Flow-Sensitive Type Recovery in Linear-Log Time *

Michael D. Adams
Indiana University

Andrew W. Keep
Indiana University

Jan Midtgaard
Aarhus University

Matthew Might
University of Utah

Arun Chauhan
Indiana University

R. Kent Dybvig
Indiana University

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. However, traditional control-flow analysis (CFA) is not ideal for this task as it ignores flow-sensitive information that can be gained from dynamic type predicates, such as JavaScript's `instanceof` and 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 implemented as an experimental optimization for 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. The algorithm processes on average over 100,000 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 type checks at run time. This cost can be reduced via type-recovery analysis [Shivers 1991], which attempts to discover variable and expression types at compile time and thereby justify the elimination of run-time type checks.

Since these 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 as the `read` function can return any type of value.

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

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

Because it does not take evaluation order into account, a flow-insensitive analysis treats all references the same. On the other hand, a flow-sensitive analysis[1] can determine that `(car x)` is reached in the first expression only after passing the pair check in `(cdr x)` and in the second expression only when the explicit `pair?` check succeeds. Thus the implicit pair check in `car` can be eliminated in both expressions.

In this paper, we present a flow-sensitive, CFA-based type-recovery algorithm that runs in linear-log time. Be-

* This research was facilitated in part by a National Physical Science Consortium Fellowship and by stipend support from the National Security Agency.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.
Copyright © 2011 ACM 978-1-4503-0940-0/11/10. . . $10.00

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

cause the analysis is intended to justify type recovery in a fast production compiler [Dybvig 2010], it is based on sub-0CFA [Ashley and Dybvig 1998], a linear-time, flow-insensitive variant of 0CFA [Shivers 1988]. We use a novel combination of data structures and algorithms to add flow sensitivity to sub-0CFA at the cost of only an additional logarithmic factor.

The analysis has been implemented as an experimental optimization for the commercial Chez Scheme compiler, where it has proven to be effective and justifies eliminating about 60% of run-time type checks. The algorithm has also proven to be fast, processing over 100,000 lines of code per second on average. Furthermore, since it runs in $O(n \log n)$ time, it scales well to large input programs.

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

This section reviews two relevant forms of control-flow analysis, Shivers's 0CFA and Ashley's sub-0CFA. It also discusses their implementations in terms of flow graphs, how top and escaped values are handled, and the representation of non-function types. Readers familiar with control-flow analysis may 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 explicit representation of contexts in our formulation differs from more traditional presentations [Nielson et al. 1999]. It is used in the rest of this paper as we extend and optimize the analysis. The operational semantics of this language is standard and is omitted.

The analysis stores a reachability flag, $\llbracket e \rrbracket_{in}$, for each subexpression of the program being analyzed. The flag is $\top$ if the expression is reachable and $\bot$ otherwise.

In addition, for each expression a flow variable $\llbracket e \rrbracket_{out}$ ranging over $\widehat{Val}$ records the abstract value that flows from the expression, i.e., a subset of the lambda terms that may be returned by the expression. For example, the LAMBDA rule says that if $\lambda x.e$ is reachable, the result of that expression includes an abstract value representing the lambda.

For $\widehat{Val}$, the $\sqsupseteq$ relation is the usual partial order on power sets. For $Bool$, it is the usual ordering where $\top \sqsupseteq \bot$.

We implicitly label all subexpressions and uniquely alpha-rename all variables before the analysis starts. Thus we distinguish duplicate expressions and variable names.

The CALL_{mid} and CALL_{fun} rules use $\mathbb{K}(e)$, which returns the source contextof e. These contexts are single-layer contexts instead of the more usual multilayer contexts, but for specifying the constraint rules, only a single layer is needed. When multilayer contexts are needed, we represent them by the juxtaposition of contexts.

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

$$
\frac{\llbracket \lambda x.e \rrbracket_{in} \sqsupseteq \top}{\llbracket \lambda x.e \rrbracket_{out} \sqsupseteq \{\lambda x.e\}}\ \textsc{Lambda}
\qquad
\frac{\llbracket e_0\ e_1 \rrbracket_{in} \sqsupseteq \hat{r}}{\llbracket e_0 \rrbracket_{in} \sqsupseteq \hat{r}}\ \textsc{Call}_{in}
$$

$$
\frac{\llbracket e_0 \rrbracket_{out} \sqsupseteq \{\hat{v}_0\} \qquad \mathbb{K}(e_0) = (\Box\ e_1)}{\llbracket e_1 \rrbracket_{in} \sqsupseteq \top}\ \textsc{Call}_{mid}
$$

$$
\frac{\begin{array}{c}\llbracket e_0 \rrbracket_{out} \sqsupseteq \{\lambda x.e_\lambda\} \\ \llbracket e_1 \rrbracket_{out} \sqsupseteq \{\hat{v}_1\} \qquad \mathbb{K}(e_1) = (e_0\ \Box)\end{array}}{\llbracket e_\lambda \rrbracket_{in} \sqsupseteq \top \qquad \llbracket x \rrbracket_{out} \sqsupseteq \{\hat{v}_1\}}\ \textsc{Call}_{fun}
$$

$$
\frac{\begin{array}{c}\llbracket e_0 \rrbracket_{out} \sqsupseteq \{\lambda x.e_\lambda\} \\ \llbracket e_1 \rrbracket_{out} \sqsupseteq \{\hat{v}_1\} \qquad \llbracket e_\lambda \rrbracket_{out} \sqsupseteq \{\hat{v}\}\end{array}}{\llbracket e_0\ e_1 \rrbracket_{out} \sqsupseteq \{\hat{v}\}}\ \textsc{Call}_{out}
$$

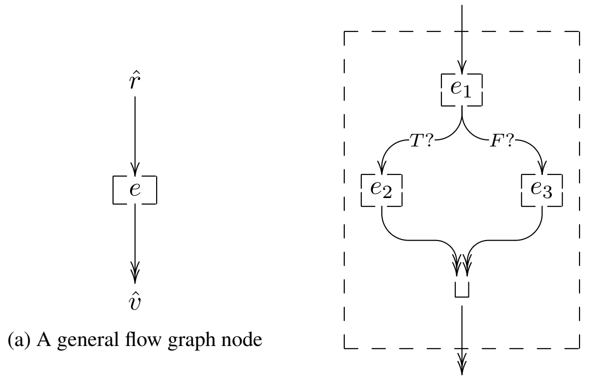
Figure 1: 0CFA with reachability

For example, the CALL_{fun} rule says that if a lambda flows to a subexpression that is contextually located inside an application, i.e., $\mathbb{K}(e_1) = (e_0\ \Box)$, and a value flows to the operand of the expression, e_1, then the body of the invoked lambda is reachable and the actual argument flows to the formal parameter.

