Flow-Sensitive Sub-Zero Control-Flow Analysis in Linear-Log Time

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

Abstract
Traditional control-flow analysis (CFA) for higher-order languages is too expensive to be used as an every-day analysis in an otherwise fast compiler that is often called upon to compile large programs. It is even more expensive when flow sensitivity is added to handle optimizations like type recovery. In particular, the fastest known algorithms for 0CFA, the least expensive of the popular kCFA family of analyses, run in cubic time and are not flow-sensitive. A less precise form of control-flow analysis, sub-kCFA, runs in linear time, but is also not flow-sensitive. This paper presents a flow-sensitive sub-0CFA algorithm that runs in linear-log time. The algorithm has been used to perform a type recovery analysis that justifies the removal of over half of the run-time checks that would otherwise be necessary to ensure type safety in the latently typed language Scheme.

1. Introduction
Control-flow analysis (CFA) is an important tool for discovering static properties of programs in higher-order languages. Control-flow analysis and data-flow analysis differ in that the latter starts with a fixed control-flow graph and discovers how data flows along it, while the former discovers the control-flow graph as it runs. This is critical for higher-order languages in which procedures are data and the flow of data can determine the flow of control and hence the shape of the control-flow graph.

Because either analysis operates at compile time, exact data values are generally not known, so run-time values are approximated by abstract values. The quality and speed of either analysis is partly determined by the way in which data values are abstracted, particularly when multiple abstract values reach a particular program point. For a control-flow analysis, the quality and speed is also determined by the extent to which the information at a call site is taken into account when analyzing a procedure that might be invoked at that call site. Shivers categorizes control-flow analyses along this second dimension (Shivers 1988). A 0th-order CFA, or 0CFA, analyzes each procedure based on information from all of its possible call sites, whereas a 1CFA analyzes a procedure separately for each of its possible call sites. In general, kCFA analyzes each procedure based on each path of k calls that can lead to the procedure.

Another characteristic that distinguishes flow-analysis algorithms is whether they are flow-sensitive (Ashley and Dybvig 1998; Palsberg and Wand 2003). A flow-sensitive algorithm takes into account the flow of control in determining the set of values a variable can take on at different points in a program. A flow-insensitive analysis, on the other hand, assumes the set of values a variable can take on at one point in a program is the same as the set of values it can take on at any other point. For example, say \( f \equiv \lambda x. \text{if } x < 0 \text{ then } e_1 \text{ else } e_2 \), and all we know from the call sites of \( f \) is that it is passed integer values. With a flow-sensitive analysis, \( x \) is known to be less than zero in \( e_1 \) and greater than or equal to zero in \( e_2 \). With a flow-insensitive analysis, \( x \) is known only to be an integer in both \( e_1 \) and \( e_2 \).

Flow sensitivity is important for optimizations like type recovery (Shivers 1990), in which the flow of control through run-time type checks helps the compiler remove redundant checks. For example, in the Scheme expression \((+ (\text{car } x) (\text{cdr } x))\), a pair check is needed only for the first of the calls to \(\text{car} \) and \(\text{cdr} \). If the first check fails, the second is not reached, and if the first test succeeds, the second succeeds as well. Type recovery is most useful in latently typed languages like Scheme but is useful even in statically typed languages for reducing the number of null-pointer and similar checks. More generally, flow sensitivity can increase the overall quality of a flow analysis, since local refinements in the values a variable can take on can lead to the refinement of other values and a reduction in the number of control-flow links.

Control-flow analysis is expensive even when each procedure is analyzed once for all of its potential call sites and even when the analysis is not flow sensitive. The fastest known algorithm for flow-insensitive OCFA is \(O(n^3/\log n)\) for programs of size \(n\) (Melski and Reps 2000; Chaudhuri 2008; Midtgaard and Van Horn 2009). This is unacceptable for every-day use in a fast compiler that is often called upon to compile large programs. Even if programmers find the overall compile time for such an analysis and the optimizations it justifies to be acceptable, more benefit might be gained from spending that time on many less expensive analyses and optimizations, even if each is less effective than theoretically possible.

Fortunately, the control flow of some programs is fairly simple and can be discovered by a flow analysis that is less expensive than OCFA. Ashley and Dybvig (1998) describe such an analysis, sub-OCFA, that operates in linear time and is nearly as effective as OCFA at justifying certain optimizations for a set of real-world programs. Others have reported similar results for other simpler analyses which run in linear or near linear time (Heintze and McAllester 1997; Henglein 1992).

None of these faster algorithms are flow-sensitive, however. The goal of the research described in this paper is, therefore, to create a flow-sensitive CFA algorithm that is effective at justifying type recovery and related optimizations yet runs in something close to linear time. This paper presents the fruit of this research, a flow-sensitive CFA algorithm that runs in \(O(n \log n)\) time, achieved through a combination of design decisions and careful choices for low-level data structures and algorithms.

The remainder of this paper describes the basics of data- and control-flow analysis (Section 2), our basic algorithm (Section 3), an optimized version of the algorithm (Section 4), benchmark results that demonstrate the effectiveness and scalability of the optimized algorithm (Section 5), related work (Section 6), and conclusions (Section 7).

2. Background
Our algorithm builds upon both data-flow analysis and control-flow analysis. In this section we review the basics of these analyses and provide an introduction to issues specifically related to our analysis.
2.1 Standard data-flow analysis

Data-flow analysis is a general technique for computing what values occur at each expression or variable in a program. It operates over the control-flow graph of the program, with all nodes initially marked out-of-date. It selects an out-of-date node and computes new result values for its out-going edges based on the current values of in-coming edges. If the values on the out-going edges change, the destination nodes of those edges are marked out-of-date. The algorithm continues selecting and recalculating out-of-date nodes until the graph stabilizes and no out-of-date nodes remain.

For example, in the control-flow graph for the expression \( (\text{if } e_1 \lor e_2 \lor e_3) \) there will be a node representing the result value of the \( \text{if} \) that has edges from the nodes representing the result values \( e_2 \) and \( e_3 \) as input. Whenever the results of \( e_2 \) or \( e_3 \) are updated, the result node for the \( \text{if} \) is marked as out-of-date. Once that node is selected for update, its output value is computed from the union of the output values of \( e_2 \) and \( e_3 \).

