Environment-Sharing Analysis and Caller-Provided Environments for Higher-Order Languages

J. A. CARR, University of Chicago, USA
BENJAMIN QUIRING, University of Maryland, USA
JOHN REPPY, University of Chicago, USA
OLIN SHIVERS, Northeastern University, USA
SKYE SOSS, University of Chicago, USA
BYRON ZHONG, University of Chicago, USA

The representation of functions in higher-order languages includes both the function's code and an *environment* structure that captures the bindings of the function's free variables. This paper explores caller-provided environments, where instead of *packaging* the entirety of a function's environment in its closure, a function can be *provided* with a portion of its environment by its caller. In higher-order languages, it is difficult to determine where functions are called, let alone what pieces of the function's environment are available to be provided by the caller, thus we need a higher-order control-flow analysis to enable caller-provided environments.

In this paper, we present a new abstract-interpretation-based analysis that discovers which pieces of a function's environment are always shared between its definition and its callers. In such cases, the caller can provide the environment to the callee. Our analysis has been formalized in the Rocq proof assistant. We evaluate our analysis on a collection of programs demonstrating that it is both scalable and provides significantly better information over the common syntactic approach and better information than *lightweight closure conversion*. In fact, it yields the theoretical upper-bound for many programs.

For caller-provided environments, deciding how to transform the program based on these revealed facts is also non-trivial and has the potential to incur extra runtime cost over standard strategies. We discuss how to make these decisions in a way that avoids the extra costs and how to transform a program accordingly. We also propose other uses of the analysis results beyond enabling caller-provided environments. We evaluate our transformation using an instrumented interpreter, showing that our approach is effective in reducing dynamic allocations for environments.

CCS Concepts: • **Software and its engineering → Compilers**; *Automated static analysis*; *Functional languages.*

Additional Key Words and Phrases: Higher-Order Flow Analysis, Environment Sharing, Alias Analysis

ACM Reference Format:
J. A. Carr, Benjamin Quiring, John Reppy, Olin Shivers, Skye Soss, and Byron Zhong. 2025. Environment-Sharing Analysis and Caller-Provided Environments for Higher-Order Languages. *Proc. ACM Program. Lang.* 9, ICFP, Article 247 (August 2025), 30 pages. https://doi.org/10.1145/3747516

Authors' Contact Information: J. A. Carr, Department of Computer Science, University of Chicago, Chicago, IL, USA, jacarr@uchicago.edu; Benjamin Quiring, Department of Computer Science, University of Maryland, College Park, MD, USA, bquiring@umd.edu; John Reppy, Department of Computer Science, University of Chicago, Chicago, IL, USA, jhr@cs.chicago.edu; Olin Shivers, Khoury College of Computer Sciences, Northeastern University, Boston, MA, USA, shivers@ccs.neu.edu; Skye Soss, Department of Computer Science, University of Chicago, Chicago, IL, USA, ssoss@uchicago.edu; Byron Zhong, Department of Computer Science, University of Chicago, Chicago, IL, USA, byronzhong@uchicago.edu.

1 Introduction

Functions in a lexically scoped language implicitly capture the bindings of their free variables at the point of definition. The runtime representation of a function, referred to as a *closure* [Landin 1964], consists of its code pointer and the *environment* at the point where the function was defined (in practice, we restrict the environment to the free variables of the function). Thus, effective environment management is one of the key aspects of implementing higher-order functional languages. A simple, but common, representation of a function value is the well-known *flat closure*, where the environment is represented as a record (or tuple) containing one field for each free variable [Cardelli 1983]. We use the term *packaged environment* to refer to the part of a function's environment that is stored in some sort of heap-allocated data structure, such as a flat closure record. As we discuss below, sometimes it is possible to avoid packaging some, or all, of the environment.

For example, consider the simple SML program in Listing 1(a). An unoptimized compilation of this program yields an executable that binds `n` to 5, creates closures for `f` and `h`, and then calls `h` on `f`. Listing 1(b) shows an explicit flat-closure implementation of the code.[1] In this example, the environment for `f` is a singleton record `{n=5}`, while the environment for `h` is empty. The function `f` packages its environment as a record because it needs access to the value bound to `n` when it is called.