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

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 [Heintze and McAllester 1997a; Jagannathan and Weeks 1995] with graph nodes denoting the flow variables $\llbracket e \rrbracket_{in}$ and $\llbracket e \rrbracket_{out}$ and directed edges denoting the flow of abstract values from one node to another. For 0CFA augmented 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.



(a) A general flow graph node

(b) A composite flow graph

Figure 2: Flow graphs in 0CFA

For example, in an analysis for a language with conditionals, the flow graph for the expression `(if` e_1 e_2 e_3`)` contains nodes for reachability and result values 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 and only if the `if` is reachable. Thus there is an edge from the input of the `if` to e_1. Likewise e_2 and e_3 are reachable if and only 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 $F?$ which 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. A node listed on the work list has a potentially stale output that needs to be updated based on new inputs to that node. Initially all nodes are on the work list. One by one, nodes are removed from the work list and a new value for their outputs calculated based on the values of their input edges. If the value of the output changes, then any node connected by an outgoing edge of the current node is placed back on the work list thus effectively marking it as potentially stale. The algorithm continues selecting nodes from the work list and recalculating stale nodes until the graph converges and no stale nodes remain.

A crucial property is that, under appropriate conditions, flow graphs quickly converge to a solution. If (1) the values that flow through a graph are members of a finite-height lattice, L, (2) the output value of each node moves only monotonically up the lattice, and (3) the output value 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 edges between functions and call sites since not all calls are known, so the algorithm adds new edges 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 added but 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. Slightly faster techniques are known for computing 0CFA but they still take $O(n^3/\log n)$ time [Chaudhuri 2008; Melski and Reps 2000; Midtgaard and Van Horn 2009].

2.3 Top and escaped functions in CFA

If a CFA is operating on a program that contains free variables such as variables imported from libraries outside the scope of the analysis, the analysis does not know anything about the values of those variables. This is handled by adding a top, $\top$, element to the lattice. This $\top$ denotes an unknown function [Shivers 1988] and is used for the value of free variables. It represents not only any function from outside the scope of the analysis but also includes any function inside the scope of the analysis. This is to say, it subsumes all other functions in $\widehat{Lam}$.

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

Top values and escaped functions can cause more top values and escaped functions. First, if the function position of a function call is $\top$, the return value of the function call 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 $\top$, the result is $\top$, and since $\top$ 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. Thus 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 $\top$. This conservative approximation of 0CFA's power-set lattice has a constant height and is shown in figure 3. Whenever two or more different functions are joined by $\sqcup$, the result is $\top$. 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 $\top$ and thus add at most a linear number of edges to the graph.

This approach lets more functions escape than in 0CFA, but it is not as bad as it might seem. Flowing to two different places does *not* cause a function to escape. 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, but h is not affected.

```
(let ([f (lambda (x) $e_1$)]
      [g (lambda (y) $e_2$)]
      [h (lambda (z) $e_3$)])
  (let ([fg (if (read) f g)])
    (f (g (fg (h (h $e_4$)))))))
```

In the general case, Ashley and Dybvig [1998] define sub-0CFA by a projection or widening operator. When this operator restricts sets of values to either singletons or $\top$, it is equivalent to what we do here. 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 projection 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. Thus we add a fixed set of primitive types, e.g., *INT*, *PAIR*, etc., to the $\widehat{Val}$ lattice. Because of the lattice flattening in sub-0CFA, however, 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 powerset of non-function types. Nevertheless, we notationally treat abstract values as sets. For example, $\{\mathit{INT}, \mathit{PAIR}, \lambda x.e\}$ is understood to mean $\langle\{\mathit{INT}, \mathit{PAIR}\}, \{\lambda x.e\}\rangle$.

