Flow-Sensitive Type Recovery in Linear-Log Time

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

Abstract

The flexibility of dynamically typed languages such as JavaScript, Python, Ruby, and Scheme comes at the cost of run-time type checks. Some of these checks can be eliminated via control-flow analysis. Traditional control-flow analysis (CFA) is not ideal for this task, however, as it ignores flow-sensitive information that can be gained from dynamic type predicates, such as JavaScript's `instanceof` or Scheme's `pair?`, and from type-restricted operators, such as Scheme's `car`. Yet, adding flow-sensitivity to a traditional CFA worsens the already significant compile-time cost of traditional CFA. This makes it unsuitable for use in just-in-time compilers.

In response, we have developed a fast, flow-sensitive type-recovery algorithm based on the linear-time, flow-insensitive sub-0CFA. The algorithm has been incorporated into the commercial Chez Scheme compiler, where it has proven to be effective, justifying the elimination of about 60% of run-time type checks in a large set of benchmarks. In practice, the algorithm processes on average over 75,000 lines of code per second and scales well asymptotically, running in only $O(n \log n)$ time. We achieve this compile-time performance and scalability through a novel combination of data structures and algorithms.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers, Optimization

General Terms Languages

Keywords Control-flow analysis, flow sensitivity, path sensitivity, type recovery

1. Introduction

Dynamically typed languages such as JavaScript, Python, Ruby, and Scheme are flexible, but this flexibility comes at the cost of run-time type checks. This cost can be reduced via a type-recovery analysis (Shivers 1990), which attempts to discover variable and expression types at compile time and thereby justify the elimination of run-time type checks.

Since these languages are higher-order languages in which the call graph is not static, the type-recovery analysis generally must be a form of control-flow analysis (Shivers 1988). A control-flow analysis (CFA) tracks the flow of function values to call sites and builds the call graph even as it tracks the flow of other values to their use sites.

To maximize the number of checks removed, the analysis must take evaluation order into account. That is, it must be *flow sensitive* (Banning 1979). In the following expression, even a flow-insensitive control-flow analysis can determine that `x` is a pair and thus `car` need not check that `x` is a pair.

```
(let ([x (cons e1 e2)]) (car x))
```

To make a similar determination in the following expressions, however, evaluation order must be taken into account.

```
(let ([x (read)]) (begin (cdr x) (car x)))

(let ([x (read)]) (if (pair? x) (car x) #f))
```

A flow-insensitive analysis treats all references the same, but a flow-sensitive analysis can determine that `(car x)` is reached only after surviving the implicit pair check of `(cdr x)` in the first expression and only when the explicit pair check succeeds in the second expression. It can thus eliminate the implicit pair check from `car` in both expressions.[1]

In this paper, we present a flow-sensitive, CFA-based type-recovery algorithm that runs in linear-log time. Because we are interested in using the analysis to justify type recovery in a fast production compiler (Dybvig 2010), we have chosen to base the analysis on sub-0CFA (Ashley and Dybvig 1998), a linear-time, flow-insensitive variant of 0CFA. We use a novel combination of data structures and algorithms to extend sub-0CFA with flow sensitivity at the cost of only an additional logarithmic factor.

The algorithm has been incorporated into the commercial Chez Scheme compiler, where it has proven to be effective,

* Indiana University, †Aarhus University, ‡University of Utah. This research was facilitated in part by a National Physical Science Consortium Fellowship and by stipend support from the National Security Agency.

[1] Our use of the term *flow sensitive* agrees with the original definition of Banning (1979), in which an analysis of two expressions take their order of evaluation into account, as well as with a glossary definition (Mogensen 2000), in which separate analysis results are computed for distinct program points. Our analysis might also be considered path sensitive, depending on the definition of path sensitivity used.

justifying the elimination of about 60% of run-time type checks in a large set of benchmarks. The algorithm has also proven to be fast, processing over 75,000 lines of code per second on average. Furthermore, since it runs in $O(n \log n)$ time, it scales well.

The remainder of this paper reviews the semantics and implementation of 0CFA and sub-0CFA (section 2), describes the traditional technique for implementing flow sensitivity (section 3), describes our more efficient technique for implementing flow sensitivity (section 4), discusses practical considerations in a real-world implementation and presents benchmark results (section 5), reviews related work (section 6), and finally concludes (section 7).

2. Background

In this section, we review two relevant forms of control-flow analysis, Shiver's 0CFA and Ashley's sub-0CFA. We also discuss their implementations in terms of flow graphs, how top and escaped values are handled, and our representation of non-function types. Readers familiar with control-flow analysis might wish to skip forward to section 3.

2.1 0CFA

Constraint rules for 0CFA on the call-by-value λ-calculus are presented in figure 1. The operational semantics of this language is standard and is omitted.

The analysis stores a reachability flag, $[\![e]\!]_{in}$, for each expression e. The flag is $\top$ if the expression is reachable and $\bot$ otherwise, and $\sqsupseteq$ is the standard partial order over $Bool$ where $\top \sqsupseteq \bot$.

In addition, for each subexpression e, there is a flow variable $[\![e]\!]_{out}$ storing the abstract value that the expression returns, i.e., a subset of the lambda terms that may flow there. For example, the LAMBDA rule says that if $\lambda x.e$ is reachable, then the result of that expression includes an abstract value representing the lambda.

The CALL_{mid} and CALL_{fun} rules use $\mathbb{K}(e)$ which returns the source context[2] of e. For example, the CALL_{fun} rule says that if a lambda flows to a subexpression that is contextually located inside an application (i.e., $\mathbb{K}(e_1) = (e_0\ \square)$), and a value flows to the operand e_1 of the expression, then the body of the invoked lambda is reachable and the actual argument flows to the formal parameter.

To find a solution to these constraints, the analysis initially assigns $\bot$ to each $[\![e]\!]_{in}$ and the empty set to each $[\![e]\!]_{out}$. Then it iteratively uses the constraint rules to update $[\![e]\!]_{in}$ and $[\![e]\!]_{out}$ until they converge to a solution. In the process $[\![e]\!]_{in}$ and $[\![e]\!]_{out}$ monotonically climb the $Bool$ and $\widehat{Val}$ lattices respectively (Nielson et al. 1999).

[2] As the above analysis is extended and optimized throughout the rest of this paper we will need an explicit representation of context to reason about. Hence for presentational purposes the contexts of expressions in the above 0CFA differ from more traditional presentations (Nielson et al. 1999).

$$
\begin{array}{rl}
\text{Expressions:} & e \in Exp = x \mid \lambda x.e \mid e\ e \\
\text{Contexts:} & \mathbb{K}(e) = \square \mid (\square\ e_1) \mid (e_0\ \square) \mid (\lambda x.\square) \\
\text{Signatures:} & [\![e]\!]_{in} \in Bool \quad \{\text{- Whether } e \text{ is reachable -}\} \\
& [\![e]\!]_{out} \in \widehat{Val} \quad \{\text{- What } e \text{ evaluates to -}\} \\
& \hat{r} \in Bool = \{\bot, \top\} \qquad \hat{v} \in \widehat{Val} = \wp(\widehat{Lam}) \\
& \widehat{Lam} = \{\lambda x_1.e_1, \lambda x_2.e_2, \lambda x_3.e_3, \ldots\}
\end{array}
$$

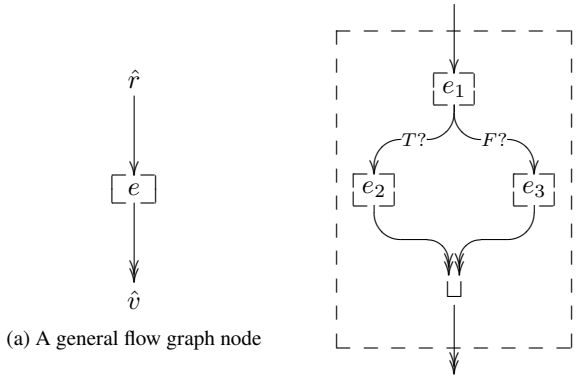
$$\frac{[\![\lambda x.e]\!]_{in} \sqsupseteq \top}{[\![\lambda x.e]\!]_{out} \supseteq \{\lambda x.e\}}\ \text{LAMBDA} \qquad \frac{[\![e_0\ e_1]\!]_{in} \sqsupseteq \hat{r}}{[\![e_0]\!]_{in} \sqsupseteq \hat{r}}\ \text{CALL}_{in}$$

$$\frac{[\![e_0]\!]_{out} \supseteq \{\hat{v}_0\} \qquad \mathbb{K}(e_0) = (\square\ e_1)}{[\![e_1]\!]_{in} \sqsupseteq \top}\ \text{CALL}_{mid}$$

$$\frac{\begin{array}{c}[\![e_0]\!]_{out} \supseteq \{\lambda x.e_\lambda\} \\ [\![e_1]\!]_{out} \supseteq \{\hat{v}_1\} \qquad \mathbb{K}(e_1) = (e_0\ \square)\end{array}}{\begin{array}{cc}[\![e]\!]_{in} \sqsupseteq \top & [\![x]\!]_{out} \supseteq \{\hat{v}_1\}\end{array}}\ \text{CALL}_{fun}$$

$$\frac{\begin{array}{c}[\![e_0]\!]_{out} \supseteq \{\lambda x.e_\lambda\} \\ [\![e_1]\!]_{out} \supseteq \{\hat{v}_1\} \qquad [\![e_\lambda]\!]_{out} \supseteq \{\hat{v}\}\end{array}}{[\![e_0\ e_1]\!]_{out} \supseteq \{\hat{v}\}}\ \text{CALL}_{out}$$

Figure 1: 0CFA (with reachability)



(a) A general flow graph node

(b) A composite flow graph

Figure 2: Flow graphs in 0CFA

2.2 Flow-graph implementation of CFA

In order to solve the constraint rules for CFA efficiently, it is common to represent the problem as a flow graph (Jagannathan and Weeks 1995; Heintze and McAllester 1997a) with nodes denoting the flow variables for an expression and directed edges denoting the flow of abstract values from one

node to another. In the present case of *0CFA with reachability*, an edge into an expression node models reachability $\hat{r}$, whereas an edge out of an expression node models (possible) result values $\hat{v}$ as depicted in figure 2a.