Efficient implementation of higher-order functions relies on efficient management of their environments, which is the focus of our 3CPS project.[2] Our prior work on binding extents [Quiring et al. 2022a,b] reduces the cost of packaged environments by placing them in high-performance machine resources such as the registers and the stack. But sometimes, the variable bindings that are referenced by a function are available in the caller's environment. In this situation, the compiler can avoid packaging those bindings into the function's environment record and instead arrange for the caller to provide them. In the example, the call-site "`k 6`" (which is a call to `f`) is in scope of the binding of `n` that the function `f` requires. Additionally, the call to `h` is also in the scope of the same binding of `n`. Listing 2(a) shows how we can modify the definitions of `f`, `h` and the calls to `h` and `k` to arrange for `n` to be provided to `h` and `f` through parameters, instead of being packaged by their respective environments. With the change shown in Listing 2(b), the packaged environments for `f` and `h` are empty, saving an allocation over the naïve flat closure conversion.

In general, identifying when one can use caller-provided environments to represent closures requires an analysis. For first-order programs, such an analysis is purely syntactic, but a syntactic

```
let
  val n = 5
  val f = fn x => x + n
  val h = fn k => k 6
in
  h f
end
```

(a) Original code

```
let
  val n = 5
  fun f_cd ({n}, x) = x + n
  fun h_cd ({}, k) = #code k (#env k, 6)
  val f = {code=f_cd, env={n=n}}
  val h = {code=h_cd, env={}}
in
  #code h (#env h, f)
end
```

(b) Explicit flat closures

Listing 1. A simple example

[1]The SML notation "`#n`" projects the field labeled "`n`" from a labeled record.
[2]More information about 3CPS can be found at https://github.com/3cps-project.

```
let
  val n = 5
  val f = fn (x, n') => x + n'
  val h = fn (k, n') => k (6, n')
in
  h (f, n)
end
```

(a) After transformation

```
let
  val n = 5
  fun f_cd ({}, (x, n')) = x + n'
  fun h_cd ({}, (k, n')) =
    #code k (#env k, (6, n'))
  val f = {code=f_cd, env={}}
  val h = {code=h_cd, env={}}
in
  #code h (#env h, (f, n))
end
```

(b) Explicit flat closures

Listing 2. The example from Listing 1 using caller-provided environment

analysis is of limited value for higher-order languages. This paper describes the notion of caller-provided environments in higher-order languages, along with an abstract-interpretation-based analysis that enables it. The abstract notion of providing an environment structure from a caller to its callee is useful for generalizing optimizations across different runtime closure implementations.

The earliest example of caller-provided environments is the use of *displays* to support lexically-nested functions in Algol 60 compilers [Hauck and Dent 1968; Randell and Russell 1964]. A display is effectively a stack of frame pointers that can be used by a callee to access variables bound in lexically-enclosing outer functions. The setting of Algol 60 is much simpler than that of our higher-order intermediate language, since Algol 60 programs typically consist only of first-order functions.[3] In the first-order setting, a call site for a function f must always be in the scope of f's definition, which means that the free variables of f must also be in scope at the call site (ignoring shadowing, which can be dealt with by α-conversion). Thus, it is always possible to provide the environment of the callee at any call site via a display or other mechanism. We use caller-provided environments to describe a form of display called *higher-order displays* in Section 5.2.

Fundamentally, the underlying analysis that describes whether it is possible to pass environment from the call must answer: is some variable binding x the same instance of x between the caller's environment and the callee's? In the control-flow analysis literature, this is known as the *environment problem.* Shivers articulated the environment problem in his dissertation [Shivers 1991], and introduced "reflow analysis" as one technique for solving the problem. At a high level, this problem is difficult because abstractions map potentially infinitely many bindings of x onto just a finite few; you can tell if bindings are definitely *not* the same (if they are different even in finite), but not if they are.

Following Shivers's dissertation, there have been several subsequent results that address the environment problem. The first, lightweight closure conversion [Siskind 1999; Steckler and Wand 1997], is an instance of integrating caller-provided environments in higher-order languages. The others are the must-alias analysis "Single and Loving It" papers [Germane and McCarthy 2021; Jagannathan et al. 1998], which aim to be more precise than LWCC, especially with respect to mutable cells. This increased precision requires the use of abstract garbage collection [Might and Shivers 2006] and per-program-point stores. In common with reflow analysis, these features cause the analysis to be notably more expensive than simple store-widened 0-CFA.

[3]The ability to pass functions as downwards arguments to other functions is allowed in Algol 60, but requires extra runtime support and is not a focus of this work.