3. Traditional flow-sensitivity

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

```
(let ([x (cons $e_1$ $e_2$)]) (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 expressions, they are treated as $\top$.

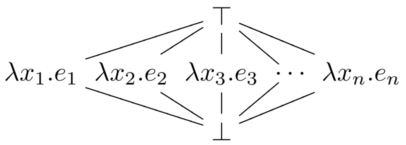


Figure 3: Sub-0CFA lattice of functions

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 when car is called. In the third expression, we can also deduce that x must be a pair when car is called, since the implicit pair check in cdr guarantees that it returns only if its argument is a pair.

We call information *constructive* when it is learned from operations that are constructing values as with the first expression. We call information *observational* when it is learned from operations that are observing values as with the second and third expressions.

Observational information is *restrictive*, since it restricts the type of a variable or value, as in the restriction of x to the pair type in those expressions. If observational information restricts a type to two or more disjoint types, the type is $\bot$. In general, wherever a $\bot$ type occurs, the following code is unreachable, i.e., 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 observation.

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. The information about a variable is the same at all references to the variable. To be flow-sensitive we adjust the abstract values of variables as we trace the flow of the program. 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 as argument. The 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 that 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 flow-insensitive 0CFA. It includes sequencing by threading the environment through the execution 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 entering the expression. The other tracks the type of variables exiting the expression. At each function call, arguments are restricted to only those abstract values that are compatible with the particular function returning. After (car x) for example, x is restricted to pairs. Both the entering and exiting environments are treated as reduced abstract domains [Cousot and Cousot 1979] and equate abstract elements with the same meaning (concretization). Hence, if any component of an environment is $\bot$, then all components of the environment are forced to be $\bot$. For example, if x is known to be an integer, then x is $\bot$ after (car x). This causes all components of the exiting environment to be $\bot$. In addition, as part of the reduced abstract domain, the return value of (car x) is $\bot$. 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 when they return. However, it fails to handle predicates like pair?. The fact that a call to pair? returns says nothing about the argument. The information is conditional, and it is whether it returns a true or false value that tells us whether the argument is a pair.

3.2 Flow-sensitivity for conditional observers

To handle conditional or predicated observers, we generalize the environments flowing through the program. For the exit of an expression, we store one environment for when the expression returns a true value and another for when the expression returns a false value. The environment 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 values for when e returns true values and the other for 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 the IF_{mid} rule. 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 abstract 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_0 and e_1 are collected. Next, RET examines any functions, $\hat{f}$, flowing out of e_0 and returns three abstract values. One 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 determines 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 has one abstract value for its argument for when it returns true and another for when it returns false. For example, with pair?, the RET function returns the abstract value for pairs in the true case and the abstract value for non-pairs in the false case. Unconditional observers like car have 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-insensitive CFA these constraint rules can be implemented by 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 a true and a false environment with each exit edge of an expression. This is depicted in figure 5a. Lines with single-headed arrows represent the flow of single values and lines with double-headed arrows 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 contain environments. The lat-

$$\begin{array}{rl}
\text{Expressions:} & e \in \mathit{Exp} = x \mid \lambda x.e \mid e\ e \mid \mathit{if}\ e\ e\ e \mid e;e \mid c \\
\text{Contexts:} & C \in \mathit{Ctxt} = \Box \mid (\Box\ e_1) \mid (e_0\ \Box) \mid (\lambda x.\Box) \mid (\Box; e_1) \mid (e_0; \Box) \mid (\mathit{if}\ \Box\ e_1\ e_2) \mid (\mathit{if}\ e_0\ \Box\ e_2) \mid (\mathit{if}\ e_0\ e_1\ \Box) \\
\text{Signatures:} & \llbracket e \rrbracket_{in} \in \mathit{Bool} \times \widehat{\mathit{Env}} \qquad \llbracket e \rrbracket_{out} \in \widehat{\mathit{Val}} \times \widehat{\mathit{Env}} \times \widehat{\mathit{Env}} \qquad \mathbb{K} \in \mathit{Exp} \to \mathit{Ctxt} \\
& \hat{r} \in \mathit{Bool} = \{\bot, \top\} \quad \hat{\rho} \in \widehat{\mathit{Env}} = \mathit{Var} \to \widehat{\mathit{Val}} \quad \hat{v} \in \widehat{\mathit{Val}} = \widehat{\mathit{Fun}} \times \wp(\widehat{\mathit{Tag}}) \quad \hat{f} \in \widehat{\mathit{Fun}} = \wp(\widehat{\mathit{Lam}} + \widehat{\mathit{Prim}}) \\
& \hat{t} \in \widehat{\mathit{Tag}} = \{\mathit{FALSE}, \mathit{TRUE}, \mathit{INT}, \mathit{FLOAT}, \mathit{PAIR}, \ldots\} \qquad \top_t = \widehat{\mathit{Val}} \backslash \{\mathit{FALSE}\} \qquad \top_f = \{\mathit{FALSE}\} \\
