scheme as experimental optimization in Chez Scheme to remove run-time type checks

For a standard set of benchmark it removes

- $\sim 69\%$ of type tests in code, corresponding to
- $\sim 55\%$ dynamic (run-time) type checks

The analysis soundly models Scheme’s undefined evaluation order.

By fixing the evaluation order it goes up to $\sim 60\%$. 
Empirical complexity analysis

Source node count versus analysis time
Flow sensitivity of sub-0CFA vs. 0CFA

Percentage of dynamic type checks removed
Selected Benchmarks

Percent of type checks removed

- mbooz
- mperm
- nboyer
- normalization
- nqueens
- noku
- nucleic
- parafins
- parsing
- peval
- pi
- pnpoly
- primes
- puzzle
- quicksort
- ray
- readd

Legend:
- Sub-0CFA flow-insensitive
- 0CFA flow-insensitive
- Sub-0CFA flow-sensitive
- 0CFA flow-sensitive
Selected Benchmarks

Percent of type checks removed
Future work

- Must-analysis
- Range analysis
- Forwards/Backwards Abstract Interpretation
- ...
A novel type-recovery algorithm which is both fast ($O(n \log n)$) and effective (eliminating $\sim 60\%$ of dynamic type checks) Suitable for just-in-time compilers etc.