For example, in an analysis for a language with conditionals the flow graph for the expression `(if` e_1 e_2 e_3`)` will contain nodes representing both the reachability, $[\![e]\!]_{in}$, and the result value, $[\![e]\!]_{out}$, of the `if`, e_1, e_2, and e_3 expressions. This is depicted schematically in figure 2b. The expressions e_1, e_2, and e_3 are drawn in outline to indicate that they may contain other nodes internal to those expressions. The expression e_1 is reachable if the `if` is reachable so there is an edge from the input of the `if` to e_1. Likewise e_2 and e_3 are reachable if e_1 outputs a true or false value respectively. Thus there are edges from e_1 to e_2 and e_3 filtered by $T?$ and $F?$ to test if the output value contains true or false values respectively. Finally, the output node of the `if` computes the lattice join ($\sqcup$) of the output nodes of e_2 and e_3.

Once the graph is constructed it is iterated until convergence using a standard work-list algorithm. Nodes listed on the work list have out-of-date outputs that need to be updated based on new inputs to the node. Initially all nodes are on the work list. A node is removed from the work list and new values for its output edges calculated based on the value of its input edges. If the value on the outgoing edge changes, the destination nodes of those edges are placed on the work list effectively marking them as out-of-date. The algorithm continues selecting nodes form the work list and recalculating out-of-date nodes until the graph converges and no out-of-date nodes remain.

A crucial property of flow graphs is that, under certain conditions, they quickly converge to a solution. Specifically, if (1) the values that flow through a graph are members of a finite-height lattice, L, (2) for each edge, the value on that edge moves only monotonically up the lattice, and (3) the output values of a node can be computed in constant time from the input values, then the lattice will converge in $O(|L|(|E|+|N|))$ time where $|L|$, $|E|$ and $|N|$ are the height of L, the number of edges, and the number of nodes in the graph, respectively.

For CFA, a minor modification has to be made to the usual flow-graph algorithm. The initial graph contains no links between call sites and functions, so the algorithm adds new edges to the flow graph as it discovers connections between functions and call sites. This does not affect convergence, however, since the maximum number of edges is bounded, and edges are only added, never removed. In the worst case, the algorithm adds an edge between each of $O(n)$ call sites and each of $O(n)$ functions in a program of size n, resulting in a graph with $O(n^2)$ edges.

0CFA uses a lattice over $\wp(\widehat{Lam})$, which has a height equal to the number of functions in the program. We thus have a lattice of height $O(n)$ and a graph of size $O(n^2)$, so a naively implemented 0CFA takes $O(n^3)$ time to compute.

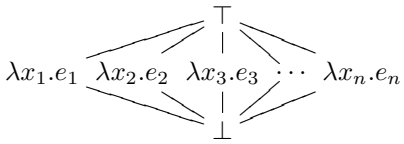


Figure 3: CFA lattice of functions

Slightly faster techniques are known for computing 0CFA but they still take $O(n^3/\log n)$ time (Melski and Reps 2000; Chaudhuri 2008; Midtgaard and Van Horn 2009).

2.3 Top and escaped functions in CFA

If a control-flow analysis is operating on a program that contains free variables, e.g., variables imported from libraries outside the scope of the analysis, then the analysis does not know anything about the values of those variables. This is handled by adding a top element to the lattice denoting an unknown function (Shivers 1988) and using it for the value of free variables. This top element represents not only any function from outside the scope of the analysis but also any function inside the scope of the analysis. Thus top subsumes all other functions in $\widehat{Lam}$.

Likewise, if a function is assigned to a free variable or exported to some library outside the scope of the analysis, then the function may be called in locations unknown to the analysis. The fact that the analysis has lost track of all places where the function flows is represented by marking the function as *escaped*.

Top values and escaped functions can cause more top values or escaped functions. First, if the function position of an application is top, then the return value of the application is top, and the arguments escape, since they are passed to an unknown function. Second, if a function escapes, then its formal parameters become top, and its return value escapes, since it might flow to places outside the scope of the analysis. Finally, when a set of functions are joined with a top value, the result is top, and since a top value does not explicitly mention the functions combined into it, those functions are marked as escaped.

This handling of top and escaped functions is standard and is assumed throughout the rest of this paper even when not explicitly mentioned.

2.4 Sub-0CFA

0CFA takes $O(n^3)$ time even without flow sensitivity but we are aiming for flow sensitivity in $O(n \log n)$ time. So rather than basing our type-recovery analysis on the more common 0CFA, we base it on sub-0CFA which takes only $O(n)$ time. Sub-0CFA bounds both the size of the graph and the height of the lattice by approximating all non-singleton sets of functions with the top element of the lattice. This conservative approximation of 0CFA's power-set lattice has

a constant height and is shown in figure 3. As a result, the values that flow to the function position of a particular call site either contain at most one function or are approximated by the top value and thus add at most a linear number of edges to the graph.

This approach lets more functions escape than in 0CFA, but this is not as bad as it might at first seem. A function flowing to two different places does *not* cause it to escape. Rather, functions escape only when two or more flow to the same point, i.e., when a call site performs some sort of dispatch. For example, when running the analysis over the following both `f` and `g` escape, since they both flow into `fg`, while `h` is not affected.

```
(let ([f (lambda (x) e1)]
      [g (lambda (y) e2)]
      [h (lambda (z) e3)])
  (let ([fg (if (read) f g)])
    (f (g (fg (h (h e4)))))))
```

In the general case Ashley and Dybvig (1998) define sub-0CFA by a projection (widening) operator. When this operator restricts sets of values to either singleton or top values the lattice is effectively constant height. Other projection operators produce lattices that can be of constant or even logarithmic height and result in linear or nearly linear analyses. For example, instead of sets of at most one function, the operator may limit sets to at most k functions for some constant k.

2.5 Non-function types

Programming languages usually have more values than just functions. To handle this we add a fixed set of primitive types, e.g., INT, $PAIR$, etc., to the $\widehat{Val}$ lattice. However, because of the lattice flattening in sub-0CFA we split abstract values into a function part and a non-function part. The function part operates over the same flattened lattice as sub-0CFA, but since we have a fixed number of non-function types we allow the non-function part to operate over the full power-set lattice. Nevertheless, we notationally treat abstract values as sets. For example, $\{INT, PAIR, \lambda x.e\}$ is understood to mean $\langle\{INT, PAIR\}, \{\lambda x.e\}\rangle$.

3. Traditional flow-sensitivity

The control-flow analyses described in section 2 are flow insensitive. This means that all references to a variable are treated as having the same value as the binding site of the variable. Consider the following examples from the introduction.

```
(let ([x (cons e1 e2)]) (car x))

(let ([x (read)]) (if (pair? x) (car x) #f))

(let ([x (read)]) (begin (cdr x) (car x)))
```

With a flow-insensitive analysis, all references to `x` in the first expression are known to be pairs while, in the second and third, they are all treated as $\top$.

Type information can be gained, however, from the explicit and implicit dynamic type checks in the second and third expressions. In the second expression, we can deduce from the explicit pair check, `(pair? x)`, that `x` must be a pair at the point where `car` is called. In the third expression, we can also deduce that `x` must be a pair at the point where `car` is called, since an implicit pair check guarantees that `cdr` returns only if its argument is a pair.

We call information *constructive* when it is learned from operations that construct a value as in the first expression. We call information *observational* when it is learned from operations that observe a value as in the second and third expressions.

Observational information is *restrictive*, since it restricts that type of variable or value, as in the restriction of `x` to the pair type in the those expressions. Observational information can restrict a type to two or more disjoint types, in which case the type is $\bot$. In general, wherever a $\bot$ type occurs, the sequentially following code is unreachable (dead) and can be discarded.

To collect observational information, we must use a flow-sensitive analysis as the type information about a variable is different at different points in the program, e.g., before and after an observer.

We present such an analysis in two stages. First, we present an analysis that is flow sensitive and gathers observational information only from functions like `car` that unconditionally restrict the type of their argument. Our approach to this form of flow sensitivity is standard. Then we generalize this and present an analysis that also gathers observational information from functions that restrict the type of their argument conditionally. For example, the argument of `pair?` is limited to pairs if and only if `pair?` returns true. Our approach to this form of flow sensitivity is novel.

3.1 Flow-sensitivity for unconditional observers

To recover observational information from functions like `car`, an analysis must be flow sensitive. A flow-insensitive analysis takes information about a variable's abstract value directly from its binding site to each reference as the information about a variable is the same at all references to the variable. To be flow-sensitive we trace the flow of the program and adjust the abstract value for in-scope variables along the way. Consider the earlier example that used `(cdr x)` and `(car x)`. With flow-sensitivity, the variable `x` starts at its binding site with the abstract value $\top$. It then flows to `(cdr x)`. On entry to `(cdr x)`, `x` still has the abstract value $\top$. Since `cdr` throws an error and does not return unless its argument is a pair, the analysis learns that `x` is a pair on exit from `(cdr x)`. This then flows to `(car x)`. Thus `(car x)` is only ever called with a pair argument. The

Expressions: $e \in Exp = x \mid \lambda x.e \mid e\ e \mid \mathit{if}\ e\ e\ e \mid e;e \mid c$

Contexts: $\mathbb{K}(e) = \Box \mid (\Box\ e_1) \mid (e_0\ \Box) \mid (\lambda x.\Box) \mid (\Box; e_1) \mid (e_0; \Box) \mid (\mathit{if}\ \Box\ e_1\ e_2) \mid (\mathit{if}\ e_0\ \Box\ e_2) \mid (\mathit{if}\ e_0\ e_1\ \Box)$

Signatures: $[\![e]\!]_{in} \in Bool \times \widehat{Env} \qquad [\![e]\!]_{out} \in \widehat{Val} \times \widehat{Env} \times \widehat{Env} \qquad \mathcal{E}(\lambda x.e) \in \widehat{Env} \qquad \hat{r} \in Bool = \{\bot, \top\}$

$\hat{\rho} \in \widehat{Env} = Var \to \widehat{Val} \qquad \hat{v} \in \widehat{Val} = \widehat{Fun} \times \wp(\widehat{Tag}) \qquad \hat{f} \in \widehat{Fun} = \wp(\widehat{Lam} + \widehat{Prim})$