A crucial property of data-flow analysis is that, under appropriate conditions, it converges to a solution in a bounded amount of time. Specifically, if the values that flow through the graph, \( G \), are members of a lattice, \( L \), and for each edge the value on that edge moves only monotonically up the lattice, and the output values of a node can be computed in constant time from the input values, then the lattice will converge in time \( O(|L| |G|) \) where \(|L| |G| \) are the height of \( L \) and the size of \( G \) respectively.

In the absence of higher-order procedures or the equivalent, the control-flow graph is linear in the size of the program. By grouping values into a finite set of types (e.g., integers, pairs, vectors, etc.) and choosing our lattice to be over sets of types, the lattice is of constant height. With this linearly sized graph and constant height lattice, the data-flow analysis then takes only linear time.

2.2 Standard control-flow analysis

Control-flow analysis can be viewed as a modification of data-flow analysis to handle the presence of higher-order procedures. Consider what happens if we naively apply standard data-flow analysis to:

\[
\begin{align*}
&\text{let } (\text{inc } (\lambda (x) (+ xure (1)))) \\
&\quad \quad (\text{double } (\lambda (f y) (f (f y))))! \\
&\text{double inc } 3)
\end{align*}
\]

The value of \( f \) will eventually include \( \text{inc} \), so the call \( (f y) \) causes a control-flow jump to \( \text{inc} \). This is known, however, only once the data-flow analysis has determined the possible values of \( f \). Thus, a complete control-flow graph cannot be constructed until after the data-flow analysis, but the data-flow analysis needs to know the complete control-flow graph in advance.

This cyclic dependency is solved by dynamically changing the graph as the data flow progresses. In its simplest incarnation, 0CFA (Shivers 1988), once the analysis determines that \( f \)’s value can be the procedure \( \text{inc} \), edges are added to the control flow graph that represent the jumps due to the function call and return. The analysis is guaranteed to converge because there are a finite number of possible edges, and edges are only added, never removed.

While the analysis is guaranteed to converge, it no longer guaranteed does so in linear time. 0CFA uses a lattice over sets of procedures. This lattice is as high as the number of procedures in the program. Thus we immediately have that the program that a program of size \( n \) with a graph of size \( O(n) \) and a lattice of size \( O(n) \) could take \( O(n^2) \) time for a data-flow analysis to solve.

The problem is made worse by the fact that 0CFA adds edges to the data-flow graph that correspond to control jumps from function calls and returns. In a higher-order language, any particular call site might jump to any procedure in the code, thus in the worst case a program of size \( n \) can contain \( O(n) \) call sites that each jump to \( O(n) \) different procedures. This results in a graph that contains \( O(n^2) \) edges. Even constructing such a graph would force our algorithm into super-linear behavior.

Taken together, these two factors mean that in a program of size \( O(n) \) we can have a lattice of height \( O(n) \) and a graph of size \( O(n^2) \). So a naively implemented 0CFA would take \( O(n^3) \) time to compute. Faster ways to compute 0CFA are known but at present the fastest known takes \( O(n^2/\log n) \) time (Melski and Reps 2000; Chaudhuri 2008; Midggaard and Van Horn 2009).

2.3 Top and escaped procedures

If control-flow analysis is operating on a program that contains free variables, e.g., imported from libraries outside the scope of the analysis, the analysis does not know anything about the values of those variables (Shivers 1988). References to these variables are thus approximated by using the top of the lattice. Top includes procedures unknown to the analysis.

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

Top values and escaped procedures can cause more top values or escaped variables. 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 might end up being passed to unknown procedures. Second, if a procedure escapes, its formal parameters are assumed to be top, and its return value escapes, since it might be called from places outside the scope of the analysis. Finally, when a set of procedures are joined with top value, the result is top. Since that top value does not explicitly mention the procedures combined into it, those procedures are marked as escaped.

2.4 Sub-zero control-flow analysis

To create a linear control-flow analysis algorithm, both the size of the graph (number of edges) and the height of the lattice must be limited. It turns out both can be accomplished simultaneously by approximating all non-singleton sets of procedures with the top element of the lattice. With this change, the height of the lattice (shown in Figure 1) is a bounded, conservative approximation of 0CFA’s power-set lattice. Furthermore, the values that flow to the function position of a particular call site either contain at most one procedure or are approximated by the top value. Thus the one-procedure cases can add at most a linear number of edges to the graph. The top-value case is handled just as it is described for standard 0CFA without needing to add any extra edges to the graph.

This approach means that more procedures escape than in 0CFA, but only procedures that escape lose information about their arguments. For example, when running the analysis over:

\[
\begin{align*}
&\text{let } ([\text{if } (\lambda (x) e_1)]) \\
&[g (\lambda (y) e_2)] \\
&[h (\lambda (z) e_3)]
\end{align*}
\]

This approach means that more procedures escape than in 0CFA, but only procedures that escape lose information about their arguments. For example, when running the analysis over:
both f and g escape, since they both flow into fg, and thus x and y go to top. However, h is not affected. A crucial point is that a procedure flowing to two different places does not cause it to escape; rather, procedures escape only when two or more flow to the same point, i.e., when a call site performs some sort of dispatch.

Ashley and Dybvig (1998) refer to this analysis as sub-0CFA. Their analysis achieves the effect of a constant-height lattice via a projection (widening) operator on values that limits sets of values to a singleton or unknown (top) value. Other lattices of constant or even logarithmic height can also be used to achieve linear or near linear analyses, provided they are conservative approximations of the power-set lattice. For example, instead of sets of at most one procedure, sets of at most k procedures for some constant k.

3. Basic algorithm

Sub-0CFA matches our desire for a linear CFA algorithm and allows us to gain constructive information. For instance, it can determine that x in the following expression is a pair

\[
\text{(let ([x (cons e1 e2)]) (if (pair? x) (car x) #f))}
\]

and can track this information through a function call as in the following expression.

\[
\text{(let ([f (lambda (x) (if (pair? x) (car x) #f))]} \text{[y (cons e1 e2))]} \text{[f y])}
\]

However, if a variable is top it cannot determine any more information beyond that point. For instance, in the expressions

\[
\text{(let ([x (read)]) (if (pair? x) (car x) #f))}
\]

and

\[
\text{(let ([x (read)]) (begin (cdr x) (car x)))}
\]

it cannot determine that x at (car x) must be a pair. This is exactly the information we would like to learn in an analysis like type recovery. This loss of precision is not peculiar to sub-0CFA, all kCFA suffer the same limitation.