& \widehat{\mathit{Lam}} = \{\lambda x_1.e_1, \lambda x_2.e_2, \lambda x_3.e_3, \ldots\} \qquad o \in \widehat{\mathit{Prim}} = \{\texttt{pair?}, \texttt{car}, \texttt{cdr}, \ldots\} \\
& \mathit{ABS}(\texttt{\#f}) = \mathit{FALSE} \qquad \mathit{ABS}(\texttt{\#t}) = \mathit{TRUE} \qquad \mathit{ABS}(n) = \mathit{INT} \qquad \ldots
\end{array}$$

$$\mathit{RET}(\hat{f}) = \bigsqcup_{f \in \hat{f}} \mathit{RET}(f) \qquad \begin{array}{c} \mathit{RET}(\lambda x.e) = \langle \hat{v},\, \hat{\rho}_t(x),\, \hat{\rho}_f(x) \rangle \text{ where } \langle \hat{v},\, \hat{\rho}_t,\, \hat{\rho}_f \rangle = \llbracket e \rrbracket_{out} \\ \mathit{RET}(\texttt{car}) = \langle \top,\, \{\mathit{PAIR}\},\, \{\mathit{PAIR}\} \rangle \\ \mathit{RET}(\texttt{pair?}) = \langle \{\mathit{FALSE}, \mathit{TRUE}\},\, \{\mathit{PAIR}\},\, \top \backslash \{\mathit{PAIR}\} \rangle \\ \ldots \end{array}$$

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

$$\frac{\llbracket c \rrbracket_{in} \sqsupseteq \langle \top,\, \hat{\rho} \rangle}{\llbracket c \rrbracket_{out} \sqsupseteq \langle \mathit{ABS}(c),\, \hat{\rho},\, \hat{\rho} \rangle}\ \textsc{Const} \qquad \frac{\llbracket \lambda x.e \rrbracket_{in} \sqsupseteq \langle \top,\, \hat{\rho} \rangle}{\llbracket \lambda x.e \rrbracket_{out} \sqsupseteq \langle \lambda x.e,\, \hat{\rho},\, \hat{\rho} \rangle}\ \textsc{Lambda}$$

$$\frac{\llbracket x \rrbracket_{in} \sqsupseteq \langle \top,\, \hat{\rho} \rangle}{\llbracket x \rrbracket_{out} \sqsupseteq \langle \hat{\rho}(x),\, \hat{\rho}[x \mapsto \hat{\rho}(x) \sqcap \top_t],\, \hat{\rho}[x \mapsto \hat{\rho}(x) \sqcap \top_f] \rangle}\ \textsc{Var} \qquad \frac{\llbracket e_0\ e_1 \rrbracket_{in} \sqsupseteq \langle \hat{r},\, \hat{\rho} \rangle}{\llbracket e_0 \rrbracket_{in} \sqsupseteq \langle \hat{r},\, \hat{\rho} \rangle}\ \textsc{Call}_{in}$$

$$\frac{\llbracket e_0 \rrbracket_{out} \sqsupseteq \langle \hat{v}_0,\, \hat{\rho}_t,\, \hat{\rho}_f \rangle \qquad \mathbb{K}(e_0) = (\Box\ e_1)}{\llbracket e_1 \rrbracket_{in} \sqsupseteq \langle \top,\, \hat{\rho}_t \sqcup \hat{\rho}_f \rangle}\ \textsc{Call}_{mid}$$

$$\frac{\llbracket e_0 \rrbracket_{out} \sqsupseteq \langle \{\lambda x.e_\lambda\},\, \hat{\rho}_t^{e_0},\, \hat{\rho}_f^{e_0} \rangle \qquad \llbracket \lambda x.e_\lambda \rrbracket_{in} \sqsupseteq \langle \top,\, \hat{\rho}_\lambda \rangle \qquad \llbracket e_1 \rrbracket_{out} \sqsupseteq \langle \hat{v}_1,\, \hat{\rho}_t^{e_1},\, \hat{\rho}_f^{e_1} \rangle \qquad \mathbb{K}(e_1) = (e_0\ \Box)}{\llbracket e_\lambda \rrbracket_{in} \sqsupseteq \langle \top,\, \hat{\rho}_\lambda[x \mapsto \hat{v}_1] \rangle}\ \textsc{Call}_{fun}$$

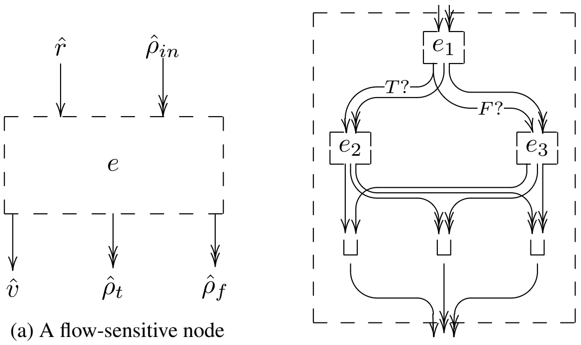
$$\frac{\llbracket e_0 \rrbracket_{out} \sqsupseteq \langle \hat{f},\, \hat{\rho}_t^{e_0},\, \hat{\rho}_f^{e_0} \rangle \quad \llbracket e_1 \rrbracket_{out} \sqsupseteq \langle \hat{v}_1,\, \hat{\rho}_t^{e_1},\, \hat{\rho}_f^{e_1} \rangle \quad \langle \hat{v},\, \hat{v}_t,\, \hat{v}_f \rangle = \mathit{RET}(\hat{f}) \quad \forall i \in \{t, f\}. \hat{\rho}'_i = \mathit{ARG}(\hat{\rho}_t^{e_1} \sqcup \hat{\rho}_f^{e_1}, e_1, \hat{v}_1 \sqcap \hat{v}_i)}{\llbracket e_0\ e_1 \rrbracket_{out} \sqsupseteq \langle \hat{v},\, \hat{\rho}'_t,\, \hat{\rho}'_f \rangle}\ \textsc{Call}_{out}$$

$$\frac{\llbracket e_0; e_1 \rrbracket_{in} \sqsupseteq \langle \hat{r},\, \hat{\rho} \rangle}{\llbracket e_0 \rrbracket_{in} \sqsupseteq \langle \hat{r},\, \hat{\rho} \rangle}\ \textsc{Seq}_{in} \qquad \frac{\llbracket e_0 \rrbracket_{out} \sqsupseteq \langle \hat{v}_0,\, \hat{\rho}_t,\, \hat{\rho}_f \rangle \qquad \mathbb{K}(e_0) = (\Box; e_1)}{\llbracket e_1 \rrbracket_{in} \sqsupseteq \langle \top,\, \hat{\rho}_t \sqcup \hat{\rho}_f \rangle}\ \textsc{Seq}_{mid} \qquad \frac{}{\llbracket e_0; e_1 \rrbracket_{out} \sqsupseteq \llbracket e_1 \rrbracket_{out}}\ \textsc{Seq}_{out}$$

$$\frac{\llbracket \mathit{if}\ e_0\ e_1\ e_2 \rrbracket_{in} \sqsupseteq \langle \hat{r},\, \hat{\rho} \rangle}{\llbracket e_0 \rrbracket_{in} \sqsupseteq \langle \hat{r},\, \hat{\rho} \rangle}\ \textsc{If}_{in} \qquad \frac{\llbracket e_0 \rrbracket_{out} \sqsupseteq \langle \hat{v}_0,\, \hat{\rho}_t,\, \hat{\rho}_f \rangle \qquad \mathbb{K}(e_0) = (\mathit{if}\ \Box\ e_1\ e_2)}{\llbracket e_1 \rrbracket_{in} \sqsupseteq \langle \top,\, \hat{\rho}_t \rangle \qquad \llbracket e_2 \rrbracket_{in} \sqsupseteq \langle \top,\, \hat{\rho}_f \rangle}\ \textsc{If}_{mid}$$

$$\frac{}{\llbracket \mathit{if}\ e_0\ e_1\ e_2 \rrbracket_{out} \sqsupseteq \llbracket e_1 \rrbracket_{out} \sqcup \llbracket e_2 \rrbracket_{out}}\ \textsc{If}_{out}$$

Figure 4: Analysis constraint rules



(a) A flow-sensitive node

(b) A composite flow graph

Figure 5: Flow graphs for flow-sensitive 0CFA

tice of an environment is the Cartesian product of the lattice for each variable, so the lattice height of an environment is linear 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 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 implement flow sensitivity efficiently, we adopt a similar approach. Instead of flowing abstract values from a variable binding site to each reference, however, we flow values from one occurrence of the variable to the next, following the control flow of the program. The abstract value of the variable is adjusted as appropriate at each occurrence.

When moving from one occurrence to the next, the true and false values for each variable may join or swap with each other. For example, consider how abstract values for x flow from (pair? x) to (car x) in the following expression. It is of the sort that may arise via the expansion of boolean connectives such as and, or, and not. Assume x is not referenced in e_1 or e_2.