$\hat{t} \in \widehat{Tag} = \{FALSE, TRUE, INT, FLOAT, PAIR, \ldots\}$

$\widehat{Lam} = \{\lambda x_1.e_1, \lambda x_2.e_2, \lambda x_3.e_3, \ldots\} \qquad o \in \widehat{Prim} = \{\texttt{pair?}, \texttt{car}, \texttt{cdr}, \ldots\}$

$$\frac{[\![c]\!]_{in} \sqsupseteq \langle \top, \hat{\rho} \rangle}{[\![c]\!]_{out} \sqsupseteq \langle ABS(c), \hat{\rho}, \hat{\rho} \rangle}\ \text{CONST} \qquad \frac{[\![\lambda x.e]\!]_{in} \sqsupseteq \langle \top, \hat{\rho} \rangle}{[\![\lambda x.e]\!]_{out} \sqsupseteq \langle \lambda x.e, \hat{\rho}, \hat{\rho} \rangle \qquad \mathcal{E}(\lambda x.e) \sqsupseteq \hat{\rho}}\ \text{LAMBDA}$$

$$\frac{[\![x]\!]_{in} \sqsupseteq \langle \top, \hat{\rho} \rangle \qquad \hat{v}_t = \hat{\rho}(x) \sqcap \top_t \qquad \hat{v}_f = \hat{\rho}(x) \sqcap \top_f}{[\![x]\!]_{out} \sqsupseteq \langle \hat{v}_t \sqcup \hat{v}_f, \hat{\rho}[x \mapsto \hat{v}_t], \hat{\rho}[x \mapsto \hat{v}_f] \rangle}\ \text{VAR}$$

$$\frac{[\![e_0\ e_1]\!]_{in} \sqsupseteq \langle \hat{r}, \hat{\rho} \rangle}{[\![e_0]\!]_{in} \sqsupseteq \langle \hat{r}, \hat{\rho} \rangle}\ \text{CALL}_{in} \qquad \frac{[\![e_0]\!]_{out} \sqsupseteq \langle \hat{v}_0, \hat{\rho}_t, \hat{\rho}_f \rangle \qquad \mathbb{K}(e_0) = (\Box\ e_1)}{[\![e_1]\!]_{in} \sqsupseteq \langle \top, \hat{\rho}_t \sqcup \hat{\rho}_f \rangle}\ \text{CALL}_{mid}$$

$$\frac{[\![e_0]\!]_{out} \sqsupseteq \langle \{\lambda x.e_\lambda\}, \hat{\rho}_t^{e_0}, \hat{\rho}_f^{e_0} \rangle \qquad [\![e_1]\!]_{out} \sqsupseteq \langle \hat{v}_1, \hat{\rho}_t^{e_1}, \hat{\rho}_f^{e_1} \rangle \qquad \mathbb{K}(e_1) = (e_0\ \Box)}{[\![e_\lambda]\!]_{in} \sqsupseteq \langle \top, \mathcal{E}(\lambda x.e_\lambda) \rangle}\ \text{CALL}_{fun}$$

$$\frac{[\![e_0]\!]_{out} \sqsupseteq \langle \{\hat{f}\}, \hat{\rho}_t^{e_0}, \hat{\rho}_f^{e_0} \rangle \quad [\![e_1]\!]_{out} \sqsupseteq \langle \hat{v}_1, \hat{\rho}_t^{e_1}, \hat{\rho}_f^{e_1} \rangle \quad \langle \hat{v}, \hat{v}_t, \hat{v}_f \rangle = RET(\hat{f}) \quad \forall i \in \{t, f\}.\hat{\rho}_i' = ARG(\hat{\rho}_t^{e_1} \sqcup \hat{\rho}_f^{e_1}, e_1, \hat{v}_1 \sqcap \hat{v}_i)}{[\![e_0\ e_1]\!]_{out} \sqsupseteq \langle \hat{v}, \hat{\rho}_t', \hat{\rho}_f' \rangle}\ \text{CALL}_{out}$$

$$\frac{[\![e_0; e_1]\!]_{in} \sqsupseteq \langle \hat{r}, \hat{\rho} \rangle}{[\![e_0]\!]_{in} \sqsupseteq \langle \hat{r}, \hat{\rho} \rangle}\ \text{SEQ}_{in} \qquad \frac{[\![e_0]\!]_{out} \sqsupseteq \langle \hat{v}_0, \hat{\rho}_t, \hat{\rho}_f \rangle \qquad \mathbb{K}(e_0) = (\Box; e_1)}{[\![e_1]\!]_{in} \sqsupseteq \langle \top, \hat{\rho}_t \sqcup \hat{\rho}_f \rangle}\ \text{SEQ}_{mid} \qquad \frac{}{[\![e_0; e_1]\!]_{out} \sqsupseteq [\![e_1]\!]_{out}}\ \text{SEQ}_{out}$$

$$\frac{[\![\mathit{if}\ e_0\ e_1\ e_2]\!]_{in} \sqsupseteq \langle \hat{r}, \hat{\rho} \rangle}{[\![e_0]\!]_{in} \sqsupseteq \langle \hat{r}, \hat{\rho} \rangle}\ \text{IF}_{in} \qquad \frac{[\![e_0]\!]_{out} \sqsupseteq \langle \hat{v}_0, \hat{\rho}_t, \hat{\rho}_f \rangle \qquad \mathbb{K}(e_0) = (\mathit{if}\ \Box\ e_1\ e_2)}{[\![e_1]\!]_{in} \sqsupseteq \langle \top, \hat{\rho}_t \rangle \qquad [\![e_2]\!]_{in} \sqsupseteq \langle \top, \hat{\rho}_f \rangle}\ \text{IF}_{mid}$$

$$\frac{}{[\![\mathit{if}\ e_0\ e_1\ e_2]\!]_{out} \sqsupseteq [\![e_1]\!]_{out} \sqcup [\![e_2]\!]_{out}}\ \text{IF}_{out}$$

$$ARG(\hat{\rho}, e, \hat{v}) = \begin{cases} \bot & \text{if } \hat{v} = \bot \\ \hat{\rho} & \text{if } \hat{v} \neq \bot \wedge e \notin Var \\ \hat{\rho}[e \mapsto \hat{\rho}(e) \sqcap \hat{v}] & \text{if } e \in Var \end{cases}$$

$$ABS(vs) = \{ABS(v) \mid v \in vs\} \qquad ABS(\texttt{\#f}) = FALSE \qquad ABS(n) = INT \qquad \ldots$$

$$\top_t = \widehat{Val} \backslash \{FALSE\} \qquad \top_f = \{FALSE\}$$

$$RET(\hat{f}) = \begin{cases} \langle \top, \top, \top \rangle & \text{if } \hat{f} = \top \\ \langle \bot, \bot, \bot \rangle & \text{if } \hat{f} = \bot \\ \langle \hat{v}, \hat{\rho}_t(x), \hat{\rho}_f(x) \rangle & \text{if } \hat{f} = \lambda x.e \text{ where } \langle \hat{v}, \hat{\rho}_t, \hat{\rho}_f \rangle = [\![e]\!]_{out} \\ \langle \top, \{PAIR\}, \{PAIR\} \rangle & \text{if } \hat{f} = \texttt{car} \\ \langle \{FALSE, TRUE\}, \{PAIR\}, \top \backslash \{PAIR\} \rangle & \text{if } \hat{f} = \texttt{pair?} \\ & \ldots \end{cases}$$

Figure 4: Analysis constraint rules

earlier `cdr` prevents non-pair values from flowing to the `car`, so we can safely omit the implicit pair check in the `car`.

To gather restrictive information, each function is annotated with the abstract value each argument must be for the function to return. For example, if `car` returns, then its argument must be a pair. This is not limited to built-in primitives. For user-defined functions we examine the abstract value of each formal parameter after flowing through the function body. In the following example, if `g` returns, then we know its argument, `y`, is a pair. Thus `x` must be a pair after returning from `(g x)` and consequently the implicit pair check in `cdr` is redundant and can be safely omitted.

```
(let ([x (read)]
      [g (lambda (y) (+ (car y) 1))])
  (g x)
  (cdr x))
```

The formal semantics for an analysis with flow sensitivity for unconditional observers is a straightforward extension of the analysis in figure 1 for 0CFA. It includes sequencing by threading the environment through the evaluation flow of the program. Each expression still has an associated reachability flag and result value, but now each expression also has two environments associated with it. One tracks the types of variables when entering the expression. The other tracks the type of variables when exiting the expression. At each function call, arguments are restricted to only those abstract values that are compatible with the particular function returning. For example, after `(car x)`, `x` is restricted to pairs. Both the entering and exiting environments are treated as reduced abstract domains (Cousot and Cousot 1979) thus equating abstract elements with the same meaning (concretization). Hence if any component of an environment is $\perp$ then all components of the environment are forced to $\perp$. For example, if `x` is known to be an integer, then after `(car x)`, `x` is $\perp$. This causes all components of the exiting environment to be $\perp$. In addition as part of the reduced abstract domain the return value of `(car x)` is $\perp$. This models the fact that `(car x)` does not return if `x` is an integer.

This simple form of flow sensitivity handles functions like `car` that unconditionally provide observational information about their arguments when they return. However, it fails to handle predicates such as `pair?`. This is because the fact that `pair?` returns says nothing by itself about the abstract value of its argument. Rather, the type information is conditional. Whether it returns a true or false value tells us whether its argument is a pair.

3.2 Flow-sensitivity for conditional observers

In order to handle conditional or predicated observers we generalize the environments flowing through the program. On exit from an expression we record one environment for when it returns a true value and another for when it returns a false value. The environment recorded for entry to the expression remains the same as before.

Figure 4 presents this formally. It includes two environments in $[\![e]\!]_{out}$. One contains abstract value information for when e returns true values and the other when e returns false values. For example, $[\![(\texttt{pair?}\ x)]\!]_{out}$ has `x` as a pair in the true environment and as a non-pair in the false environment. These true and false environments are used by `if`. The true environment of the test flows to the entering environment of the true branch. The false environment of the test flows to the entering environment of the false branch.