Our analysis regains some of this lost precision by adding flow sensitivity. In its simplest form, flow sensitivity adds reachability information to CFA. A more sophisticated version of flow sensitivity can be used to learn restrictive information about values on the branches of a conditional. In the two examples above, the (pair? x) check and the (cdr x) (which makes an implicit pair check) provide the information we want.

Our CFA algorithm uses a constant-height lattice, a la sub-0CFA, and augments this with flow sensitivity. For instance, in the (pair? x) example above, the incoming value of x is top, but (pair? x) informs the analysis that x is a pair along the “then” branch and not a pair along the “else” branch. Similarly, if (cdr x) succeeds in the second example, we learn that x must be a pair from that point forward, as it raises an exception otherwise.

3.1 Overview of the algorithm

Conceptually, we can model flow sensitivity by flowing environments through the flow graph. In this model, each expression has an input environment and two output environments. Each environment maps variables to values that inhabit them. Environments with uninhabited variables are unreachable edges in the control-flow graph. The input environment represents what we know about variables coming into the expression. One output environment represents what we know about these variables if the return value of the expression is true and the other represents what we know about these variables if the return value of the expression is false. Splitting the output environment in this way allows the analysis to use the true output environment as evaluation proceeds along the “then” branch of a conditional and the false output environment as evaluation proceeds along the “else” branch.

For example, the output environments of (pair? x) contain a value for x that is a pair type in the true case and the set of all non-pair types in the false case. In a larger expression such as

\[
\text{(let ([x (read)]) (if (pair? x) (car x) #f)} \text{[cdr x]} \text{[error])...}
\]

the environment flowing out of (pair? x) is split by the inner if. The values of x when (pair? x) is true are carried into the “then” branch while the values of x when (pair? x) is false are carried into the “else” branch. In other words, down the “then” branch, x is known to be a pair, and down the “else” branch, x is known to be some type other than a pair.

Continuing this example, the output environment of (car x) indicates that x is a pair in both true and false cases since the result of car can be either true or false depending on the contents of x. In Scheme, branching is always handled through a boolean test in an if expression. In languages with native algebraic data types and case statements, environments might need to be further divided.
of the pair. On the other hand since #f is always false, the output environment of #f indicates in the true case that x is uninhabited, i.e., bottom, but in the false case is a non-pair.

The corresponding parts of these two output environments for the “then” and “else” branches are joined to produce the output environment of the inner if expression. For the true case, x is the union of the pair type from the “then” branch and the uninhabited type from the “else” branch, i.e., the pair type. Likewise for the false case, x is the union of the pair type from the “then” branch and all non-pairs types from the “else” branch, i.e., top.

A similar process happens for the outer if which means that going into the “then” branch x is known to be a pair, and going into the “else” branch x is top. Coming out of the “then” branch, x is a pair in both the true and false cases, but since error or doesn’t return, it evaluates to bottom. Coming out of the “else” branch, x is bottom in both the true and false cases. Coming out of the outer if, in the

data (S, E, K) ∈ State = ψ(Enter + Exit) × Envmap × Stack

\begin{align*}
(S, E, K) \in \text{State} &= \psi(\text{Enter} + \text{Exit}) \times \text{Envmap} \times \text{Stack} \\
(e, \hat{\rho})_{in} \in \text{Enter} &= \mathcal{E}xp \times \mathcal{E}nv \\
(\hat{t}, \hat{\rho}, f)_{out} \in \text{Exit} &= \hat{\mathcal{Ty}}pe \times \hat{\mathcal{E}nv} \times \mathcal{E}nv \times \mathcal{E}xp \\
\hat{\rho} \in \mathcal{E}nv &= \text{Var} \mapsto \hat{\mathcal{Ty}}pe \\
K \in \text{Stack} &= \mathcal{E}xp \mapsto \psi(\text{Kont} \times \mathcal{E}xp) \\
\hat{\rho} = \psi(\text{Proc} \times \psi(\text{Tag}))
\end{align*}

true case, x is known to be the union of pair from the “then” branch and bottom from the “else” branch, i.e., pair, and in the false case, x is known to be the union of pair from the “then” branch and bottom from the “else” branch, i.e., pair. Thus the analysis determines that x has a pair type in any expressions following the outer if.

3.2 Semantics

We model in Figure 2 the concrete semantics as a CEK machine (Flanagan et al. 1993), a canonical three component abstract machine operating over expressions, environments, and stacks. The machine operates by repeated transitions between eval and apply states. We assume that a number of primitives are bound in the machine’s initial environment. Each primitive is represented by a unique tag o, which is interpreted by a helper function η upon application. Note this version of CEK machine is not tail-call optimized, as it leaves a frame on top of the stack in all tail positions.
The proof relies on a number of lemmas, stating that the concretization function over values and environments are mutually recursive. Concretization of a set of abstract stacks takes the union of the concretization of each abstract stack in the set. Concretization of an abstract state $(S, E, K)$ joins the concretization of each element of $S$. An abstract out-state corresponds to a series of traces: a suffix thereof represents the evaluation of the expression at hand in a concrete true or false environment and ends in a concrete apply state with a true or false value, respectively. We are now in position to state the soundness theorem.

**Theorem 3.1 (Soundness).**

$$
\Sigma \subseteq \gamma(S, E, K) \implies F(\Sigma) \subseteq \gamma(F(S, E, K))
$$

**Proof.** The proof relies on a number of lemmas, stating that the concretization functions are monotone in each of their arguments. Another lemma states that the concretization function over values is a complete meet morphism: $\gamma(\langle \sigma \cap \nu', E \rangle) = \gamma(\langle \sigma, E \rangle) \cap \gamma(\langle \nu', E \rangle)$.

Assuming $\Sigma \subseteq \gamma(S, E, K)$, we argue that a given $\gamma \in F(\Sigma)$ also belongs to $\gamma(F(S, E, K))$. We proceed by case analysis. If $\sigma \in \Sigma_0$ we argue that $\sigma \in \gamma(\Sigma_0)$. If on the other hand $\sigma \in \sigma_\text{ss} \setminus \sigma_\text{ss'}$, we know $\sigma' \in \sigma''s'$ and proceed by case analysis on the transition $s \rightarrow s'$. Most cases are straightforward utilizing that $\sigma''s \in \gamma(S, E, K)$. The most intricate (return) case further utilizes that the domain of traces is prefix-closed in order to reconstruct the original call site from the string $\sigma''s$. □

By fixed point induction it follows that the abstract fixed point is an approximation of the concrete fixed point: If $F \subseteq \gamma(\mathbb{I}p F)$. Hence the analysis safely approximates all possible evaluation traces of the concrete semantics.