The analysis we describe is qualitatively different from these, using a simple, direct strategy to prove that elements in the abstraction are equal in the concrete. Instead of relying on *single*ness, we essentially track witnesses of equality between abstract bindings, which are propagated through function calls as the abstract interpretation runs. We term this an *environment-sharing* analysis, and this strategy enables solving the environment problem in different programs than prior work. We provide a detailed discussion of these differences in Section 7. We also provide a quantitative comparison to lightweight closure conversion [Steckler and Wand 1997] in Section 4, as that prior work is the most similar to our caller-provided environment transformation.

This paper makes the following contributions, which are broadly split between two categories. First, it describes an analysis for environment sharing.

- It presents an abstract-interpretation-style analysis that finds the environment sharing facts: what portions of environment between a caller (call site) and its callees (function values being called) are always the same (Section 3). The analysis has the same cubic worst-case complexity as standard monovariant CFAs.
- This analysis has been formalized and proven sound in the Rocq proof assistant.[4]
- It evaluates the cost and quality of the analysis on a number of benchmark programs that show that the analysis is scalable and precise. Our prototype, which handles a substantial subset of Standard ML, can analyze thousands of lines of code in a fraction of a second. A comparison of our analysis against both the standard syntactic analysis and Lightweight Closure Conversion [Steckler and Wand 1997] demonstrates that our analysis provides better information. In 88% of our benchmarks, our analysis yields the same results as the theoretical upper-bound (Section 4).

Second, this paper explains how a compiler might use these facts to make optimization decisions.

- It presents an interprocedural, flow-based transformation that turns packaged bindings into parameters, given a decision for which bindings this should be done (supported by the facts produced by the analysis).
- It describes the pitfalls that might degrade the program's performance and proposes two policies for making sharing decisions (Section 5).
- It describes how the information that the analysis provides can be used to implement *higher-order displays* (Section 5.2) and flow-directed closure sharing (Section 5.3).
- It evaluates the transformed programs using both proposed policies, demonstrating that both policies result in a reduction of free-variable dynamic allocations for closures in most programs (Section 6).

2 Environment Sharing

We use the word *environment* to refer to the mapping from variables to values that is used to evaluate variables. In general, there are two kinds of environments: the *evaluation environment* for the current program location and a packaged environment in each closure that provides the variable bindings for the closure's free variables. For simplicity, the abstract machine packages *all* variable bindings in scope when constructing a closure, rather than restricting to just the free variables.

The core information that underlies caller-provided environment is *environment sharing*: considering both the possible environments at a call site and the environments of the possible closures it invokes, which variable bindings are always the same (*i.e.*, map to the same value) in all of these environments?

[4]Proofs are available as supplemental material.

Consider the function call "f *arg*." When evaluating the call, f refers to a closure that packages an environment, which (in the abstract machine underlying the analysis) contains all variable bindings in scope when the closure was created. We need to know which bindings the packaged environment shares with the environment at the call. As a trivial example, if f is a λ-term then the call shares all environment with the callee.

We say that "f shares variables $x_1, \dots, x_n$" if the bindings for (*syn.* values associated with) $x_1, \dots, x_n$ are the same between the binding environment of f and the environment of all possible closures associated with f. This is what our analysis computes: for each variable f, what variable bindings do the environments of all closures possibly associated with f share with the environment in which f is bound? Since any call to f occurs in an environment that extends f's binding environment, the variables that f's closures share with its binding environment are still shared in the calling environment.

To motivate how the program analysis works, consider the code from Listing 1(a), where we have added labels to identify different locations in the code.

```
let
[l1] val n = 5
[l2] val f = fn x => [l3] x + n
[l4] val h = fn k => [l5] k 6
in
  [l6] h f
end
```

The goal is to determine that at the call "k 6" the variable k will always refer to a closure that has the same binding of n that the environment of the call has access to. Let us walk through how this program executes. To begin, n is bound and added it to the environment. Then the program binds f to a closure containing the environment {n}. Now, f *must* share the binding n with the current environment, as f's closure just packaged the current environment. We can track this information by associating n with f. The current state of the program is,

location:	l_4
environment:	{f ↦ ⟨**fn** x **=>** x + n, {n ↦ 5}⟩}
sharing:	{f ↦ {n}}

Next, h is bound to a closure packaging {n, f}, and we similarly record that h shares both n and f with the current environment.