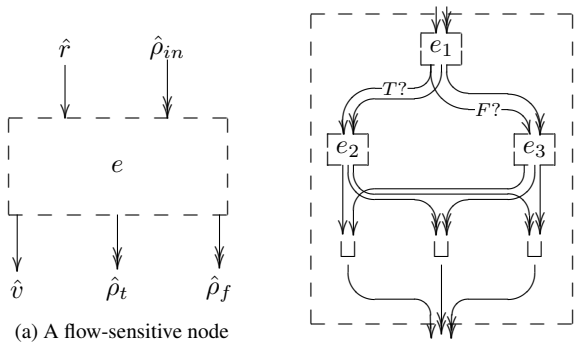
In the semantics of figure 4, gathering restrictive information from a function call, e.g., `(car x)` or `(pair? x)`, is implemented by the CALL_{out} rule. First, the values and environments that flow out of e_1 and e_2 are collected. Next, RET examines any functions, $\hat{f}$, flowing out of e_0 and returns three abstract values. One value is the return value of $\hat{f}$. The other two are the values that the argument to $\hat{f}$ must be for $\hat{f}$ to return either true or false respectively. Finally, ARG uses this information to determine for both the true and false cases if the function could return to this call site given the abstract value of the call site's argument. For example, `(car 3)` does not return. If the argument is a variable, ARG restricts the variable in the environment to the appropriate abstract value. Thus after `(pair? x)`, `x` is restricted to pairs and non-pairs in the true and false cases respectively.

Each primitive or function is annotated with one abstract value for when the primitive or function returns true and another for when it returns false. For example, `pair?` is annotated with the abstract value for pairs for the true case and the abstract value for non-pairs for the false case. Unconditional observers like `car` are annotated with the same abstract value in both the true and false cases. For user-defined functions, the same information is obtained from the exiting true and false environments of the body of the function.

3.3 Flow-graph representation of flow-sensitivity

As with standard flow-sensitive CFA these constraint rules can be implemented in terms of a flow graph. Yet, while information can flow directly from variable bindings to variable references in a flow-insensitive CFA, this is not sufficient in a flow-sensitive CFA. Instead, the abstract value for a variable must be threaded through each expression. In addition to the usual reachability flags and result abstract values, we associate an environment with each entry edge of an expression and two environments (true and false) with each exit edge of an expression as depicted in figure 5a. The single-arrowhead lines represent the flow of single values and the double-arrowhead lines represent the flow of environments. Figure 5b shows how to extend the composite flow graph for `if` from figure 2b to account for the extra edges and illustrates the flow of the true and false environments.

This graph formulation still has only linearly many edges, but some of these edges now flow environments. The lattice of an environment is the Cartesian product of the lattice for each variable, so the lattice height of an environment is linear



(a) A flow-sensitive node

(b) A composite flow graph

Figure 5: Flow graphs for flow-sensitive 0CFA

in the number of variables. Thus, the flow graph has a linear number of edges with linear height lattices and consequently takes quadratic time to converge in the worst case.

4. Efficient flow-sensitivity

The flow-graph based algorithm for flow-sensitive CFA described in section 3 is quadratic because the environments are threaded through each expression. Contrast this with flow-insensitive sub-0CFA where the abstract value of a variable flows directly from its binding location to each reference without threading through intervening expressions. To efficiently implement flow sensitivity, we adopt a similar approach. However, instead of flowing abstract values from a variable binding site to each reference, we flow the information directly from one occurrence of the variable to the next, following the control flow of the program. The abstract value of a variable is adjusted at each occurrence as appropriate.

To see how this works, first consider what the flow-sensitive analysis from section 3 does in the following examples. Consider, for example, how abstract values for `x` flow from `(pair? x)` to `(car x)` in the following expression. Assume `x` is not referenced in e_1 or e_2. Such expressions can arise via the expansion of boolean connectives such as `and`, `or`, and `not`.

```
(if (if (pair? x) e1 e2) (car x) ...)
```

What the flow-sensitive analysis learns about `x` at `(car x)` depends on the return values of e_1 and e_2. If e_1 evaluates to only true values and e_2 evaluates to only false values, then `x` at `(car x)` is always a pair. On the other hand, if e_1 evaluates to only false values and e_2 evaluates to only true values, then `x` at `(car x)` is never a pair. Other cases arise if either expression diverges or returns both true and false values.

Likewise, consider the following expression where restrictive information is learned from `(car x)` in one of the branches of the inner `if`. As before, assume `x` is not referenced in e_0, e_1, or e_2.

```
(let ([x (read)])
  (if (if e0 (begin (car x) e1) e2)
   (cdr x)
   ...))
```

After `(car x)`, `x` is known to be a pair, but the abstract value of `x` at `(cdr x)` depends upon the return values of e_1 and e_2. If e_2 returns a true value then `(cdr x)` is reachable without passing through the `(car x)`. If e_2 evaluates to only false values, however, then all paths to `(cdr x)` go through `(car x)`, and `x` at `(cdr x)` must be a pair. Interestingly, the return value of e_0 is not needed to determine this, since if e_2 always evaluates to a false value, then `(cdr x)` is reachable only when e_0 is a true value and sends control through `(car x)`

The remainder of this section derives an optimized algorithm that takes all such cases into account and produces the same results as the traditional algorithm, but requires only linear-log time. Section 4.1 starts by defining theorems about how to move abstract values from one place to another. Then section 4.2 gives an algorithm to determine where to copy abstract value information. Section 4.3 defines a variable-independent auxiliary cache so the transformations defined in section 4.1 can be computed efficiently. Finally, section 4.4 puts all of these together into a linear-log time algorithm that computes results identical to those of the traditional algorithm.

4.1 Context skipping

In the earlier example with `pair?`, the traditional algorithm first flows the abstract value of `x` from the exiting environments of `(pair? x)` into the entering environments of e_1 and e_2, then through e_1 and e_2, and finally from the exiting environments of e_1 and e_2 through both `if` expressions and into `(cdr x)`. When flowing through e_1 and e_2, the abstract value of `x` is threaded through every subexpression of e_1 and e_2. However, it turns out that if `x` is not referenced in e_1 and e_2, then these expressions can be skipped. Intuitively, the true and false environments that contain `x` might be swapped or joined, but the value of `x` in each environment does not fundamentally change. The following lemma reflects this intuition.

Lemma 4.1 (Expression Skipping)**.** *If* $\mathtt{x}$ *is a variable not mentioned in* e *then*

$$\langle \hat{\rho}_t(x),\ \hat{\rho}_f(x)\rangle = \langle \mathcal{T}?(e,\ \hat{\rho}_{in}(x)),\ \mathcal{F}?(e,\ \hat{\rho}_{in}(x))\rangle$$

where $\langle \hat{r},\ \hat{\rho}_{in}\rangle = [\![e]\!]_{in}$

and $\mathcal{T}?(e,\ u)$ *and* $\mathcal{F}?(e,\ u)$ *are as defined in figure 6.*

Proof. By induction on e and constraint rules in figure 4. □

This lemma allows us to directly compute the abstract values of `x` at the end of e_1 and e_2 given the abstract values at the

$$\mathcal{T}?(e,\ u) = (\hat{v} \sqcap \top_t) \neq \bot\ ?\ u\ :\ \bot \text{ where } \langle \hat{v},\ \hat{\rho}_t,\ \hat{\rho}_f \rangle = [\![e]\!]_{out}$$
$$\mathcal{F}?(e,\ u) = (\hat{v} \sqcap \top_f) \neq \bot\ ?\ u\ :\ \bot \text{ where } \langle \hat{v},\ \hat{\rho}_t,\ \hat{\rho}_f \rangle = [\![e]\!]_{out}$$

Figure 6: True and false expression guards

$$\mathcal{V}_{C,e}\langle \hat{v}_t,\ \hat{v}_f,\ \hat{v}_{in} \rangle = \langle \mathcal{T}?(C[e],\ \hat{v}'_t),\ \mathcal{F}?(C[e],\ \hat{v}'_f),\ \hat{v}'_{in} \rangle$$
$$\text{where } \langle \hat{v}'_t,\ \hat{v}'_f,\ \hat{v}'_{in} \rangle = \mathcal{V}'_C\langle \hat{v}_t,\ \hat{v}_f,\ \hat{v}_{in} \rangle$$

$$\begin{aligned}
\mathcal{V}'_{(if\ \square\ e_2\ e_3)}\langle \hat{v}_t,\ \hat{v}_f,\ \hat{v}_{in} \rangle &= \langle \hat{v}'_t,\ \hat{v}'_f,\ \hat{v}_{in} \rangle \\
\text{where } \hat{v}'_t &= \mathcal{T}?(e_2,\ \hat{v}_t) \sqcup \mathcal{T}?(e_3,\ \hat{v}_f) \\
\hat{v}'_f &= \mathcal{F}?(e_2,\ \hat{v}_t) \sqcup \mathcal{F}?(e_3,\ \hat{v}_f) \\
\mathcal{V}'_{(if\ e_1\ \square\ e_3)}\langle \hat{v}_t,\ \hat{v}_f,\ \hat{v}_{in} \rangle &= \langle \hat{v}'_t,\ \hat{v}'_f,\ \hat{v}_{in} \rangle \\
\text{where } \hat{v}'_t &= \hat{v}_t \sqcup \mathcal{T}?(e_3,\ \hat{v}_{in}) \\
\hat{v}'_f &= \hat{v}_f \sqcup \mathcal{F}?(e_3,\ \hat{v}_{in}) \\
\mathcal{V}'_{(if\ e_1\ e_2\ \square)}\langle \hat{v}_t,\ \hat{v}_f,\ \hat{v}_{in} \rangle &= \langle \hat{v}'_t,\ \hat{v}'_f,\ \hat{v}_{in} \rangle \\
\text{where } \hat{v}'_t &= \hat{v}_t \sqcup \mathcal{T}?(e_2,\ \hat{v}_{in}) \\
\hat{v}'_f &= \hat{v}_f \sqcup \mathcal{F}?(e_3,\ \hat{v}_{in}) \\
\mathcal{V}'_{(\lambda x.\square)}\langle \hat{v}_t,\ \hat{v}_f,\ \hat{v}_{in} \rangle &= \langle \hat{v}_{in},\ \hat{v}_{in},\ \hat{v}_{in} \rangle \\
\mathcal{V}'_{(e_2\ \square)}\langle \hat{v}_t,\ \hat{v}_f,\ \hat{v}_{in} \rangle &= \langle \hat{v}_t \sqcup \hat{v}_f,\ \hat{v}_t \sqcup \hat{v}_f,\ \hat{v}_{in} \rangle \\
\mathcal{V}'_{(\square\ e_2)}\langle \hat{v}_t,\ \hat{v}_f,\ \hat{v}_{in} \rangle &= \langle \hat{v}_t \sqcup \hat{v}_f,\ \hat{v}_t \sqcup \hat{v}_f,\ \hat{v}_{in} \rangle \\
\mathcal{V}'_{(\square;e_2)}\langle \hat{v}_t,\ \hat{v}_f,\ \hat{v}_{in} \rangle &= \langle \hat{v}_t \sqcup \hat{v}_f,\ \hat{v}_t \sqcup \hat{v}_f,\ \hat{v}_{in} \rangle \\
\mathcal{V}'_{(e_1;\square)}\langle \hat{v}_t,\ \hat{v}_f,\ \hat{v}_{in} \rangle &= \langle \hat{v}_t,\ \hat{v}_f,\ \hat{v}_{in} \rangle
\end{aligned}$$