```
(if (if (pair? x) e₁ e₂) (car x) e₃)
```

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. If, however, 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 expressions e_0, e_1, or e_2.

```
(let ([x (read)])
  (if (if e₀ (begin (car x) e₁) e₂)
    (cdr x) e₃))
```

After (car x), it is known that x is 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 (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.

Dealing with how the abstract values of variables change as they flow through the program but still taking only $O(n \log n)$ time is the primary technical challenge of this analysis. The remainder of this section shows how to do this. The result is an analysis that takes only linear-log time and produces the same results as the analysis in section 3.

First, section 4.1 defines a skipping function $\mathcal{V}_{C,e}$ that allows values to move from one point in the program to another without threading through each intervening expression, while accounting for changes that happen as they flow through each expression. Next, section 4.2 explains how to determine where to use the skipping functions versus the constraint rules. Then section 4.3 describes the data structures used to cache the skipping functions efficiently so that each one can be computed or updated in logarithmic time. Since the algorithm in section 4.2 ensures that only a linear number of skipping functions are used, the entire analysis then takes only linear-log time. Finally, section 4.4 puts all of these together into a linear-log time algorithm that computes results identical to those of the less efficient 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 out of both if expressions and into (car 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, 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 x is a variable not mentioned in e then*

$$\langle \hat{\rho}_t(x),\ \hat{\rho}_f(x)\rangle = \langle \mathcal{G}_t(e,\ \hat{\rho}_{in}(x)),\ \mathcal{G}_f(e,\ \hat{\rho}_{in}(x))\rangle$$
$$\textit{where } \langle \hat{r},\ \hat{\rho}_{in}\rangle = [\![e]\!]_{in} \textit{ and } \langle \hat{v},\ \hat{\rho}_t,\ \hat{\rho}_f\rangle = [\![e]\!]_{out}$$
$$\textit{and } \mathcal{G}_t(e,\ u) \textit{ and } \mathcal{G}_f(e,\ u) \textit{ are as defined in figure 6.}$$

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

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

Figure 6: True and false expression guards

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 start of e_1 and e_2. The $\mathcal{G}_t(e,\ u)$ and $\mathcal{G}_f(e,\ u)$ functions act as guards and return the abstract value u if and only if $[\![e]\!]_{out}$ contains true or false values respectively.

This lemma deals only with the flow of abstract values from the entry of an expression to its exit. As seen in the examples, however, we are also interested in how abstract values flow from an expression through its surrounding context. In the preceding examples, abstract values flow 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 $\mathcal{V}_{C,e}$ defined 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 x is a variable not mentioned in the single-layer context C then*

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

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

There are a corresponding context-skipping function and lemma for values entering a context, but they are omitted here as they are straightforward and are equivalent to a simple reachability check.

For example, consider `(if (pair? x)` e_1 e_2`)` and the resulting $\mathcal{V}_{C,e}$. The context of `(pair? x)` is ($\mathit{if}\ \square\ e_1\ e_2$). Thus, if e_1 returns only true values and e_2 returns only false values, the skipping function, $\mathcal{V}_{C,e}$, does not change the values in the environment as they flow from `(pair? x)`. On the other hand, if e_1 returns only false values and e_2 returns only true values, the skipping function swaps the values of the true and false environments. Other possibilities arise depending on the values returned by e_1 and e_2.

$$\begin{aligned}
\mathcal{V}_{C,e} \langle \hat{v}_t,\ \hat{v}_f,\ \hat{v}_{in} \rangle &= \langle \mathcal{G}_t(C[e],\ \hat{v}'_t),\ \mathcal{G}_f(C[e],\ \hat{v}'_f),\ \hat{v}'_{in} \rangle \\
\text{where } \langle \hat{v}'_t,\ \hat{v}'_f,\ \hat{v}'_{in} \rangle &= \mathcal{V}'_C \langle \hat{v}_t,\ \hat{v}_f,\ \hat{v}_{in} \rangle
\end{aligned}$$

$$\begin{aligned}
\mathcal{V}'_{(\mathit{if}\ \square\ e_2\ e_3)} \langle \hat{v}_t,\ \hat{v}_f,\ \hat{v}_{in} \rangle &= \langle \hat{v}'_t,\ \hat{v}'_f,\ \hat{v}_{in} \rangle \\
\text{where } \hat{v}'_t &= \mathcal{G}_t(e_2,\ \hat{v}_t) \sqcup \mathcal{G}_t(e_3,\ \hat{v}_f) \\
\hat{v}'_f &= \mathcal{G}_f(e_2,\ \hat{v}_t) \sqcup \mathcal{G}_f(e_3,\ \hat{v}_f) \\
\mathcal{V}'_{(\mathit{if}\ e_1\ \square\ e_3)} \langle \hat{v}_t,\ \hat{v}_f,\ \hat{v}_{in} \rangle &= \langle \hat{v}'_t,\ \hat{v}'_f,\ \hat{v}_{in} \rangle \\
\text{where } \hat{v}'_t &= \hat{v}_t \sqcup \mathcal{G}_t(e_3,\ \hat{v}_{in}) \\
\hat{v}'_f &= \hat{v}_f \sqcup \mathcal{G}_f(e_3,\ \hat{v}_{in}) \\
\mathcal{V}'_{(\mathit{if}\ e_1\ e_2\ \square)} \langle \hat{v}_t,\ \hat{v}_f,\ \hat{v}_{in} \rangle &= \langle \hat{v}'_t,\ \hat{v}'_f,\ \hat{v}_{in} \rangle \\
\text{where } \hat{v}'_t &= \hat{v}_t \sqcup \mathcal{G}_t(e_2,\ \hat{v}_{in}) \\
\hat{v}'_f &= \hat{v}_f \sqcup \mathcal{G}_f(e_2,\ \hat{v}_{in}) \\
\mathcal{V}'_{(\lambda x.\square)} \langle \hat{v}_t,\ \hat{v}_f,\ \hat{v}_{in} \rangle &= \langle \hat{v}_{in},\ \hat{v}_{in},\ \hat{v}_{in} \rangle \\
\mathcal{V}'_{(e_2\ \square)} \langle \hat{v}_t,\ \hat{v}_f,\ \hat{v}_{in} \rangle &= \langle \hat{v}_t \sqcup \hat{v}_f,\ \hat{v}_t \sqcup \hat{v}_f,\ \hat{v}_{in} \rangle \\
\mathcal{V}'_{(\square\ e_2)} \langle \hat{v}_t,\ \hat{v}_f,\ \hat{v}_{in} \rangle &= \langle \hat{v}_t \sqcup \hat{v}_f,\ \hat{v}_t \sqcup \hat{v}_f,\ \hat{v}_{in} \rangle \\
\mathcal{V}'_{(\square ; e_2)} \langle \hat{v}_t,\ \hat{v}_f,\ \hat{v}_{in} \rangle &= \langle \hat{v}_t \sqcup \hat{v}_f,\ \hat{v}_t \sqcup \hat{v}_f,\ \hat{v}_{in} \rangle \\
\mathcal{V}'_{(e_1 ; \square)} \langle \hat{v}_t,\ \hat{v}_f,\ \hat{v}_{in} \rangle &= \langle \hat{v}_t,\ \hat{v}_f,\ \hat{v}_{in} \rangle
\end{aligned}$$

Figure 7: Context skipping function