location:	l_6
environment:	{f ↦ ⟨**fn** x **=>** x + n, {n ↦ 5}⟩, h ↦ ⟨**fn** k **=>** k 6, {f ↦ ⟨**fn** x **=>** x + n, {n ↦ 5}⟩, n ↦ 5}⟩}
sharing:	{f ↦ {n}, h ↦ {n, f}}

The call "h f" invokes the function **fn** k => k 6. At this point, we know that h (the function) shares the binding for n with the current environment, and that f (the argument) *also* shares the binding for n with the current environment and therefore, due to transitivity, (the value associated with) h shares the binding for n with (the value associated with) f. Then, as the program steps into the body of h, the current environment *becomes* the environment from h's closure, extended with the new binding for k. Now, k is bound to the same value as f; any sharing between the environment of f's closure and h's closure is inherited to k's closure with the (new) current environment.

location:	l_5
environment:	{n ↦ 5, k ↦ ⟨**fn** x **=>** x + n, {n ↦ 5}⟩}
sharing:	{k ↦ {n}}

$$\begin{array}{rcll} x \in \mathit{Var} & = & \textit{a set of variables} & \\ \mathit{lam} \in \mathit{Lam} & ::= & (\lambda\ (x_1\ \ldots\ x_n)\ e) & \text{Functions} \\ \mathit{arg}, f \in \mathit{Arg} & = & \mathit{Lam} \cup \mathit{Var} & \text{Arguments} \\ e \in \mathit{Exp} & ::= & (f\ \mathit{arg}_1\ \ldots\ \mathit{arg}_n) & \text{Calls} \end{array}$$

Fig. 1. Grammar of the core CPS intermediate representation

The program finishes by calling `k` and returning twice, which does not change the sharing information associated with `k`. After finishing, we can inspect in particular the program states associated with the two calls "`h f`" and "`k 6`" and find that in both cases, the value of the variable being called shares the binding `n` with the environment of the call.

Each piece of sharing information associating a variable f with a shared variable is essentially a *witness* that two values are equal. Equality witnesses are created through reflexivity (creating a closure), and we can use transitivity to derive new witnesses. Finally, we can also propagate equality witnesses through calls.

3 Analysis

In this section, we present our abstract-interpretation-style analysis for determining environment sharing. We develop this analysis in two phases: first we define a small-step operational collecting semantics that determines valid sharing information for each variable based on concrete program traces. Unfortunately this information is not computable in general, so in the second phase we define an abstract semantics that is a sound over-approximation. The abstract machine is the computable fact-finder for per-variable sharing information, from which we can extract the per-call sharing information (our static analysis).

The relationship between the concrete and the abstract is often complex. To manage the complexity, we follow a standard strategy of incorporating the complex parts of the analysis into a collecting semantics [Shivers 1991, §2.3]. In this case, we lift the discovery of sharing information into the collecting semantics, where it is easier to understand, and the abstraction of this semantics is straightforward. This approach also has the effect of factoring the proof: one goal is to show that the equality witnesses produced by the collecting semantics are valid (*i.e.*, they represent equality), and the other is to show that the analysis only finds a subset (*i.e.*, sound over-approximation) of these witnesses.

3.1 CPS

We present our analysis using the CPS-style intermediate representation (IR) that is defined in Figure 1. Our CPS IR is deliberately minimal (only a single expression variant) as the focus of this work is on *functions*. The analysis we present along with the later discussions of caller-provided environment can be naturally extended to support other language features such as **`letrec`**, data structures, and conditional branches (all of which our implementation handles), or other flavors of IR such as ANF. As is typical in a compiler, we assume that variable names are renamed to be unique.

We use $\ell \in \mathit{Lam} \cup \mathit{Exp}$ to denote non-variable terms. We also make use of several syntactic properties of non-variable terms:

- $\mathit{availVars} : \mathit{Lam} \cup \mathit{Exp} \longrightarrow \mathcal{P}(\mathit{Var})$ maps a term to the set of variables that are in scope at that location in the program.
- $\mathit{freeVars} : \mathit{Lam} \cup \mathit{Exp} \longrightarrow \mathcal{P}(\mathit{Var})$ maps a term to the set of *free variables* in the term, and

- *enclosingFn* : $Lam \cup Exp \rightharpoonup Lam$ maps a term to the immediately enclosing function (undefined for the top-level term).
- *callsOf* : $Lam \to \mathcal{P}(Exp)$ maps a function to calls that are enclosed by the function; *i.e.*, $call \in callsOf(lam) \implies enclosingFn(call) = lam$.