Figure 7: Context skipping function

start of e_1 and e_2. However, it deals only with the flow of abstract values from the entry of an expression to its exit. As seen in the examples we are more interested in how abstract value information flows from an expression to its surrounding context. In the preceding examples abstract value information flows from the outputs of `(pair? x)` and `(car x)` to the output of the surrounding `(if (pair? x)` e_1 e_2`)` and `(if` e_0 `(begin (car x)` e_1`)` e_2`)` respectively. To account for this we compute the flow across a context by means of the context-skipping function $\mathcal{V}_{C,e}$ in figure 7. Given an expression, e, in a single-layer context, C, it computes the abstract value of a variable in the exit environments of $C[e]$ given the abstract value in the exit environments of e and the entry environment of $C[e]$. The following lemma states this formally. Here again the intuition is that the true and false information about a variable might join or swap but do not fundamentally change.

Lemma 4.2 (Single-Layer Context Skipping)**.** *If $\boldsymbol{x}$ is a variable not mentioned in the single-layer context C then*

$$\mathcal{V}_{C,e}\langle \hat{\rho}^c_t(x),\ \hat{\rho}^c_f(x),\ \hat{\rho}_{in}(x) \rangle = \mathcal{V}_{C,e}\langle \hat{\rho}^e_t(x),\ \hat{\rho}^e_f(x),\ \hat{\rho}_{in}(x) \rangle$$
$$\textit{where } \langle \hat{v}_e,\ \hat{\rho}^e_t,\ \hat{\rho}^e_f \rangle = [\![e]\!]_{out}$$
$$\langle \hat{v}_c,\ \hat{\rho}^c_t,\ \hat{\rho}^c_f \rangle = [\![C[e]]\!]_{out} \quad \langle \hat{r},\ \hat{\rho}_{in} \rangle = [\![C[e]]\!]_{in}\,.$$

Proof. By lemma 4.1, constraint rules and unfolding. □

In addition to this context-skipping function and lemma for values exiting a context, there are a corresponding context-skipping function and lemma for values entering a context. They are omitted here as they are straightforward and are equivalent to a simple reachability check.

Lemma 4.2 only handles single layer contexts. For composite, multilayer contexts, the following lemma applies. (This is also the reason why the tuple returned by $\mathcal{V}_{C,e}$ has a third component even though it is unused in lemma 4.2.)

Theorem 4.3 (Multilayer Context Skipping)**.** *If $\boldsymbol{x}$ is a variable not mentioned in a single-layer or multilayer context C then the equation from lemma 4.2 holds where $\mathcal{V}_{C,e}$ on a composite context is*

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

Proof. By induction on C and lemma 4.2. □

Critically, even under composition the universe of possible $\mathcal{V}_{C,e}$ is small and finite. Any particular $\mathcal{V}_{C,e}$ can be represented in a constant-size, canonical form as stated in the following theorem.

Theorem 4.4 (Canonical Skipping Functions)**.**
For any particular C, e, $[\![e]\!]_{out}$ and $[\![C[e]]\!]_{out}$,

$$\mathcal{V}_{C,e}\langle \hat{v}_t,\ \hat{v}_f,\ \hat{v}_{in} \rangle = \langle \bigsqcup T,\ \bigsqcup F,\ \hat{v}_{in} \rangle$$

for some $T, F \subseteq \{\hat{v}_t, \hat{v}_f, \hat{v}_{in}\}$.

Proof. By induction on C and unfolding $\mathcal{V}_{C,e}$. □

Intuitively there are only so many ways to join and swap the true and false values of a variable. Diagrammatically these canonical forms are all sub-graphs of the graph in figure 8 that omit zero or more edges leading to the three abstract join ($\sqcup$) nodes. Functions of this form have compact, constant-size representations, are closed under composition, and form a finite height lattice that they monotonically climb when $[\![e]\!]_{out}$ and $[\![C[e]]\!]_{out}$ climb the abstract value lattice.

When composing $\mathcal{V}_{C,e}$ we always reduce the composition to this canonical form as a matter of course. Thus all $\mathcal{V}_{C,e}$ can be applied to a given value in constant time even if the $\mathcal{V}_{C,e}$ is from the composition of many $\mathcal{V}_{C,e}$. We take advantage of this to flow abstract values across multilayer contexts quickly. Since $\mathcal{V}_{C,e}$ is the same for all variables not in C, we can compute $\mathcal{V}_{C,e}$ once and use it for all variables

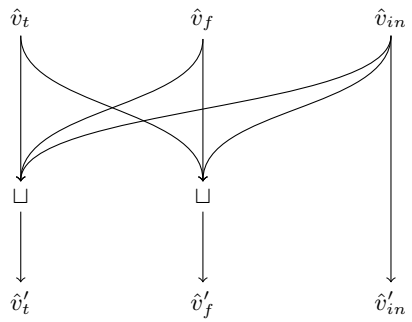


Figure 8: Graph form of canonical skipping functions

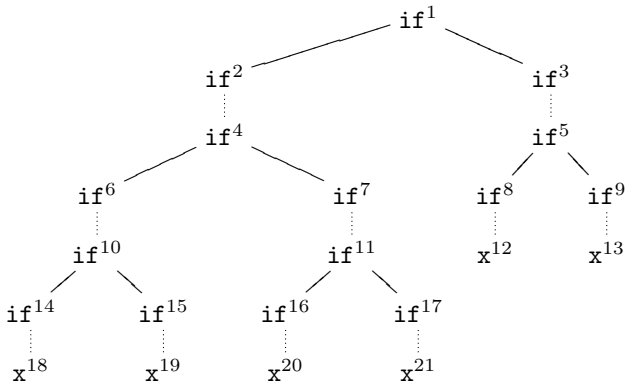


Figure 9: Example AST for skipping context selection

not in C. Section 4.3 shows how to efficiently compute and update these compositions when $[\![e]\!]_{out}$ and $[\![C[e]]\!]_{out}$ change.

This skipping function is the key insight of our technique. We still have to choose which contexts to skip and how to efficiently compute $\mathcal{V}_{C,e}$, but those aspects of the algorithm are only so that we can use skipping functions to more efficiently flow information through the program.

4.2 Selecting context skips

We now have two ways to flow abstract values through a program. The first is the constraint rules in figure 4. The second is the skipping function, $\mathcal{V}_{C,e}$. For each variable, we use a combination of these methods that ensures the analysis takes only linear-log time while maintaining semantic equivalence with the traditional algorithm.

We do this by selecting the longest skips for which theorem 4.3 holds for a given variable and fall back to the constraint rules in figure 4 when it does not. A different set of skips is selected for each variable. We select longest skips for a particular variable, x, by starting with each reference to it, e, and finding the largest context, C, of e that does not contain x. C is then one of the contexts that we skip.

Since C is the largest context of e not containing x, the parent of $C[e]$, p, contains references to x other than the ones in e. Thus we cannot use theorem 4.3 to skip past p, and at p we fall back to the constraint rules from figure 4. We repeat the process by finding the largest context of p that does not contain x and choose that context as a skip. This repeats until we have all the skips needed to flow x through the entire program.

As an example consider the abstract-syntax tree in figure 9 and selecting skips for `x`. The dotted edges in the diagram represent multiple layers of the abstract-syntax tree that are omitted and which do not contain references to `x`.

To select the skips, initially all references to `x` are examined. In this case that is expressions 12, 13, 18, 19, 20 and 21. For each such expression, the largest context not containing `x` is selected. For expression 12, this is the context going from expression 12 to just past expression 8. For expression 13, this is the context going just past expression 9, and so on. The parents of these contexts are places where the constraint rules are used instead of the context skipping function. For example, for moving type information about `x` from expression 8 to its parent, expression 5, theorem 4.3 does not hold and the context skipping function cannot be used because of the reference to `x` in expression 9, the other child of expression 5.

The process repeats with the parents of each of the skips. For example, expression 5 is the parent of the contexts ending at expression 8 and expression 9 so the algorithm selects largest context of expression 5 that does not contain `x`. Likewise for expressions 10 and 11.

In the end the only places where the algorithm falls back to using the constraint rules are expressions 1, 4, 5, 10, and 11. Everywhere else uses context skipping functions. The entire scope of x is tessellated by the skipping contexts and the points where we fall back to the constraint rules.

This part of the algorithm is linear because the selected skips form an implicit tree structure. The expressions at which we use the constraint rules are the nodes of the tree. The contexts being skipped are the edges of the tree. The references to the variable are the leaves. Since the numbers of edges and nodes in a tree are linearly bounded by the number of leaves, the number of skips and the number of uses of the constraint rules for a particular variable are both linearly bounded by the number of references to that variable. Summing over all variables we are linear in the size of the program.

Finding the largest context not containing a particular variable is the most computationally complex part of this process. It is implemented in terms of a lowest common ancestor algorithm (Aho et al. 1973; Alstrup et al. 2004) that takes linear time for preprocessing and constant time for each query. Finding the largest skips amounts to finding the lowest common ancestor of an expression and the imme-

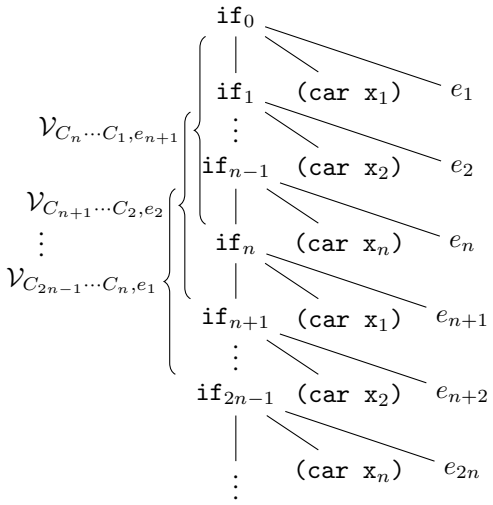


Figure 10: Example of quadratic $\mathcal{V}_{C,e}$ calculation

diately preceding and following references to the variable being considered.