### 3.3 Lattice refinement and escaping lambdas

The powerset lattice $\mathcal{P}(\text{Lam} + \text{Prim})$ has linear height and hence violates the constant height condition. Thus as a first step we substitute it with the flat, constant height lattice $\mathcal{P}(\text{Lam} + \text{Prim})_\text{flat}$, depicted in Figure 1. As a consequence the analysis needs to handle calls to $\top$. We do so by adding an additional abstract transition:

$$
\langle \hat{\nu}_1, \hat{\rho}_1, \hat{\rho}_f, s' \rangle_\text{out} \xrightarrow{\text{E.K.}} (\top, \hat{\rho}_f, \hat{\rho}_f, c_0 e_1)_\text{out}
$$

(call top)

Intuitively the transition says that the result of calling an unknown procedure can be anything. As a result the join of the true and false environments constitutes a conservative output environment.

In a separate compilation setting the analysis can lose track of lambda’s when they escape by being saved in a cons-cell, passed as arguments to external procedures, etc. We keep track of such escaping lambda’s by including them in a explicit escape set as traditional (Shivers 1991). Thus whenever a lambda $\lambda x.e$ escapes it can potentially be invoked from anywhere, hence we conservatively add it to the escape set any functions returned by the lambda.

The result of these modifications is a flow-sensitive sub-OCFA. However directly implementing the analysis by flowing the abstract environment through the program would still not meet our linear-logarithmic time objective. Even though the flat lattice has constant height, the abstract environment lattice still has linear height. Furthermore a pointwise join of two such environments takes linear

Due to space constraints we only include a representative selection of concretization rules.

---

Figure 4. Concretization functions

This more directly matches the transfer from return to call sites in the forthcoming analysis.

Figure 3 shows the corresponding abstract semantics. The abstract semantics operates over a set of abstract enter and exit states as well as three global maps, recording environment, stack, and escape information. Values in the analysis are represented by a pair of lattices: a procedural lattice and a non-procedural lattice. The procedural lattice is the powerset of all lambdas and primitives. The non-procedural lattice is the powerset of tags. As in it’s concrete counterpart a helper function $\hat{\gamma}$ interprets primitives upon application. Furthermore another helper function $R$ sharpens information about arguments upon return. We have overloaded the $\top$, $\bot$, $\bot$, and $\cap$ notation to respectively mean top, bottom, join, and meet of the corresponding lattice.

We relate states of the abstract semantics to traces of the concrete semantics. Formally a trace is a sequence of states: $s_0 \ldots s_n = \sigma \in \text{State}^*$ such that for $i \in \{0, \ldots, n-1\}$: $s_i \rightarrow s_{i+1}$. We now consider a trace-based collecting semantics by taking lfp $F$, the least fixed point of the following transition function $F$, over the domain of prefix-closed traces $\Sigma$ (Cousot and Cousot 2004).

$$
F(\Sigma) = \Sigma_0 \cup \{ \sigma ss' \mid \sigma s \in \Sigma \land s \rightarrow s' \}
$$

with $\Sigma_0$ being the singleton set of the initial state of the machine.

The abstract transition function $F$ mimics the above, with lfp $\hat{F}$ constituting a computable, yet asymptotically unoptimized analysis.

$$
\hat{F}(S, E, K) = \Sigma_0 \cup \bigcup_{\delta \in S} \{ (\hat{\sigma'}, E', K') \}
$$

with $\Sigma_0$ being the abstract initial state of the analysis.

\[\varepsilon 1325\]

5

\[2010/7/15\]

\[\text{2010/7/15}\]
time when implemented naively, thus still leading to a cubic time analysis. Nevertheless it provides a good starting point from which to design linear-log time algorithm.

4. Optimized algorithm

We now turn to the question of implementing the analysis in linear-log time. We do this in two steps. First, we recast the program in the style of Midtgaard and Jensen (2009). These are shown in Figure 5. Each expression is associated with its syntactic context, $K(e)$. A reachability flag, $\hat{r}$ is added to simplify representing the analysis as a data-flow graph. To define a data-flow graph for the program, five nodes are associated with each source expression. Two nodes correspond to $\hat{r}$ and $\hat{\rho}_n$ in $[e]_{in}$, and other three correspond to $\hat{\nu}$, $\hat{\rho}_1$ and $\hat{\rho}_f$ in $[e]_{out}$.

Consider the expression $(\text{if } e_1 \; e_2 \; e_3)$ and its flow graph in Figure 6. The flow graphs for $e_1$, $e_2$, and $e_3$ are omitted, but since every expression has two nodes going in and three nodes going out, the flow graph for if connected to them the same way no matter what they are. Double headed arrows represent edges that flow entire environments. Single headed arrows represent edges that flow types or reachability flags. The input and output edges for each expression are (from left to right) $\hat{\nu}$ and $\hat{\rho}_n$ on the top and $\hat{\nu}$, $\hat{\rho}_1$ and $\hat{\rho}_f$ on the bottom. The $T^?$ and $F^?$ nodes test whether the input type contains any true or false values respectively.

The constraints for call and return involve extra work as they add edges to the flow graph while the graph is running and connections between functions and call sites are discovered. This adds at most a linear number of edges to the graph since the type that flows to the function position of a call contains at most one function. Otherwise it becomes $T$. Thus, the size of the flow graph remains linear in the size of the source program.

Since the transfer functions in the flow graph are monotone, the flow graph converges in a number of updates proportional to the product of the number of edges and the lattice height of each edge. Unfortunately, the values flowing over edges are entire environments. The type of each variable in an environment is over a constant-height lattice, but the lattice height for an environment as a whole is linear in the number of variables. In the worst case this leads to a quadratic number of updates.