3.2 Collecting Semantics

The semantic domains of the collecting semantics are provided in Figure 2. We use the notation $[X]^2$ to denote the set of unordered pairs of Xs. The abstract machine uses small-step environment semantics. A state $\varsigma \in State$ contains a program location, an environment $\rho \in Env$ that maps variables to addresses *Addr*, and a store $\sigma \in Store$ that maps addresses to values $clo \in Value$. Values are just closures, consisting of a λ-expression and an environment. To find the value associated with a variable its address is located in the environment, which is then looked up in the store. The collecting semantics represents actual traces through the program. As the program steps, it allocates a fresh address for each new binding. The association of a variable with its address uniquely describes the variable binding in an execution.

Finally, a state holds sharing information $sh \in Share$, which is a map from addresses to sets of variables. Instead of associating each *variable* with a set of shared variables, as done in the example in Section 2, we generalize to associating *addresses*, which provides extra precision in the analysis.[5] An entry in a *sh* map, $[a \mapsto \mathcal{X}]$, indicates that the environment that binds a variable to address a shared every variable in $\mathcal{X}$ with the environment of the closure that a refers to in the store. In other words, given a variable x, we track the set of bindings that the value of x shares with the *binding environment* of x. When x is called, the environment of the caller is an extension of x's binding environment. Extending the environment does not destroy sharing (assuming unique variable bindings), so x will still share $\mathcal{X}$ at the call and its closure can be provided those bindings from the scope of the call.

We formally define correctness for *sh* in a program state to be that this information witnesses actual equality of variable bindings:

Definition 3.1 (Sharing Invariant).

$$\begin{aligned} &validShare\ \varsigma = \forall a \in \mathrm{dom}(\sigma), \rho' \in envs, x \in Var\ .\ (a \in \mathrm{im}(\rho') \wedge x \in sh\ a \implies \rho''\ x = \rho'\ x) \\ &\quad \text{where} \quad \begin{array}{lcl} \langle \_, \rho, \sigma, sh \rangle & = & \varsigma \\ envs & = & \{\rho\} \cup \{\rho' \mid \langle lam, \rho' \rangle \in \mathrm{im}(\sigma)\} \\ \langle \_, \rho'' \rangle & = & \sigma\ a \end{array} \end{aligned}$$

This property states that for any environment ρ' found in the program state (the environment of the current location or the environment in a closure), if the address a is found in ρ', then any sharing indicated by $sh\ a$ should indeed exist between ρ' and the environment of the closure that a refers to, ρ''. When checking equality, we check that the uniquely-allocated *addresses* are equal, which is a stronger property than the variables taking on the same *value*.

Instead of directly computing the sharing information for each state, we track and propagate these witnesses because the equality between addresses does not *abstract*. The set of addresses becomes finite in the abstraction, which the concrete addresses map onto. If two abstract addresses are equal, they could still refer to distinct concrete addresses; we cannot lift equalities from the abstraction back to the concrete. The opposite is true, however, distinct abstract addresses imply distinct concrete addresses.

[5]Instead of one fact per variable, we can have different facts associated with a variable, depending on the context (*e.g.*, the value that it is bound to, or what the most recent function call was). For 0-CFA, however, this generalization provides no increase in precision.

$$\begin{array}{rcl@{\qquad}rcl}
\varsigma \in \mathit{State} & \triangleq & \mathit{Exp} \times \mathit{Env} \times \mathit{Store} \times \mathit{Share} & a \in \mathit{Addr} & \triangleq & \text{an infinite set} \\
\rho \in \mathit{Env} & \triangleq & \mathit{Var} \rightharpoonup \mathit{Addr} & d \in \mathit{Value} & \triangleq & \mathit{Lam} \times \mathit{Env} \\
\sigma \in \mathit{Store} & \triangleq & \mathit{Addr} \rightharpoonup \mathit{Value} & \mathit{sh} \in \mathit{Share} & \triangleq & \mathit{Addr} \rightharpoonup \mathcal{P}(\mathit{Var})
\end{array}$$

Fig. 2. Collecting-semantics domains

Function call transition

$(\rightsquigarrow_{\mathit{State}}) \subseteq \mathit{State} \times \mathit{State}$