4.3 Caching context skips

Theorem 4.3 allows us to skip over a context and move abstract value information quickly across multiple layers. Once the $\mathcal{V}_{C,e}$ is computed and reduced to the canonical form by theorem 4.4, it takes only constant time to move abstract value information across C for any variable not referenced in C.

However, we must be careful that the total time to construct the various $\mathcal{V}_{C,e}$ does not exceed our linear-log time bound. For example, consider the abstract-syntax tree in figure 10 where a different $\mathcal{V}_{C,e}$ is needed for each of the n variables and each context is n layers deep. Computing the $\mathcal{V}_{C,e}$ for each variable separately would take $O(n^2)$ time.

To ensure a linear-log time bound we keep a cache of $\mathcal{V}_{C,e}$ for selected C such that

- only linear-log many $\mathcal{V}_{C,e}$ are stored in the cache,
- for any C, a $\mathcal{V}_{C,e}$ can be computed from the composition of only logarithmically many $\mathcal{V}_{C,e}$ from the cache, and
- when more abstract value information is learned about an expression, only logarithmically many $\mathcal{V}_{C,e}$ in the cache need to be updated and each $\mathcal{V}_{C,e}$ takes only constant time to update.

The cache can be thought of as starting with the single-layer $\mathcal{V}_{C,e}$. That is, it stores the skipping information necessary to flow any variable by a single step from the exit environment of one expression to the enclosing expression's exit environment. If the cache stores only these, then when the abstract value of an expression changes, it takes only constant time to update, but the $\mathcal{V}_{C,e}$ for multilayer contexts require the composition of linearly many $\mathcal{V}_{C,e}$ from the cache.

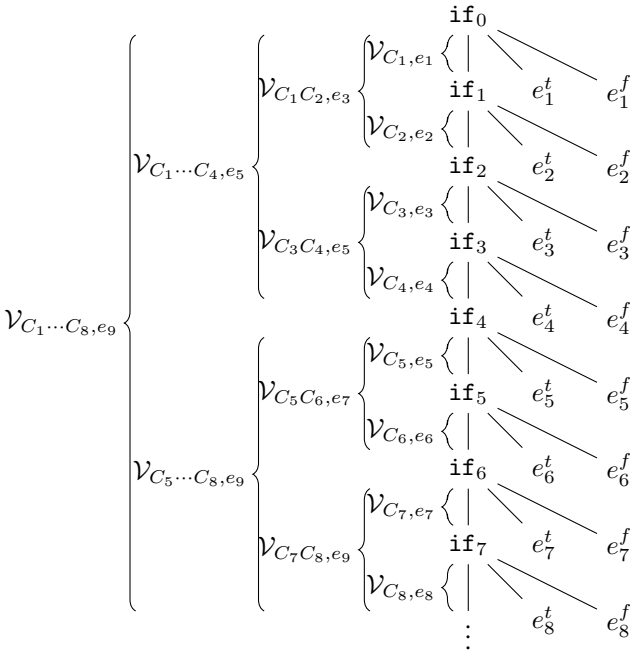


Figure 11: Layered structure of the $\mathcal{V}_{C,e}$ cache

Next, the single-layer $\mathcal{V}_{C,e}$ are paired together. Each single layer C that goes from depth $2k$ to depth $2k+1$ is paired with each of its single-layer, child contexts which go from depth $2k+1$ to $2k+2$. The $\mathcal{V}_{C,e}$ for each of these pairings is included in the cache. These double-layer $\mathcal{V}_{C,e}$ are then also paired together. Each double-layer C that goes from depth $4k$ to depth $4k+2$ is paired with each of its double-layer, child contexts which go from depth $4k+2$ to $4k+4$. The $\mathcal{V}_{C,e}$ for each of these pairings is also included in the cache. This process continues iteratively, pairing each 2^i-layer context that goes from depth $2^{i+1}k$ to depth $2^{i+1}k+2^i$ with each of its 2^i-layer child contexts which go from depth $2^{i+1}k+2^i$ to $2^{i+1}k+2^{i+1}$.

As an example of this, the cached values for one path down a program tree are depicted in figure 11. The same pairing of values occurs on all other paths down the program tree. Each $\mathcal{V}_{C,e}$ that is shared between different paths is stored only once in the cache.

This selection of cached $\mathcal{V}_{C,e}$ has the three important properties that ensure our linear-logarithmic bound. First, only linear-log many $\mathcal{V}_{C,e}$ functions are cached, since only logarithmically many $\mathcal{V}_{C,e}$ are cached for any particular e. Second, any $\mathcal{V}_{C,e}$ that is not cached can be computed from the composition of logarithmically many cached $\mathcal{V}_{C,e}$. Third, when the $\mathcal{V}_{C,e}$ for a single-layer context is updated, the double-layer $\mathcal{V}_{C,e}$ composed from it are also updated.

If the new double-layer $\mathcal{V}_{C,e}$ changes as a result, then the quadruple-layer $\mathcal{V}_{C,e}$ composed from it are updated. Thus, when abstract value information is learned about an expression, at most logarithmically many $\mathcal{V}_{C,e}$ in the cache are updated and since each multilayer $\mathcal{V}_{C,e}$ in the cache is composed of exactly two $\mathcal{V}_{C,e}$, each update takes constant time.

This caching strategy can be generalized by considering the path from each expression to the root. Storing this path as a perfectly balanced variation of a skip list (Pugh 1990) is equivalent to the caching strategy just described. However by using a variation of Myers applicative random access stacks (Myers 1984), the number of cached values and the total time spent updating the cache both become linear in the size of the program. For an arbitrary C, computing $\mathcal{V}_{C,e}$ may still require logarithmically many cached values, so this does not improve the overall asymptotic bounds, but it improves the constants involved. This is the representation used by the implementation described in section 5.

4.4 Algorithm summary

Putting all these pieces together the optimized algorithm works as follows. First, as described in section 4.3, the cache of skip functions, $\mathcal{V}_{C,e}$ is constructed using flow-graph nodes. This creates linearly many nodes in linear time. Next, for each variable, context skips are selected as described in section 4.2, and flow-graph nodes are constructed that take logarithmically many $\mathcal{V}_{C,e}$ from the cache and build a $\mathcal{V}_{C,e}$ for the skipped context. In total there are linearly many context skips and each one involves composing logarithmically many skipping functions. Each composition takes one flow-graph node, so in total this process creates linear-log nodes in linear-log time. Finally, for each non-skipping point where a variable is referenced or the constraint rules are used for a particular variable, a flow-graph node is constructed that computes the type of the variable at that point in terms of the non-skipping points that flow to the point and the $\mathcal{V}_{C,e}$ that skips from them to the current non-skipping point. A similar process is used for flows entering rather than exiting a context. Since in total there are linearly many skipping points, this creates linearly many nodes in linear time. Overall this entire process then takes linear-log time to construct the flow graph and produces a flow graph with a linear-log number of nodes. The values flowing over the edges of the graph all monotonically increase over constant-height lattices, and nodes recompute in terms of their inputs in constant time. Thus, the flow-graph for the optimized analysis converges in linear-log time.

5. Implementation

We have implemented the CFA algorithm described in section 4 and incorporated it into the Chez Scheme (Dybvig 2010) compiler. It is used to perform type recovery and justify the elimination of run-time type checks. The implementation supports the full Scheme language and successfully compiles and runs both Chez Scheme itself and the entire Chez Scheme test suite without errors.

5.1 Implementation structure

To implement type recovery, a post-processing pass is added after the CFA pass. The post-processing pass uses the type information gathered during the CFA pass to determine where run-time type checks are unnecessary. Primitive calls where some or all of the run-time type checks are unnecessary are replaced by an "unsafe" variant of the call which does not perform the unnecessary run-time type check. For instance `(car x)` is replaced by `(unsafe-car x)` when `x` is determined to be a pair. If a primitive makes multiple run-time type checks and only some of those type checks can be omitted, then a "semi-unsafe" variant is used. These cases arise when a primitive does more than one run-time type check or when the checks involve information not tracked by the analysis. For example, a vector range check cannot be eliminated because the analysis does not track the lengths of individual vectors. Another example is when the analysis determines that the vector argument of a `vector-ref` is always a vector but not that the index argument is always a nonnegative integer.

5.2 Implementation features

Our implementation handles a variety of language constructs and features that are not described in section 4. Among these are mutable variables and the unspecified order of evaluation for function call arguments and `let` bindings.

A mutable variable's type can change between the site where type information is recovered and its next use. For instance, an intervening function call could arbitrarily mutate the variable and invalidate what is learned.[3] Thus, for mutable variables, our implementation gathers only constructive information.

The unspecified order of evaluation for function-call arguments and `let` bindings can be handled by choosing a fixed evaluation order prior to this analysis. At present, however, the decision is made later in the compiler, during register allocation. Instead, we process function-call arguments independently, as we do for the branches of an `if`. While the resulting environments are unioned for `if`, they are intersected for function-call arguments. The bindings of a `let` are handled similarly.

5.3 Effectiveness

We tested the effectiveness of the type-recovery algorithm on a standard set of R6RS Benchmarks (Clinger 2008). Each test is run first with type recovery enabled and then with type recovery disabled. The number of type checks performed at run-time are then compared. An average of 71.6% of type

[3] This issue arises only in higher-order languages. The analysis can process restrictive information for mutable variables in first-order languages, including, for example, the output language of a closure-conversion pass in a typical compiler for a higher-order language.