4.1 Flow-graph representation

We recast the analysis as a constraint system by reading the constraints directly off the small-step abstract semantics in the style of Midtgaard and Jensen (2009). These are shown in Figure 5. Each expression is associated with its syntactic context, $K(e)$. A reachability flag, $\hat{r}$ is added to simplify representing the analysis as a data-flow graph. To define a data-flow graph for the program, five nodes are associated with each source expression. Two nodes correspond to $\hat{r}$ and $\hat{\rho}_n$ in $[e]_{in}$, and other three correspond to $\hat{\nu}$, $\hat{\rho}_1$ and $\hat{\rho}_f$ in $[e]_{out}$.

Consider the expression $(\text{if } e_1 \; e_2 \; e_3)$ and its flow graph in Figure 6. The flow graphs for $e_1$, $e_2$, and $e_3$ are omitted, but since every expression has two nodes going in and three nodes going out, the flow graph for if connected to them the same way no matter what they are. Double headed arrows represent edges that flow entire environments. Single headed arrows represent edges that flow types or reachability flags. The input and output edges for each expression are (from left to right) $\hat{\nu}$ and $\hat{\rho}_n$ on the top and $\hat{\nu}$, $\hat{\rho}_1$ and $\hat{\rho}_f$ on the bottom. The $T^?$ and $F^?$ nodes test whether the input type contains any true or false values respectively.

The constraints for call and return involve extra work as they add edges to the flow graph while the graph is running and connections between functions and call sites are discovered. This adds at most a linear number of edges to the graph since the type that flows to the function position of a call contains at most one function. Otherwise it becomes $T$. Thus, the size of the flow graph remains linear in the size of the source program.

Since the transfer functions in the flow graph are monotone, the flow graph converges in a number of updates proportional to the product of the number of edges and the lattice height of each edge. Unfortunately, the values flowing over edges are entire environments. The type of each variable in an environment is over a constant-height lattice, but the lattice height for an environment as a whole is linear in the number of variables. In the worst case this leads to a quadratic number of updates.

4.2 $\phi$-Placement

To get to a linear-log time bound, instead of flowing whole environments, our algorithm moves type information directly from one syntactic occurrence of the variable to the next syntactic occurrence of that variable, skipping over intermediate expressions. For example, consider the following code fragment.

$(\\lambda (x_1 \; x_2 \; \ldots \; x_n) \; \ (\text{car } x_1) \; (\text{car } x_2) \; \ldots \; (\text{car } x_n))$

If the analysis flows whole environments through each step of the program, then it can take quadratic time for the flow graph to converge. This happens if, say, each $x_i$ argument to the function is sequentially discovered to be of pair type. When the $x_i$ argument is discovered to be a pair the environment entering $(\text{car } x_i)$ is updated with the new type information for $x_i$. This in turn updates the output environments of that expression which are fed into $(\text{car } x_2)$. These updated environments continue propagating all the way down to $(\text{car } x_n)$. This takes $O(n)$ updates. When the $x_2$ argument is discovered to be a pair a similar sequence of $O(n)$ updates occurs. Over all $n$ variables, there are thus $O(n^2)$ updates.

We observe, however, that the type for each $x_i$ could be moved directly to the next occurrence of $x_i$ in the code, since expressions where $x_i$ is not mentioned neither care about nor change the type of $x_i$. In straight-line code, linking one reference to the next is simple, but we must take care when the code branches. For example, consider a program fragment with the control flow schematically depicted in Figure 7. Assume type information is discovered at each reference to $x$. Also assume each $e_i$ expression contains one reference to $x$ and an indeterminate argument evaluation order. With $n$ references to $x$ in the $n$ branches, connecting each of them directly to each of the $m$ references in the $e_i$ expressions results in a quadratic number of edges.

An obvious solution, inspired by single-static-assignment form (SSA), is to place $\phi$-nodes at the join points where the values of $x$ merge together. However, though $\phi$-node placement algorithms for SSA are linear in the number of $\phi$-nodes placed, they can place a quadratic number of $\phi$-nodes (Cytron et al. 1991). An example of this is Figure 8. The standard SSA $\phi$-placement algorithm places a $\phi$-node at the end of each if for each $x_i$. If there are $m$ branches and $n$ variable references then this results in $O(nm)$ $\phi$-nodes in a program of size $O(n + m)$.
φ values too early. By placing nodes later in the program, fewer need to be placed. However, placing the φ-nodes too late also gives rise to a quadratic number of nodes as seen in the example in Figure 7. We need a middle ground between these two extremes that places at most a linear number of nodes.

The solution that our analysis uses is best explained by an example. The program is considered in terms of its abstract-syntax tree, so for example consider the skeletal abstract-syntax tree shown in Figure 8. Another example for φ-node placement

Figure 5. Analysis constraint rules

\[ [e]_{in} ∈ Bool \times Env \quad [e]_{out} ∈ T_{type} \times Env \times Env \]

\[ [e]_{in} \supset (φ, 0) ⇒ [e]_{out} \supset \langle ABS(e), \hat{φ}, 0 \rangle \quad (const) \]

\[ [λx.e]_{in} \supset (φ, 0) ⇒ [λx.e]_{out} \supset \langle λx.e, \hat{φ}, 0 \rangle \quad \land E(λx.e) \supset \hat{φ} \]

\[ [e_0 \; e_1]_{in} \supset (φ, 0) ⇒ [e_0]_{in} \supset \langle φ, 0 \rangle \quad (operator) \]

\[ [if \; e_0 \; e_1 \; e_2]_{in} \supset (φ, 0) ⇒ [e_0]_{in} \supset \langle φ, 0 \rangle \quad (cond) \]

\[ [e_0]_{out} \supset \langle v_0, \hat{φ}_i, \hat{φ}_f \rangle, E(e_0) = (\square e_1) \]

\[ ⇒ [e_1]_{in} \supset \langle \top, \hat{φ}_i \cup \hat{φ}_f \rangle \quad (operand) \]

\[ [e_0]_{out} \supset \langle v_0, \hat{φ}_i, \hat{φ}_f \rangle, E(e_0) = (\square e_1) \]

\[ ⇒ [e_1]_{in} \supset \langle \top, \hat{φ}_i \cup \hat{φ}_f \rangle \quad (seq \ next) \]

\[ [e_0]_{out} \supset \langle v_0, \hat{φ}_i, \hat{φ}_f \rangle, E(e_0) = (if \; e_1 \; e_2) \]

\[ ⇒ [e_1]_{in} \supset \langle \top, \hat{φ}_i \rangle \quad (cond \ true) \]