$\langle (f\ \mathit{arg}_1\ \ldots\ \mathit{arg}_n), \rho, \sigma, \mathit{sh}\rangle \rightsquigarrow_{\mathit{State}} \langle e', \rho', \sigma', \mathit{sh}'\rangle$

$$\begin{array}{rcl}
\text{where } d_f, & = & \mathcal{A}\, f\, \rho\, \sigma \\
d_i & = & \mathcal{A}\, \mathit{arg}_i\, \rho\, \sigma \\
\langle (\lambda\ (x_1\ \ldots\ x_n)\ e'), \rho_\lambda\rangle & = & d_f \\
a_i & = & \mathit{fresh} \text{ addresses (all distinct)} \\
\rho' & = & \rho_\lambda[x_i \mapsto a_i \text{ for all } i] \\
\sigma' & = & \sigma[a_i \mapsto d_i \text{ for all } i] \\
\mathit{sh}' & = & \mathit{updateSharing}\ \mathit{sh}\ \rho\ f\ \{\langle \mathit{arg}_1, a_1\rangle, \ldots, \langle \mathit{arg}_n, a_n\rangle\}
\end{array}$$

Argument evaluation

$\mathcal{A} : \mathit{Arg} \times \mathit{Env} \times \mathit{Store} \rightharpoonup \mathit{Value}$

$\mathcal{A}\, x\, \rho\, \sigma \quad \triangleq \sigma(\rho\, x)$

$\mathcal{A}\, \mathit{lam}\, \rho\, \sigma \quad \triangleq \langle \mathit{lam}, \rho\rangle$

Sharing propagation

$\mathit{updateSharing} : \mathit{Share} \times \mathit{Env} \times \mathit{Arg} \times \mathcal{P}(\mathit{Arg} \times \mathit{Addr}) \rightharpoonup \mathit{Share}$

$\mathit{updateSharing}\ \mathit{sh}\ \rho\ f\ \{\ldots, \langle \mathit{arg}_i, a_i\rangle, \ldots\} = \mathit{sh}[a_i \mapsto \mathit{sharingBetween}\ \mathit{sh}\ \rho\ \{f, \mathit{arg}_i\} \text{ for all } i]$

$\mathit{sharingBetween} : \mathit{Share} \times \mathit{Env} \times [\mathit{Arg}]^2 \rightharpoonup \mathcal{P}(\mathit{Var})$

$$\begin{array}{ll}
\mathit{sharingBetween}\ \mathit{sh}\ \rho\ \{x, x'\} & = \mathit{sh}\ (\rho\ x)\ \cap\ \mathit{sh}\ (\rho\ x') \\
\mathit{sharingBetween}\ \mathit{sh}\ \rho\ \{\mathit{lam}, x\} & = \mathit{availVars}\ \mathit{lam}\ \cap\ \mathit{sh}\ (\rho\ x) \\
\mathit{sharingBetween}\ \mathit{sh}\ \rho\ \{\mathit{lam}, \mathit{lam}'\} & = \mathit{availVars}\ \mathit{lam}\ \cap\ \mathit{availVars}\ \mathit{lam}'
\end{array}$$

Fig. 3. Transition relation for the collecting semantics

The instrumentation and propagation that tracks witnesses abstracts perfectly well, which is why it provides a sound analysis. Thus, we establish the correctness property in Definition 3.1 as an *invariant* of the program state as it steps. We prove this invariant for the concrete semantics only, meaning the abstraction only needs to prove that it finds an over-approximation of the concrete semantics.

Collecting Semantics Transition. The basis of the collecting semantics is a small-step transition relation ($\rightsquigarrow_{\mathit{State}}$). The initial program state $\langle e_0, \bot, \bot, \bot\rangle$ consists of the top-level expression to be analyzed and empty environments, stores, and sharing information.

The transition relation is found in Figure 3 and consists of two main pieces: the first executes the program in a standard way amenable to abstraction *à la* Van Horn and Might [2010] and the second deals with the sharing information. For the standard execution, first the function and arguments are evaluated using the utility function $\mathcal{A}$, which either looks up the value of a variable via the environment and store or packages a function with the current environment into a closure, depending on the form of the function and argument. Then, the closure being called is deconstructed into its components, and new addresses are created for the new variable bindings. Following, the environment coming from the closure and the store are updated appropriately, mapping the variables to the addresses and the addresses to the argument values, respectively. Finally, the sharing information is updated, which is described below.