While lemma 4.2 handles only single layer contexts, the following theorem generalizes this to multilayer contexts.[2]

Theorem 4.3 (Multilayer Context Skipping)**.** *If 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*

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

Proof. By induction on C and lemma 4.2. □

Note that even for multilayer contexts the universe of possible $\mathcal{V}_{C,e}$ is finite and small. Any particular $\mathcal{V}_{C,e}$ can be represented in a canonical form of constant size.

Theorem 4.4 (Canonical Skipping Functions)**.** *There exist $T, F \subseteq \{\hat{v}_t, \hat{v}_f, \hat{v}_{in}\}$ for any C, e, $[\![e]\!]_{out}$ and $[\![C[e]]\!]_{out}$ such that*

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

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 of the dashed edges that lead to the two join ($\sqcup$) nodes. Functions of this form have compact, constant-size representations, are closed under composition,

[2] This theorem is the reason why the tuple returned by $\mathcal{V}_{C,e}$ has a third component even though it is unused in lemma 4.2.

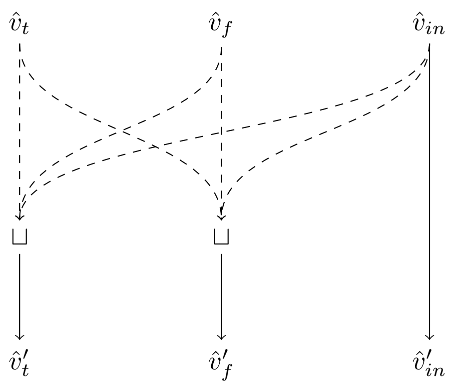


Figure 8: Graph form of canonical skipping functions

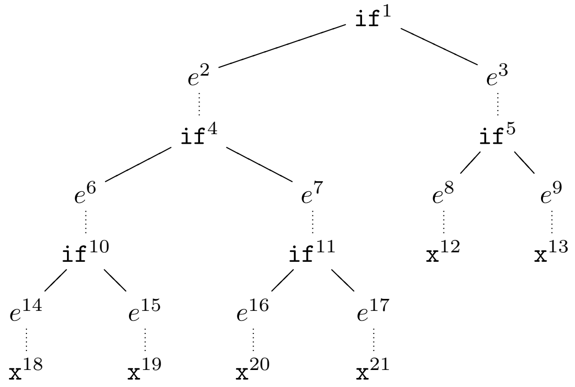


Figure 9: Example AST for skipping context selection

and form a finite height lattice that they monotonically climb when $[\![e]\!]_{out}$ and $[\![C[e]]\!]_{out}$ climb the value lattice.

When composing $\mathcal{V}_{C,e}$ we always reduce the composition to this canonical form. 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 quickly across multilayer contexts. 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 not in C. Section 4.3 shows how to compute and update these compositions efficiently.

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

4.2 Selecting context skips

We now have two ways to flow abstract values through a program. The first is via the constraint rules in figure 4. The second is via 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 contexts to be skipped for which theorem 4.3 is valid for a given variable. The length of a context is measured by the number of layers in the context. We fall back to the constraint rules in figure 4 when theorem 4.3 is not valid.

A different set of skips is selected for each variable, and we select longest skips for a particular variable, x, by starting with each reference to it, e, and finding the longest context, C, of e that does not contain x. C is then one of the contexts that we skip.

Since C is the longest 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 longest context of p that does not contain x and choose that context as one to be skipped. 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 longest contexts skipped 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`. The two children of each `if` are the consequent and the alternative. The test part of `if` is omitted for simplicity.

To select skips, all references to `x` are examined. In this case they are expressions 12, 13, 18, 19, 20 and 21. For each such expression, the longest 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, because of the reference to `x` in expression 9, the other child of expression 5, theorem 4.3 does not hold for moving type information for `x` from expression 8 to its parent, expression 5. Thus the context skipping function cannot be used there.

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

In the end the only places where the algorithm uses the constraint rules are expressions 1, 4, 5, 10, and 11. Everywhere else it uses context skipping functions. The entire scope of x is thus 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 context skips form a 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 variables are the leaves. Since the number of edges and the number of nodes in a tree are both 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

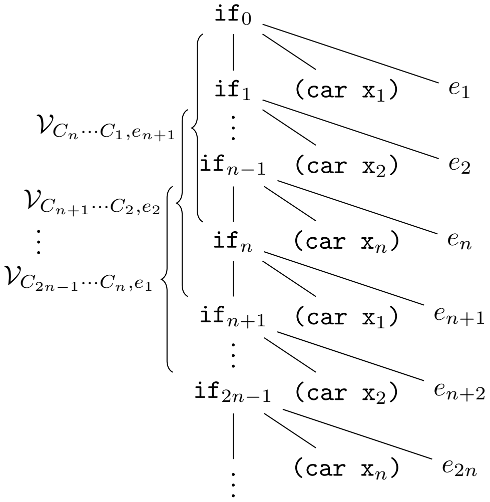


Figure 10: Example of quadratic $\mathcal{V}_{C,e}$ calculation

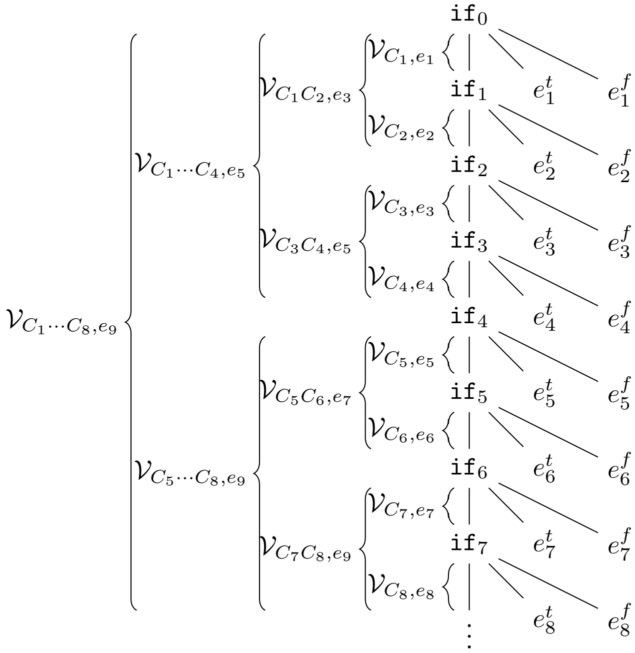


Figure 11: Layered structure of the $\mathcal{V}_{C,e}$ cache

linearly bounded by the number of references to that variable. Summing over all variables we are thus linear in the size of the program.

Finding the longest 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 construction and constant time for each query. Finding the longest skips amounts to finding the lowest common ancestor of an expression and the immediately 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 information about variables quickly across multiple layers. Once the skipping function, $\mathcal{V}_{C,e}$, is computed and reduced to the canonical form in theorem 4.4, it takes only constant time to move information across C for any variable not referenced in C.

We must be careful that the total time to construct all the skipping functions 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 takes $O(n^2)$ 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 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.

Figure 11 shows an example of the cached values for one path down a program's abstract syntax tree. Together all of these cached $\mathcal{V}_{C,e}$ form a tree structure. The same structure occurs on all other paths down the program tree. Each $\mathcal{V}_{C,e}$ shared between paths is stored only once in the cache.

The cache can be thought of as starting with the $\mathcal{V}_{C,e}$ single-layer contexts. 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.

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 $2^{i+1}k+2^{i+1}$.

This selection of cached $\mathcal{V}_{C,e}$ has the three important properties that ensure our linear-logarithmic bound. First, only $O(n \log n)$ skipping 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, the quadruple-layer $\mathcal{V}_{C,e}$ composed from it are updated, and so on. Thus, when abstract-value information is learned about an expression, at most logarithmically many $\mathcal{V}_{C,e}$ in the cache are updated. Each update takes constant time since each multilayer $\mathcal{V}_{C,e}$ in the cache is composed of two $\mathcal{V}_{C,e}$.