\[ [e_0]_{out} \supset \langle v_0, \hat{φ}_i, \hat{φ}_f \rangle, E(e_0) = (if \; e_1 \; e_2) \]

\[ ⇒ [e_2]_{in} \supset \langle \top, \hat{φ}_f \rangle \quad (cond \ false) \]

\[ \mathbb{K}(e) = \square (\square e_0) \mid (e_1) \mid (\square λx.e) \mid (\square e_1) \mid (e_0; e_1) \mid (if \; e_1 \; e_2) \mid (if \; e_0 \; e_1) \]

\[ [x]_{in} \supset (φ, 0) ⇒ [x]_{out} \supset \langle φ \cup \hat{f}, \hat{φ}[x \mapsto φ] \rangle \]

where \( φ = \hat{φ}(x) \cap \top \quad (var) \]

\[ [e_0]_{out} \supset \langle λx.e, \hat{φ}_i, \hat{φ}_f \rangle \quad (call) \]

\[ [e_0]_{out} \supset \langle ∩, \hat{φ}_i, \hat{φ}_f \rangle \]

\[ [e_0]_{out} \supset \langle ∩, \hat{φ}_i, \hat{φ}_f \rangle \]

\[ [e_0]_{out} \supset \langle ∩, \hat{φ}_i, \hat{φ}_f \rangle \]

\[ [e_0]_{out} \supset \langle ∩, \hat{φ}_i, \hat{φ}_f \rangle \]

\[ K(e_0) = (e_1) \]

\[ K(e_1) = (if \; e_0 \; e_2) ⇒ [if \; e_0 \; e_1 \; e_2]_{out} \supset [e_1]_{out} \quad (ret \ then) \]

\[ K(e_2) = (if \; e_0 \; e_1) ⇒ [if \; e_0 \; e_1 \; e_2]_{out} \supset [e_2]_{out} \quad (ret \ else) \]

\[ K(e_1) = (e_0; e_1) ⇒ [e_0; e_1]_{in} \supset [e_1]_{out} \quad (seq \ last) \]

However, the nodes that are placed by an SSA-style placement algorithm are often redundant. In this example, they each join in the same type information from the initial \( x_i \) occurrences at the start of the program with that from the \( x_i \) occurrences in the if branch. This information needs to be joined only once. In this example we can place the φ-nodes at the end of just the outermost if. This places one φ-node per variable and is thus linear.

The SSA φ-placement algorithm is quadratic because it joins values too early. By placing nodes later in the program, fewer need to be placed. However, placing the φ-nodes too late also gives rise to a quadratic number of nodes as seen in the example in Figure 7. We need a middle ground between these two extremes that places at most a linear number of nodes.

The solution that our analysis uses is best explained by an example. The program is considered in terms of its abstract-syntax tree, so for example consider the skeletal abstract-syntax tree shown in Figure 8. Another example for φ-node placement

Figure 9. Dotted lines represent where there may be multiple layers in the AST that are omitted from the diagram.

Each variable is considered independently and each reference to a particular variable examined. For each reference the furthest ancestor that does not contain additional references is found and the immediate parent of that ancestor marked as a position for φ-node placement. For example, the algorithm might start with node 18. The furthest ancestor without additional references is node 14, so the parent node, node 10, is marked.

In our analysis, φ-nodes behave like references to a variable (just like φ-nodes in SSA behave like assignments). When a node is marked for φ-node placement, that node is also examined and the furthest ancestor found that does not contain additional references to the particular variable in question. References already contained by the node do not count. In this example, the furthest ancestor of node 10 that does not contain additional references to \( x \) is node 6. Thus its parent, node 4, is marked. Again the process repeats, marking node 1. When node 19 is examined, node 15 is determined to be the furthest ancestor that contains no additional references to \( x \), so node 10 should be marked. However, since node 10 is already marked, the algorithm does no further work with node 10. Instead, it continues with the next reference to \( x \). In total, the algorithm marks nodes 1, 4, 5, 10 and 11 for φ-node placement.

Pseudo-code for this algorithm is in Figure 10. This algorithm ensures each node in \( Φ \) has at least two nodes under it that are either also in \( Φ \) or are a reference to variable \( v \). For example in Figure 9 nodes 1, 4, 5, 10 and 11 will be placed into \( Φ \). Node 1 has both 4 and 5 under it. Node 4 has both 10 and 11 under it. And node 10 has two references to \( x \) under it. Thus the nodes form a tree-like structure. The leaves of the tree are the references to the variable, and the internal nodes of the tree are the nodes in \( Φ \). Since the number of internal nodes of a tree is linearly bounded by the number of leaves, the size of \( Φ \) is linearly bounded by the number of references to \( v \).

The construction of this algorithm is such that all edges between φ-nodes are directly either up or down the abstract syntax tree. Thus the number of edges is also linear in the number of references to \( v \).

For each variable \( v \), the algorithm is linear in the number of references to that variable, so when this algorithm is iteratively run
if e false values and (car x) is always a pair. On the other hand, if only true values and (cdr x) is never a pair. Other cases arise if either expression diverges or returns both true and false values.

However, the types that flow to (car x) depend only on the types of e1 and e2. They do not depend on the internal structure of e1 and e2. This gives rise to the following theorem where T?(e, u) and F?(e, u) are defined in Figure 11.

**Theorem 4.1 (Expression Skipping).** If x is a variable not mentioned in e then

\[
(\hat{\rho}(x), \hat{\rho}(x)) = (T?(e, \hat{\rho}_{in}(x)), F?(e, \hat{\rho}_{in}(x)))
\]

where \( (\hat{r}, \hat{\rho}_{in}) = [e]_{\text{in}} \)

Proof. By induction on e and using the constraint rules as needed to symbolically compute \( (\hat{\rho}(x), \hat{\rho}(x)) \) from \( \hat{\rho}_{in}(x) \).

This theorem allows us to directly compute the types of x at the end of e1 and e2 given the types at the start of e1 and e2. The constraint rules can then move types from e1 and e2 to (car x).

### 4.3.2 Skipping over contexts

Theorem 4.1 is not general enough for movements between φ-nodes. Consider the following expression where the test context of if is nested multiple times. Assume x is not referenced in t1 or f1, and all-pair? returns true if and only if all arguments are pairs.

\[
(\text{if} (\text{if} \ldots (\text{if} \text{all-pair?} x_1 \ldots x_n) \ldots t_m f_m) t_1 f_1)
\]

This theorem allows us to move type information from all-pair? to the immediately enclosing if based on the values of t_m and f_m. From there it can be propagated outward through each enclosing if based on the values of each t_i and f_i.