Updating Sharing. We utilize the interpretation of the sharing information as witnesses of equalities to construct new witnesses. To do this, we take advantage of three key properties. The first property is how information is initially produced, which all comes back to λ: a closure will always share its entire environment with the state that created it, which is *reflexivity*. Consider another program example:

```
let
  val h = fn (f', g') => g' (6, f')
  val n = 5
  val f = fn x => x + n
  val g = fn (x, f'') => x + n + f'' n
in
  h (f, g) + g (7, f)
end
```

We know that the closure for `f` is created in an environment with the binding for `n`, so the closure for `f` will share that binding with the environment that constructs the closure. After this information is initially produced, the remaining two properties describe how to propagate it correctly as the program executes.

The second property is *transitivity*: if an environment ρ (coming from a program state or a closure) shares a variable x with another environment ρ', and ρ' also shares x with yet another environment ρ'', then we know that x is shared between ρ and ρ''. This is the same as composing two witnesses of equality into a third via transitivity.

The third property is *preservation-through-calls*: if environment is shared between two variables f and x, and x is passed as an argument to f, then this sharing can be propagated to the variable bound by the function f refers to, since f's packaged environment becomes the current environment upon stepping into it.

In the example, we know that both of the closures for `f` and `g` share the binding of `n` with the environment that created their closures. At the two calls "`h (f, g) + g (7, f)`," we know that the closure for `f` shares the binding of `n` with the caller's environment and the closure for `g` shares the binding of `n` with the caller's environment. Therefore, we deduce that the closures for `f` and `g` share the binding of `n`.

The function used to update sharing information is named *updateSharing* (Figure 3). It takes as input the current known sharing information *sh*, the environment of the caller ρ, the current store σ, the function argument being called f, and the set of arguments paired with the addresses the argument values are flowing to. At a high level, when fresh addresses are allocated during an application, this function comes up with the new sharing information for every new binding, and updates *sh*. For each new binding $x_i \mapsto a_i$, it determines which bindings are shared between the environment that binds the x_i (the environment packaged by the value of f) and the environment for the values that x_i refers to (the value of arg_i). In other words, the sharing for each fresh a_i is the same as the sharing between f and arg_i, which is calculated by the function *sharingBetween*.

The function *sharingBetween* takes two elements that are either variables or λ-abstractions, and determines all of the bindings that are shared between the values those elements refer to. If the two elements are both variables, we apply transitivity and intersect their sharing sets: if each shares a variable y with the current context, then they must share it with each other. If one element is a λ-abstraction, the created closure value shares *everything* in scope with the current context, and then we intersect it with the sharing set of the other element, applying transitivity. If both arguments are λ-abstractions, then *availVars* yields the same set.

$$
\begin{array}{rcl@{\qquad}rcl}
\widetilde{\varsigma} \in \widetilde{State} & \triangleq & Exp \times \widetilde{Env} \times \widetilde{Store} \times \widetilde{Share} & \widetilde{a} \in \widetilde{Addr} & \triangleq & \text{a finite set} \\
\widetilde{\rho} \in \widetilde{Env} & \triangleq & Var \rightharpoonup \widetilde{Addr} & \widetilde{d} \in \widetilde{Value} & \triangleq & \mathcal{P}(Lam \times \widetilde{Env}) \\
\widetilde{\sigma} \in \widetilde{Store} & \triangleq & \widetilde{Addr} \rightharpoonup \widetilde{Value} & \widetilde{sh} \in \widetilde{Share} & \triangleq & \widetilde{Addr} \rightharpoonup \mathcal{P}(Var)
\end{array}
$$