This caching strategy can be improved 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 skipping functions is constructed as a flow graph. 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 requires one flow-graph node, so this process creates $O(n \log n)$ nodes. 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 non-skipping point. A similar process is used for entering rather than exiting a context. Since in total there are linearly many skipping points, this creates linearly many nodes. Overall, this entire process takes linear-log time to construct the flow graph, and it 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 algorithm described in section 4 as an experimental optimization for 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 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 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 notes

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 evaluation order for function call arguments and `let` bindings.

A mutable variable's type can change between where type information is recovered and it is used. 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 evaluation order 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. We therefore 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 `let` are handled similarly.

5.3 Effectiveness

We tested the effectiveness of the type-recovery algorithm on a standard set of Scheme benchmarks [Clinger 2008]. Each test was run both with type recovery enabled and with type recovery disabled. The number of type checks performed in these two cases were then compared with each

[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.

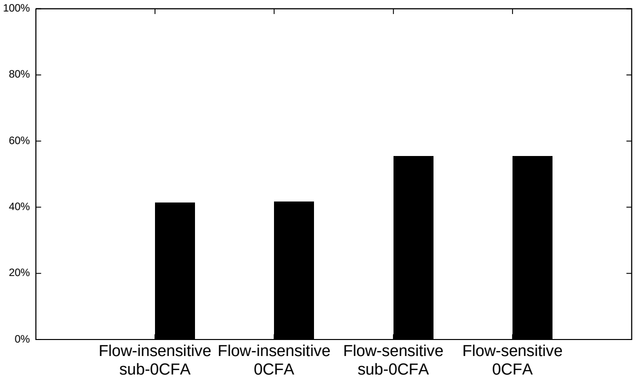


Figure 12: Percent of type checks removed

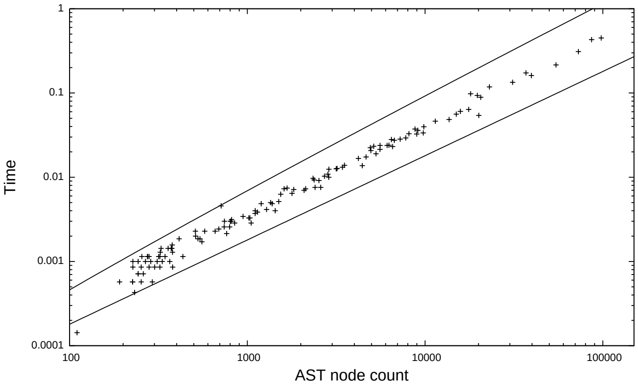


Figure 13: Source node count versus analysis time

other. Our tests look only at type checks caused by pair and vector primitives (e.g. car, cdr, cadr, vector-ref, vector-set!, etc.). An average of 69.1% of type checks are eliminated from the code, which results in 55.35% fewer type checks at run-time. We compared these results with a flow-insensitive version of the analysis which performed significantly worse, eliminating 41.3% of run-time type checks. We also compared the flow-sensitive sub-0CFA results with a flow-sensitive 0CFA. The 0CFA version performed only slightly better, eliminating 55.39% of type checks at run time. A flow-insensitive 0CFA performed slightly better then the flow-insensitive sub-0CFA, eliminating 41.7% of run-time type checks. Figure 12 compares the average percent of type checks eliminated for the flow-insensitive sub-0CFA, flow-insensitive 0CFA, flow-sensitive sub-0CFA, flow-sensitive 0CFA. Figure 14 gives the percent of checks eliminated for each individual program.

While these results are promising, they are not necessarily a good predictor for how well type-recovery will perform in general. Using the same set of tests, and counting only the checks made by pair primitives, only 36.4% of checks can be eliminated, resulting in the elimination 30.6% of run-time checks. Pairs are a hard case for type-recovery. While vectors store a number of items, pairs store only two. At best we expect to see a call to `car` paired with a call to `cdr` and can eliminate only one of the two type checks. In operations such as `cadr` and `cddr` only the first of two pair checks can be eliminated. This is because the contents of pairs are not tracked by the analysis. Hence, it cannot determine that the `cdr` of a pair is also a pair, so the nested pair must always be type checked. It is also common in Scheme to structure data into proper lists, and use an explicit `null?` check to determine when the end of the list is reached. Unfortunately, the `null?` check does not eliminate the need for the `pair?` checks implicit in `car` and `cdr`, since it tells the analysis only that the value is not null. Beyond the difficulties in handling pairs, some opportunities for eliminating type checks are already handled by the source optimizer before getting to the type recovery analysis.

Although the analysis does not require the evaluation order 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. For instance, in `(f (car x) (cdr x))` 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 evaluation order, we tested with both left-to-right and right-to-left evaluation orders. In both cases, although the benefit is significant in a few cases, the average number of type checks at run time improved by only around 5%.

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 algorithm and measuring the time it takes for the algorithm to run. For this test, we used the same Scheme benchmarks as before along with the 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 281,500 AST nodes (approximately 100,000 lines of code) per second.

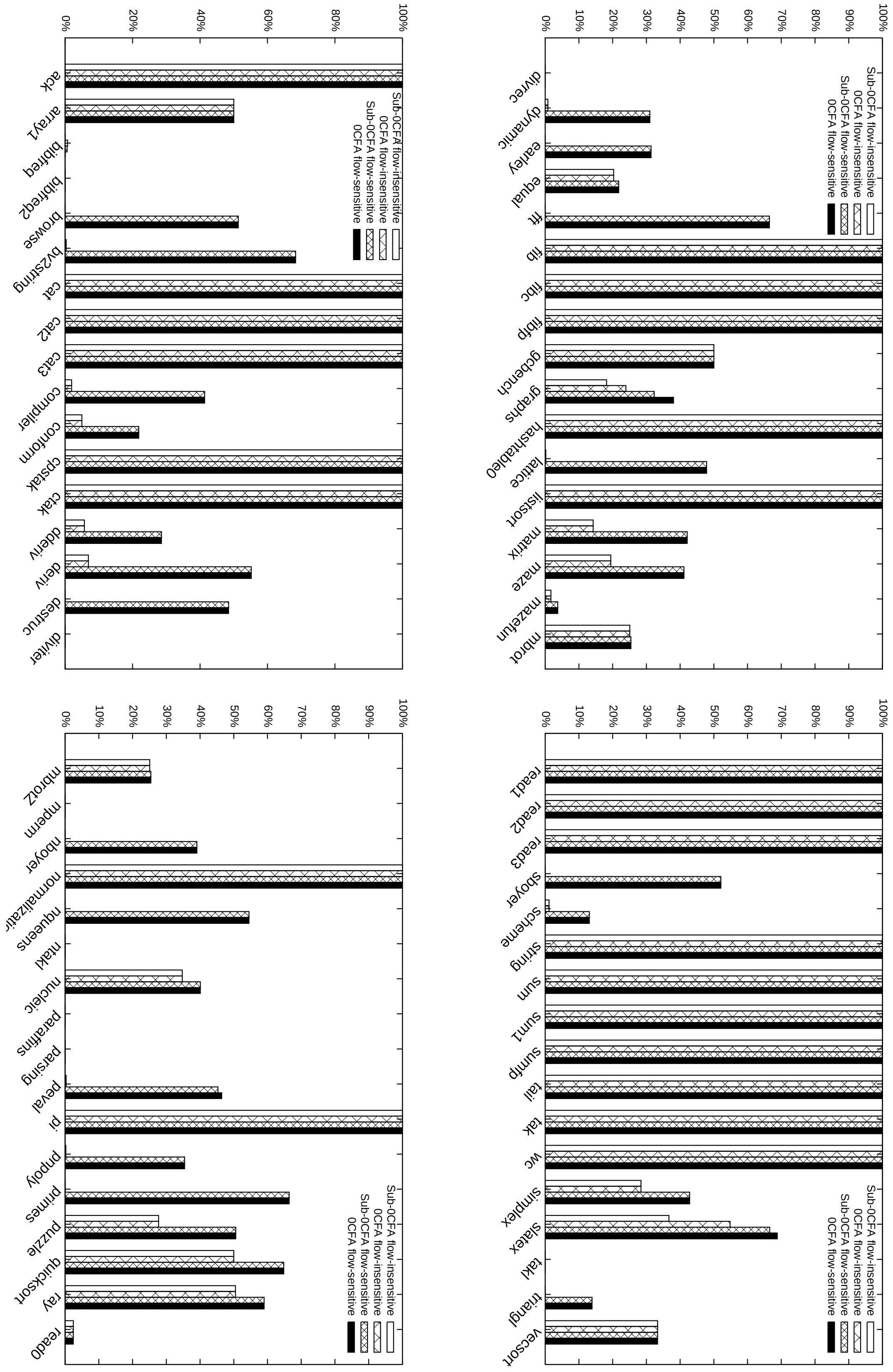


Figure 14: Percent of type checks removed

6. Related work

6.1 CFA and CFA-based type recovery

Shivers [1991] 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. We do not treat quantity information in our analysis, and instead rely 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 than ours.

Serrano [1995] argues that 0CFA is useful in compilers for functional languages 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 41.7% of type tests. The difference is likely attributable to different sets of benchmarks being used and to different strategies for inserting and counting type checks. As it is based on 0CFA, Serrano's analysis takes $O(n^3)$ time in the worst-case.

Heintze and McAllester [1997b] describe a linear-time CFA. It is targeted at typed languages and assumes bounds on the sizes of types. In linear time it can list up to a constant number of targets for all call sites, and in quadratic time it can list all targets for all call sites. 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 analyses 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. 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 runtime exceptions.