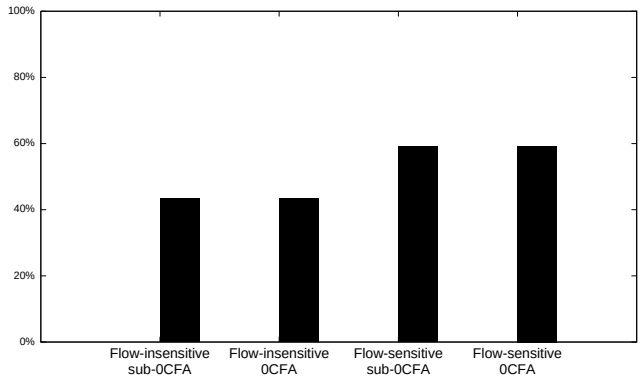


Figure 12: Percent of type checks removed

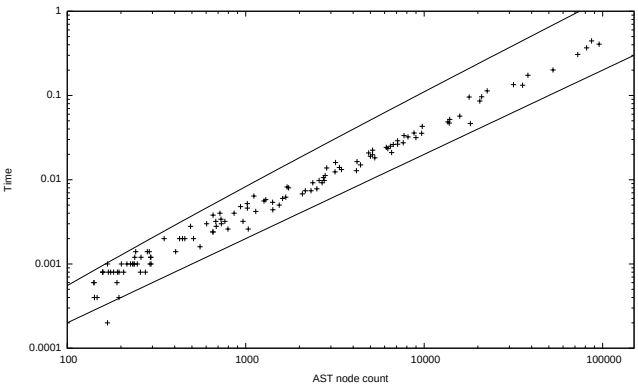


Figure 13: Source node count versus analysis time

checks are eliminated from the code, which results in 59.1% fewer type checks at run-time. We compared these results with a flow-insensitive version of this analysis and both flow-insensitive and flow-sensitive versions of 0CFA. On average, flow-insensitive analysis peformed significantly worse, with the sub-0CFA version eliminating 43.2% and the CFA version eliminating 43.4% of type checks at run time. The flow-sensitive 0CFA analysis peformed only slightly better, eliminating 59.2% of type checks at run time. Figure 12 compares the percent of type checks eliminated, on average, for the flow-insensitive sub-0CFA, flow-insensitive 0CFA, flow-sensitive sub-0CFA, flow-sensitive 0CFA. Figure 14 and figure 15 at the end of this paper give the percent of checks eliminated for each individual program.

Although the analysis does not require the order of evaluation of `let` bindings and function-call arguments to be specified, type information learned in one argument or binding might be useful for eliminating a type check in another argument or binding. In `(f (car x) (cdr x))`, for instance, a specified evaluation order would allow the implicit pair check to be eliminated from one of the two argument expressions. To determine the impact of fixing the order of operations, we tested with both left-to-right and right-to-left evaluation orders, and found that in both cases, the average number of type checks at run time improved by only a few percent, although the benefit is more significant in a few cases.

These results are encouraging, and we expect to be able to make additional improvements as we refine the implementation. The analysis currently treats all pairs and all vectors the same, although we could treat each occurrence of `cons` and `make-vector` in the source code as a separate element in the lattice analogously to the way we handle `lambda` expressions, and thus get more information about the contents of pairs and vectors.

5.4 Efficiency

Beyond the effectiveness of our analysis, we also verified its asymptotic behavior and measured its speed by counting the number of source tree nodes on input to the type-recovery pass and measuring the time it takes for the CFA algorithm to run. For this test, we used the R6RS benchmarks, as before, along with the various compilation units that comprise the Chez Scheme compiler. Figure 13 plots these times on a logarithmic scale along with linear (lower) and linear-log (upper) reference lines. The quantization of the numbers at the lower end of the graph results from timer granularity. The graph shows that the processing times trend between the linear and worst-case linear-log lines, as expected. The type recovery is also acceptably fast, handling 100,000 AST nodes (approximately 30,000 lines of code) in less than a second for the largest of the programs. Averaging over all of the programs, the implementation handles about 256,600 AST nodes (approximately 75,000 lines of code) per second.

6. Related work

6.1 CFA and CFA-based type recovery

Shivers (1990) uses an extension of 0CFA to perform type recovery. Instead of directly discovering type information about variables, he adds a level of indirection and discovers information about the *quantity* a variable contains. This approach allows information learned about one variable to be shared with its aliases but leads to potential correctness problems if multiple quantities flow to the same variable. Shivers addresses this by introducing a reflow semantics to correct for the problems caused by the indirection around quantities. We do not treat quantity information in our analysis, instead relying on a pass earlier in the compiler that performs copy propagation and aggressive inlining. This keeps our analysis relatively simple while still yielding some of the benefits of his quantities. Since it is based on 0CFA rather than sub-0CFA, Shivers's analysis is more precise though asymptotically more expensive.

Serrano (1995) argues that 0CFA is useful in functional language compilers by presenting two use cases: an analysis for reducing closure allocation and an analysis for reducing dynamic type tests. Serrano reports that the latter algorithm eliminates 65% of dynamic type tests. This differs from our results, which show that a 0CFA-based analysis eliminates only 43% of tests. The difference is likely attributable to the different set of benchmarks and to different strategies for inserting and counting type checks. As it is based on 0CFA, his analysis takes $O(n^3)$ time in the worst-case.

Heintze and McAllester (1997b) describe a linear time CFA. It is specifically targeted at typed languages and assumes bounds on the sizes of types. In quadratic time, it can either list up to a constant number of targets for all call sites, or list all targets for each call site. Mossin (1998) independently developed a similar quadratic analysis for explicitly typed programs based on *higher-order flow graphs*. Whereas these analyses are based on *inclusion*, Henglein's *simple closure analysis* (Henglein 1992b) computes a cruder approximation based on equality constraints, and can be solved in almost linear time via unification. None of these are flow-sensitive.

Our notion of sub-0CFA is close to that of Ashley and Dybvig (1998). They effectively use a more restrictive lattice than ours but provide a general framework through which more general lattices can be constructed. Their analysis achieves a limited form of flow sensitivity when the test of an `if` is a type predicate applied to a variable by creating new bindings for the variable in the *then* and *else* parts of the `if` whose abstract values are restricted by the test. They also describe a more general form of flow sensitivity that tracks variable assignments, but it does not gather observational information from nested conditionals, type-restricted primitives, or user-defined functions, and they do not make any claims about its asymptotic behavior.

6.2 Type recovery based on type inference

Soft typing (Cartwright and Fagan 1991) and more recently, gradual typing (Siek and Taha 2006) are designed to produce, through type inference, statically well-typed programs from dynamically typed programs by introducing run-time checks or casts. CFA-based type recovery can be seen as an alternative mechanism for accomplishing a similar effect. While soft typing and gradual type systems might reject some programs, our implementation never rejects programs, because type errors are semantically required to cause run-time exceptions.

Henglein (1992a) presents a fast $O(n\alpha(n))$ (almost linear time) tagging optimization algorithm for Scheme. The goal of the algorithm is to statically eliminate dynamic tagging and untagging operations, similar to our static elimination of dynamic type tests. Whereas untagging is related to type testing, it is subtly different. For example, based on what can flow to a conditional, Henglein will potentially optimize away an untagging operation of the test expression, whereas we will potentially optimize the conditional's branches based on the static knowledge gained from the test expression. A companion paper (Henglein 1994) treats the theory of *dynamic typing* in the form of a calculus with explicit type coercions and an equational theory.

The concept of *occurrence typing* developed in the context of Typed Scheme (Tobin-Hochstadt and Felleisen 2010), is closely related to the present analysis in that different occurrences of the same variable are typed differently depending on the control flow through type-testing predicates. The type system of Typed Scheme express types as formulas in a propositional logic that has some similarities to the lattice structure underlying our analysis.

6.3 Recent type-recovery applications

After two decades of research in type recovery, the topic is as relevant as ever with the success of dynamically typed languages such as JavaScript, Python, and Ruby.

Jensen et al. (2009) develop a type analysis for JavaScript. Their analysis is context-sensitive and incorporates both *recency abstraction* and *abstract garbage collection*. They focus however on precision over computational complexity. As a result, their analysis sometimes requires a few minutes to process JavaScript programs of only several hundred lines.

Vardoulakis and Shivers (2010) describe a *summarization*-based CFA with a degree of flow sensitivity. In addition to precise call-return matching, their analysis models precisely the top stack frame of arguments. Their focus is, however, more on precision than efficiency. The analysis has since been re-targeted to JavaScript in the form of DoctorJS (Mozilla Corporation 2011).

To type check dynamically typed programs, Guha et al. (2011) combine a type system and a flow analysis such that the latter can boost the precision of the former. Like our analysis their flow analysis is flow-sensitive and computes tag sets for each variable occurrence. Unlike our analysis, it is not interprocedural, relying instead on the type system at function boundaries. Furthermore it has a quadratic worst case time complexity.

6.4 Other related work

To prove soundness of the analysis, we use the concretization framework of abstract interpretation (Cousot and Cousot 1992). Cousot and Cousot (1979) originally used traces (paths) over a flow graph to prove soundness of classical data-flow analyses. Flow graphs were later generalized to transition systems (Cousot 1981), and paths were extended to traces thereof.

Wegman and Zadeck (1991) formulated fast constant propagation algorithms for a first-order imperative language. Their *conditional constant propagation* relates to our CFA in that they track reachability and may gain information from a test in a conditional. Wegman and Zadeck list elimination of run-time type checks in a LISP dialect as a possible use case of their approach. Whereas they consider multiple ways

to handle functions, including aliasing of pass-by-reference parameters, they do not consider how to handle first-class functions.

As an illustration of a general *property simulation* algorithm in ESP, Das et al. (2002) instantiate their general framework to a flow-sensitive constant-propagation algorithm. The resulting work-list algorithm is polynomial, however, as it involves invoking a theorem prover at each conditional expression for symbolic evaluation.

7. Conclusions and future work

This paper describes a flow-sensitive type-recovery algorithm based on sub-0CFA that runs in linear-log time. It justifies, on average, the removal of about 60% of run-time type checks in a standard set of benchmarks for the dynamically typed language Scheme. It handles, on average 75,000 lines of code in less then a second.

The implementation conservatively handles the unspecified evaluation order of arguments and bindings. Making evaluation-order decisions earlier in the compiler would allow the analysis to produce more precise information, particularly if the decisions were influenced by the needs of the analysis. Our experiments show that the typical benefit is likely to be minimal, but the benefit in some cases would be substantial.

Employing an extended lattice that differentiates pairs and vectors based on their allocation sites, as the analysis already does for functions, should also lead to more precise information. In a statically typed variant of the analysis, the lattice can also be refined to differentiate functions with different static types. Even in a dynamically typed language, functions can be grouped by arity.