However, this is quadratic because moving the type for x_1 from (all-pair? x_1 \ldots) to (car x_1) involves m steps. This linear factor is paid by each of the n different variables so the cost of transferring the type information for all variables is O(mn) in this expression of size O(m + n).

This computation can be optimized by observing that the transformation applied to the type of x_1 is the same for all of the other x_i. Thus we can pre-compute the transformation though all if expressions once and apply that transformation to each variable. If the resulting transformation can be applied in constant time, then this takes only O(m + n) time.

Before formally specifying how to compute these transformations we consider one more example. Suppose we have the following code. Assume t_i returns both true and false values and x is not in t_i or e_i, and consider whether (cdr x) needs a pair check.

\[
(\text{if} (\text{if} t_1 \ldots (\text{if} \text{all-pair?} x e_n \ldots) e_2) e_1)
\]

The answer depends on the type of each e_i. If they never return true, then test of the outermost if is true only when control flow passes through (car x). On the other hand, if any e_i ever returns true, then (cdr x) can be reached without passing through (car x).
it computes in logarithmic time the type transformation across an arbitrary multilayer context. Second, since the transformation depends on the types of the expressions in the context, the cache updates itself in logarithmic time when those types change.

The cache can be thought of as starting with single-layer \( V_{C,e} \) transformations. That is to say it stores the transformation information necessary to move a variable by a single step from the outgoing environment of one expression to the enclosing expression’s outgoing environment. If the cache stores only these, then when the type of an expression changes, the cache updates itself in constant time, but computing transformations for multilayer contexts from individual \( V_{C,e} \) takes linear time. Caching the results of all queries does not help as the cache is invalidated whenever a constituent \( V_{C,e} \) changes due to new type information about an expression, and the cached results have to be recomputed to determine whether values depending on those results need to be updated. However, by strategically selecting intermediate transformations to cache we can guarantee logarithmic time for both query and update.

4.4.1 Myers stacks

The structure for caching these transformations is based on the applicative random-access stacks (Myers 1984). A Myers stack is similar to an ordinary reference list with one major difference: in addition to the data (i.e., car) and next (i.e., cdr) fields each cell also contains a jump field pointing further down the stack. These jump fields are arranged so that any location in the stack can be reached from the head in a logarithmic number of jumps. The structure of the jump fields in a fifteen element Myers stack are shown in Figure 13.

When adding a new cell to the head of a Myers stack, the target of the jump field depends on the jump length of the existing head and the jump length of the cell pointed to by the jump field of the existing head. If they are equal, then it points to the cell pointed to by the jump field of the cell pointed to by the jump field of the existing head. Otherwise, the jump field points to the existing head. For example, when adding cell 14 to the jump list in Figure 13, the existing head, cell 13, has a jump field pointing to cell 10 which has a jump field pointing to cell 7. These jumps are of the same length, so the jump field of cell 14 points to cell 7.

In a Myers stack it is always possible to navigate from one location to another following logarithmically many links by following the jump field whenever it does not go past the desired destination and following the next field otherwise. For example to get from cell 15 to cell 2, one would follow the next field of cell 15 to cell 14. Then take the jump field of cell 14 to cell 7. Then the next field of cell 7 to cell 6, the jump field of cell 6 to cell 3 and finally the next field of cell 3 to cell 2.

4.4.2 Transformation composition with Myers stacks

To implement our cache, we modify the Myers stack design by storing a transformation matrix along with each jump and each next pointer. Each next field is associated with transformation matrix for a single \( V_{C,e} \), and each jump field is associated with the composition of the \( V_{C,e} \) skipped over by the jump.

To compute the transformation between any two locations, the transformations attached to the logarithmically many edges to navigate between those locations are composed. Since two transformations compose in constant time, the transformation between any two points can thus be computed in logarithmic time.

Further, when a single-step transformation is updated, only logarithmically many stored transformation functions need to be recomputed. The structure of Myers stacks ensure both that there are only a logarithmic number of links that cross over a particular point, and that each link can be computed from the composition of three links within it. In Figure 13, for example, if the transformation be-
appropriate transform the type information between occurrences.

At a ϕ-node the type information for the variable is modified before being passed to the next occurrence depending on the AST node with which it is associated. For example, at the end of an if the types from the consequent and alternative are joined together, but between the test and consequent the type for the true case is selected.

The cache and the ϕ-node connections are represented using the same data-flow framework as the rest of the analysis. The cache adds linearly many flow-graph edges and the ϕ-node connections add linear-log many edges. The values flowing over these edges monotonically increase over constant-height lattices and updated values can be computed in constant time. Thus the flow-graph for the optimized analysis converges in linear-log time.

5. Empirical evaluation

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, successfully compiles Chez Scheme itself, and runs the entire Chez Scheme test suite without errors.

Type recovery and the elimination of type checks is performed by three passes that run in sequence: a preprocessing pass, the CFA pass, and a postprocessing pass. The purpose of the preprocessing pass is to make explicit all run-time type checks in the code. For example, the call (car e) implicitly contains a pair check. The preprocessing pass makes this explicit as follows.

\[
(\lambda (x) \quad 
  (\text{if} \ (\text{pair} \ ? \ x) \quad 
    (\text{unsafe-car} \ x) \quad 
    (\text{bottom} \ (\text{car} \ x))))
\]

In the code produced by the preprocessor, unsafe-prim is a variant of prim that assumes its argument is correct and thus performs no run-time check, while plain prim checks its argument and raises an exception if it is not of the correct type. The bottom wrapper indicates to the CFA algorithm that the call to the safe primitive is not expected to return. The preprocessing pass runs in linear time, and the size of the preprocessed code is linear in the size of the source code. The CFA pass uses the optimized algorithm to determine type information. The postprocessing pass eliminates dead and useless code resulting from type recovery, as well as any left-over bottom wrappers. It also recognizes when no tests have been eliminated in the expansion of a primitive call and reverts back to the original call to avoid generating unnecessary extra code. For example, if the analysis determines that e always evaluates to a pair in the example above, the unnecessary test is eliminated and the body of the \lambda expression reduces to, simply, (unsafe-car x). Otherwise, it reverts to (car x). For primitives that perform multiple checks, like vector-ref, some tests might be eliminated while others are retained. Explicit type checks in the source code can also be eliminated along with dead code that results from knowing the outcome of the test.