Henglein [1992a] presents a fast $O(n\alpha(n))$ tagging optimization algorithm for Scheme. Using the terminology of Steenkiste [1991], the goal of the algorithm is to statically eliminate dynamic *tag insertion* and *tag removal* operations. In contrast, we seek to eliminate dynamic *tag checks*. Values in Chez Scheme are always tagged as the garbage collector relies upon them. Henglein reports that his algorithm is able to eliminate around 40% of the executed tag insertion operations and around 55% of the executed tag removal operations across six non-numerical benchmarks. Although related, these numbers are not comparable to our number of eliminated tag checking operations. In a companion paper, Henglein [1994] addresses 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 expresses types as formulas in a propositional logic that has some similarities to the lattice structure underlying our analysis.

6.3 Recent type-recovery applications

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. However, their focus is more on precision than efficiency. The analysis has since been re-targeted to JavaScript in the form of DoctorJS [Mozilla Corporation 2011].

For type checking dynamically typed programs, Guha et al. [2011] combine a type system and a flow analysis such that the latter boosts the precision of the former. Like our analysis, their flow analysis is flow-sensitive and computes tag sets for each occurrence of a variable. Unlike our analysis, it is not interprocedural. Instead it relies on the type system at function boundaries. It has a quadratic worst-case time complexity.

6.4 Other related work

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 conditionals. Wegman and Zadeck list elimination of run-time type checks in a Lisp dialect as a possible use 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. However, the resulting work-list algorithm is polynomial 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 100,000 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 be more precise, particularly if the decisions were influenced by the needs of the analysis. Our experiments show that the typical benefit is likely 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] is 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

Comments from Dan Friedman, Jeremiah Willcock, Lindsey Kuper, Alex Hearn, and the anonymous reviewers led to several improvements in this paper's presentation.

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*, pages 253–265. ACM, 1973. 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(3): 441–456, May 2004. doi: 10.1007/s00224-004-1155-5.

J. Michael Ashley and R. Kent Dybvig. A practical and flexible flow analysis for higher-order languages. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(4):845–868, July 1998. doi: 10.1145/291891.291898.

John P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 29–41. ACM, 1979. doi: 10.1145/567752.567756.

Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292. ACM, 1991. doi: 10.1145/113445.113469.

Swarat Chaudhuri. Subcubic algorithms for recursive state machines. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 159–169. ACM, 2008. doi: 10.1145/1328438.1328460.

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

Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282. ACM, 1979. doi: 10.1145/567752.567778.

Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 57–68. ACM, 2002. doi: 10.1145/512529.512538.

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 *Programming Languages and Systems*, volume 6602, pages 256–275. Springer Berlin / Heidelberg, 2011. doi: 10.1007/978-3-642-19718-5_14.

Nevin Heintze and David McAllester. On the complexity of set-based analysis. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 150–163. ACM, 1997a. doi: 10.1145/258948.258963.

Nevin Heintze and David McAllester. Linear-time subtransitive control flow analysis. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 261–272. ACM, 1997b. doi: 10.1145/258915.258939.

Fritz Henglein. Global tagging optimization by type inference. *ACM SIGPLAN Lisp Pointers*, V(1):205–215, January 1992a. doi: 10.1145/141478.141542.

Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, June 1994. doi: 10.1016/0167-6423(94)00004-2.

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

Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 393–407. ACM, 1995. doi: 10.1145/199448.199536.

Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. Single and loving it: Must-alias analysis for higher-order languages. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 329–341. ACM, 1998. doi: 10.1145/268946.268973.

Simon Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Static Analysis*, volume 5673, pages 238–255. Springer Berlin / Heidelberg, 2009. doi: 10.1007/978-3-642-03237-0_17.

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, October 2000. doi: 10.1016/S0304-3975(00)00049-9.

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, December 2000. doi: 10.1023/A:1026551132647.

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 *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 66–75. ACM, 1984. doi: 10.1145/800017.800517.

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999. ISBN 978-3-540-65410-0.

William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, June 1990. doi: 10.1145/78973.78977.

Manuel Serrano. Control flow analysis: A functional languages compilation paradigm. In *Proceedings of the 1995 ACM symposium on Applied computing*, pages 118–122. ACM, 1995. doi: 10.1145/315891.315934.

Olin Shivers. Data-flow analysis and type recovery in Scheme. In Peter Lee, editor, *Topics in Advanced Language Implementations*, pages 47–87. The MIT Press, 1991.

Olin Shivers. Control flow analysis in Scheme. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 164–174. ACM, 1988. doi: 10.1145/53990.54007.

Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, September 2006.

Peter A. Steenkiste. The implementation of tags and run-time type checking. In Peter Lee, editor, *Topics in Advanced Language Implementations*, pages 3–24. The MIT Press, 1991.

Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 117–128. ACM, 2010. doi: 10.1145/1863543.1863561.

Dimitrios Vardoulakis and Olin Shivers. CFA2: A context-free approach to control-flow analysis. In *Programming Languages and Systems*, volume 6012, pages 570–589. Springer Berlin / Heidelberg, 2010. doi: 10.1007/978-3-642-11957-6_30.

Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):181–210, April 1991. doi: 10.1145/103135.103136.