Another avenue for further investigation is to supplement the current techniques with an efficient *must-alias analysis*, such that for two aliased variables `x` and `y`, information learned about `x` is reflected in `y`. The higher-order must-alias analysis by Jagannathan et al. (1998) would be a natural starting point for such an investigation.

Finally, we conjecture that the same techniques we have used to extend sub-0CFA with flow sensitivity can be applied more generally to kCFA with the addition of a single logarithmic factor to the asymptotic cost.

Acknowledgments

TBD.

References

Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. On finding lowest common ancestors in trees. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, STOC '73, pages 253–265, New York, NY, USA, 1973. ACM. doi: 10.1145/800125.804056.

Stephen Alstrup, Cyril Gavoille, Haim Kaplan, and Theis Rauhe. Nearest common ancestors: A survey and a new algorithm for a distributed environment. *Theory of Computing Systems*, 37: 441–456, 2004. ISSN 1432-4350. 10.1007/s00224-004-1155-5.

Michael J. Ashley and R. Kent Dybvig. A practical and flexible flow analysis for higher-order languages. *ACM Transactions on Programming Languages and Systems*, 20(4):845–868, 1998.

John Banning. An efficient way to find side effects of procedure calls and aliases of variables. In Rosen (1979), pages 29–41.

Robert Cartwright and Mike Fagan. Soft typing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292, 1991.

Swarat Chaudhuri. Subcubic algorithms for recursive state machines. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th Annual ACM Symposium on Principles of Programming Languages*, pages 159–169, San Francisco, California, January 2008.

William D. Clinger. Description of benchmarks, 2008. URL `http://www.larcenists.org/benchmarksAboutR6.html`.

Patrick Cousot. Semantic foundations of program analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, 1981.

Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In Rosen (1979), pages 269–282.

Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: Path-sensitive program verification in polynomial time. In Laurie J. Hendren, editor, *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, pages 57–68, Berlin, June 2002.

R. Kent Dybvig. *Chez Scheme Version 8 User's Guide*. Cadence Research Systems, 2010.

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing local control and state using flow analysis. In Gilles Barthe, editor, *Programming Languages and Systems, 20th European Symposium on Programming, ESOP 2011*, volume 6602 of *Lecture Notes in Computer Science*, pages 256–275, Saarbrücken, Germany, Mar-Apr 2011. Springer-Verlag.

Nevin Heintze and David McAllester. On the complexity of set-based analysis. In Mads Tofte, editor, *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, pages 150–163, Amsterdam, The Netherlands, June 1997a.

Nevin Heintze and David McAllester. Linear-time subtransitive control flow analysis. In Ron K. Cytron, editor, *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Languages Design and Implementation*, pages 261–272, Las Vegas, Nevada, June 1997b.

Fritz Henglein. Global tagging optimization by type inference. In William Clinger, editor, *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. V, No. 1, pages 205–215, San Francisco, California, June 1992a.

Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, 1994.

Fritz Henglein. Simple closure analysis. Technical Report Semantics Report D-193, DIKU, Computer Science Department, University of Copenhagen, 1992b.

Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In Peter Lee, editor, *Proceedings of the 22nd Annual ACM Symposium on Principles of Programming Languages*, pages 393–407, San Francisco, California, January 1995.

Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. Single and loving it: must-alias analysis for higher-order languages. In Luca Cardelli, editor, *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages*, pages 329–341, San Diego, California, January 1998.

Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In Jens Palsberg and Zhendong Su, editors, *Static Analysis, 16th International Symposium, SAS 2009*, volume 5673 of *Lecture Notes in Computer Science*, pages 238–255, Los Angeles, CA, USA, August 2009. Springer-Verlag.

David Melski and Thomas Reps. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science*, 248(1-2):29–98, 2000.

Jan Midtgaard and David Van Horn. Subcubic control flow analysis algorithms. Computer Science Research Report 125, Roskilde University, Roskilde, Denmark, May 2009. Revised version to appear in Higher-Order and Symbolic Computation.

Torben Æ. Mogensen. Glossary for partial evaluation and related topics. *Higher-Order and Symbolic Computation*, 13(4):355–368, 2000.

Christian Mossin. Higher-order value flow graphs. *Nordic Journal of Computing*, 5(3):214–234, 1998.

Mozilla Corporation. Doctor JS, 2011. `http://doctorjs.org/`.

Eugene W. Myers. Efficient applicative data types. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 66–75, 1984.

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33:668–676, June 1990. ISSN 0001-0782. doi: 10.1145/78973.78977. URL `http://doi.acm.org/10.1145/78973.78977`.

Barry K. Rosen, editor. *Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages*, San Antonio, Texas, January 1979.

Manuel Serrano. Control flow analysis: a functional languages compilation paradigm. In *Proceedings of the 1995 ACM Symposium on Applied Computing*, pages 118–122, Nashville, Tennessee, February 1995.

Olin Shivers. Control flow analysis in Scheme. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, volume 23, pages 164–174, July 1988.

Olin Shivers. Data-flow analysis and type recovery in scheme. Technical Report CMU-CS-90-115, CMU School of Computer Science, Pittsburgh, PA, March 1990.

Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, September 2006.

Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In Paul Hudak and Stephanie Weirich, editors, *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP'10)*, pages 117–128, Baltimore, Maryland, Sep 2010.

Dimitrios Vardoulakis and Olin Shivers. CFA2: a context-free approach to control-flow analysis. In Andrew D. Gordon, editor, *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010*, volume 6012 of *Lecture Notes in Computer Science*, pages 570–589, Paphos, Cyprus, March 2010. Springer-Verlag.

Mark N. Wegman and Kenneth F. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13:181–210, 1991.

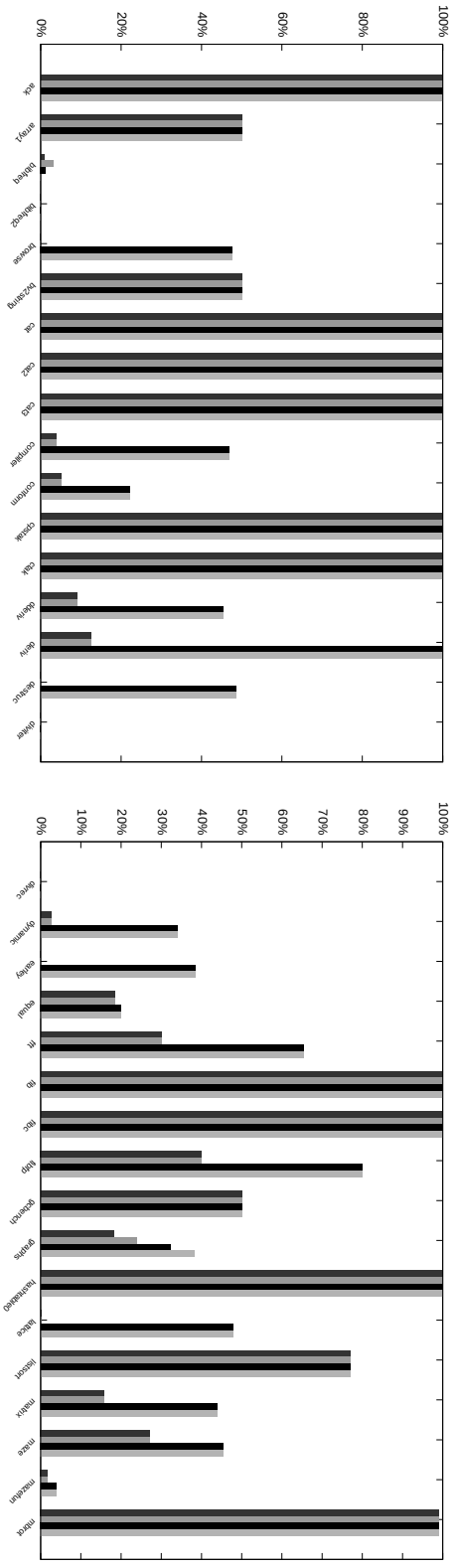


Figure 14: Percent of type checks removed (1/2)

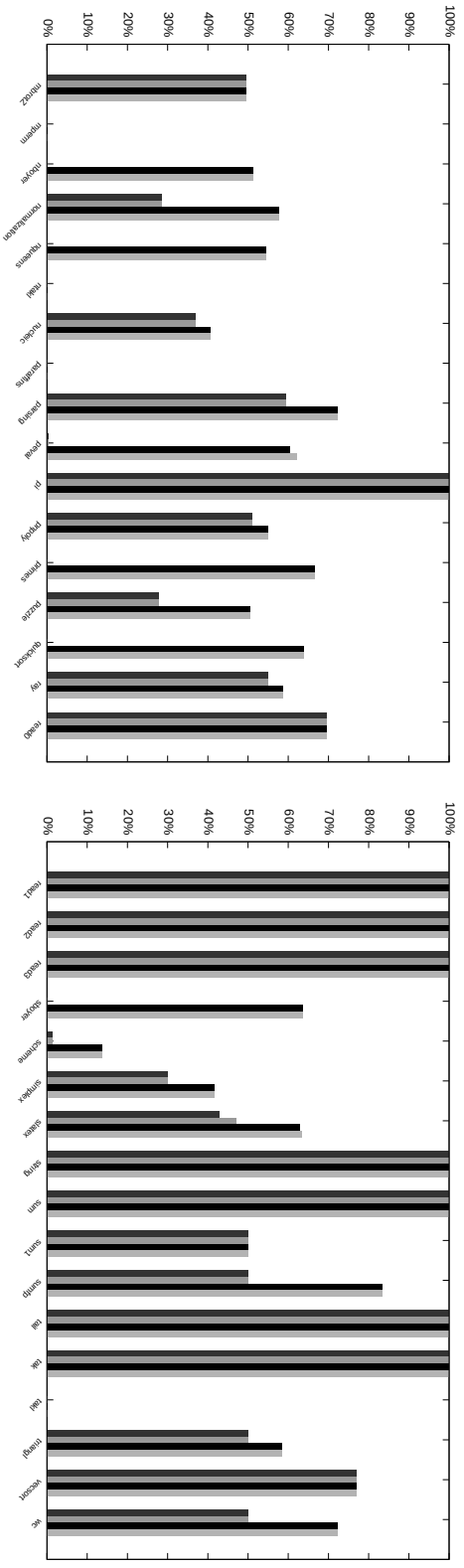


Figure 15: Percent of type checks removed (2/2)