4.4.3 Myers stacks on trees

In the analysis, transformation are calculated between two locations in the abstract syntax-tree, and of course the abstract syntax-tree is a tree and not a list. Our analysis maps Myers stacks to this tree structure by taking advantage of tail sharing. Consider for example the skeletal AST structure in Figure 14. The solid lines and bullets represent the shape of the AST with the root of the AST at the top of the figure. The path from node 11 to the root is one Myers stack with the jump links depicted by arrows. The path from node 13 to the root is another Myers stack but it shares a common tail with the first Myers stack starting at node 4. This tail sharing ensures that the size of the cache is linear in the size of the AST as each AST node contains a single jump link.

4.5 Algorithm Summary

To put all these pieces together the optimized algorithm works as follows. First, the program’s AST is annotated with the Myers stacks that act as the transformation cache. Recall that queries to this cache are valid only for variables not mentioned between the start and end points of the query. Next the ϕ-placement algorithm places linearly many ϕ-nodes. For a particular variable, regions between ϕ-nodes contain no reference to that variable. References and ϕ-nodes chain together using query results from the cache to appropriately transform the type information between occurrences.

Figure 13. Example of Myers applicative random-access stacks

Figure 14. An example AST, overlaid with Myers stacks
The analysis handles a variety of language features, like \texttt{letrec}, that the algorithm described in Section 4 does not handle, and it properly handles the unspecified order of evaluation for function calls, \texttt{set}, and \texttt{letrec} bindings. The analysis also handles mutable variables, though these gather only constructive information. Restrictive information cannot be used for these variables, since their type may change between the site where restrictive type information is recovered and the next use if, for instance, an intervening function call alters the value. The analysis uses a database of primitives to supply the information that flows through the graph, so this information is not built into the code.

We tested the effectiveness of the type recovery by counting the number of type checks made at run time with and without type recovery on a set of published R6RS Benchmarks (Clinger 2008). Figure 15 shows the percentage of type checks eliminated. On average, 58.1% of run-time type checks are eliminated. In the cases where 100% of the type checks are eliminated, either the programmer included explicit type checks or the primitive was called directly on a value whose construction site was along the path for the primitive.

These results are encouraging, and we expect to be able to make additional improvements as we refine the implementation. The primitive database currently includes information for only a small set of basic primitives such as \texttt{pair?}, \texttt{cons}, \texttt{car}, \texttt{cdr}, \texttt{set-car!}, \texttt{set-cdr!}, \texttt{vector?}, \texttt{make-vector}, \texttt{vector-ref}, and \texttt{vector-set!}. Information from other primitives, like \texttt{assq} and \texttt{map}, should lead to better type recovery: \texttt{assq} if it returns, always returns \texttt{#f} or a pair, and \texttt{map}, if it returns, always returns a pair if passed a pair. The analysis currently also treats all pairs and all vectors the same, although we could treat each occurrence of \texttt{cons} and \texttt{make-vector} in the source code as a separate element in the lattice, in a manner analogous to the way we handle \texttt{lambda} expressions, to get more information about the contents of some pairs and vectors. Furthermore, evaluation orders for function calls, \texttt{let} bindings, and \texttt{letrec} bindings are currently selected by the register allocator, which runs later in the compiler, but the analysis would generally be more successful if the orders were fixed prior to flow analysis. For instance, in the expression \texttt{(f (car x) (cdr x))}, the analysis does not know whether the call to \texttt{car} will occur before the call to \texttt{cdr}, so it cannot currently eliminate the implicit pair check from either based on the other.

Beyond the effectiveness of our analysis, we also verified its asymptotic behavior 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. Figure 16 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. A combination of timings from the R6RS Benchmarks and the self-compilation of the Chez Scheme compiler and run-time system make up the data for the graph.

6. Related work

Heintze and McAllester (1997) describe a linear time CFA. It is specifically targeted at typed languages and assumes bounds on the types. It can either (a) list up to a constant number of targets (or \texttt{T}) for all call sites, or (b) list all targets for each call site in quadratic time. 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 \textit{simple closure analysis} (Henglein 1992) 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 sub-OCFA analysis, without flow sensitivity, 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 (but still potentially finite-height) lattices can be constructed. Their analysis achieves a limited form of flow sensitivity when the test of an \texttt{if} is a type predicate applied to a variable by creating new bindings for the variable in the “then” and “else” parts of the \texttt{if} whose abstract values are as restricted by the test. Vardoulakis and Shivers (2010) describes a \textit{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.

To prove soundness of the analysis we have used the concretization framework of abstract interpretation (Cousot and Cousot 1992). Following Hoare (1978), Cousot and Cousot (1979) 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.

Shivers (1990) uses an extension of OCF A to perform type recovery. Instead of directly discovering type information about variables, he adds a level of indirection and discovers information about the \textit{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 re-flow semantics to correct for the problems caused by the impression around quantities. We do not to 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, even without the additional indirection.

Soft typing (Cartwright and Fagan 1991) and, more recently, gradual typing (Siek and Taha 2006) are designed to produce, through type inference, statically typed programs from latently typed programs, possibly containing some type declarations, 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.

7. Conclusions and future work
While less accurate than 0CFA, sub-0CFA has been shown to run in linear time and to be effective at justifying certain optimizations. This paper describes an extension of sub-0CFA that runs in linear-log time and adds flow sensitivity. It is thus capable of performing type recovery and justifying the elimination of run-time type and null-pointer checks. Even with a small database of primitives, it justifies, on average, the removal of more than half of all run-time checks in a standard set of benchmarks for the latently typed language Scheme.

While the analysis conservatively handles the unspecified evaluation order of function calls and binding forms, 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. 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 latently type language, functions can be grouped by arity.

Our α-placement algorithm assumes programs are structured, i.e., do not contain arbitrary gotos. We would like to see if it can be extended to handle non-structured code within the linear-log time bound.

We conjecture that the same techniques we have used to extend sub-0CFA analysis with flow sensitivity can be applied more generally to kCFA analyses with the addition of a single log factor. It would be useful to verify this and to compare the resulting analyses with our flow-sensitive 0CFA analysis.

References