Fig. 4. Abstract semantics domains

Function call transition

$$
\begin{array}{l}
(\rightsquigarrow_{\widetilde{State}}) \subseteq \widetilde{State} \times \widetilde{State} \\
\langle (f\ arg_1\ \ldots\ arg_n), \widetilde{\rho}, \widetilde{\sigma}, \widetilde{sh} \rangle \rightsquigarrow_{\widetilde{State}} \langle e', \widetilde{\rho}', \widetilde{\sigma}', \widetilde{sh}' \rangle \\
\text{where } \begin{array}[t]{lcl}
\widetilde{d}_f & = & \widetilde{\mathcal{A}}\, f\, \widetilde{\rho}\, \widetilde{\sigma} \\
\widetilde{d}_i & = & \widetilde{\mathcal{A}}\, arg_i\, \widetilde{\rho}\, \widetilde{\sigma} \\
\multicolumn{3}{l}{\langle (\lambda\ (x_1\ \ldots\ x_n)\ e'), \widetilde{\rho}_\lambda \rangle \ \in\ \widetilde{d}_f} \\
\widetilde{a}_i & = & \ldots \text{ abstract addresses for the new bindings } \ldots \\
\widetilde{\rho}' & = & \widetilde{\rho}_\lambda [x_i \mapsto \widetilde{a}_i \text{ for all } i], \\
\widetilde{\sigma}' & = & \sigma \sqcup [\widetilde{a}_i \mapsto \widetilde{d}_i \text{ for all } i] \\
\widetilde{sh}' & = & \widetilde{updateSharing}\ \widetilde{sh}\ \widetilde{\rho}\ f\ \{\langle arg_1, \widetilde{a}_1 \rangle, \ldots, \langle arg_n, \widetilde{a}_n \rangle\}
\end{array}
\end{array}
$$

Argument evaluation

$$
\begin{array}{l}
\widetilde{\mathcal{A}} : Arg \times \widetilde{Env} \times \widetilde{Store} \rightharpoonup \widetilde{Value} \\
\widetilde{\mathcal{A}}\ x\ \widetilde{\rho}\ \widetilde{\sigma} \quad\ \ \triangleq \widetilde{\sigma}(\widetilde{\rho}\ x) \\
\widetilde{\mathcal{A}}\ lam\ \widetilde{\rho}\ \widetilde{\sigma} \ \triangleq \{\langle lam, \widetilde{\rho} \rangle\}
\end{array}
$$

Sharing propagation

$$
\begin{array}{l}
\widetilde{updateSharing} : \widetilde{Share} \times \widetilde{Env} \times Arg \times \mathcal{P}(Arg \times \widetilde{Addr}) \rightharpoonup \widetilde{Share} \\
\widetilde{updateSharing}\ \widetilde{sh}\ \widetilde{\rho}\ f\ \{\ldots, \langle arg_i, \widetilde{a}_i \rangle, \ldots\} = \widetilde{sh} \sqcup [\widetilde{a}_i \mapsto \widetilde{sharingBetween}\ \widetilde{sh}\ \widetilde{\rho}\ \{f, arg_i\} \text{ for all } i]
\end{array}
$$

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\widetilde{sharingBetween} : \widetilde{Share} \times \widetilde{Env} \times [Arg]^2 \rightharpoonup \mathcal{P}(Var)} \\
\widetilde{sharingBetween}\ \widetilde{sh}\ \widetilde{\rho}\ \{x, x'\} & = & \widetilde{sh}\ (\widetilde{\rho}\ x)\ \cap\ \widetilde{sh}\ (\widetilde{\rho}\ x') \\
\widetilde{sharingBetween}\ \widetilde{sh}\ \widetilde{\rho}\ \{lam, x\} & = & availVars\ lam\ \cap\ \widetilde{sh}\ (\widetilde{\rho}\ x) \\
\widetilde{sharingBetween}\ \widetilde{sh}\ \widetilde{\rho}\ \{lam, lam'\} & = & availVars\ lam\ \cap\ availVars\ lam'
\end{array}
$$

Fig. 5. Transition relation for the abstract semantics

Correctness. If one reconstructs the sharing maps after each transition using the specification, the result is equivalent to that of the sharing propagation from the previous state. In other words, the definition of transition respects the sharing invariants. A proof of this property is mechanized in Rocq and we discuss our proof strategy in Section 3.4.

3.3 Abstract Semantics

The goal of the abstract semantics is to develop a computable approximation of the collecting semantics. By front-loading the complexity of the analysis (particularly sharing propagation) in the collecting semantics, the collecting semantics becomes straightforward to abstract. This approach yields a much simpler simulation argument for the abstract semantics.

For example, proving that abstract sharing propagation respects (*syn.* simulates) sharing propagation in the collecting semantics is straightforward compared to showing that abstract sharing propagation respects outright equality comparisons in the collecting semantics. Effectively, moving complexity into the concrete semantics decomposes the correctness proof into a simple proof of soundness, and a proof of the sharing invariants.

The abstract semantic domains are found in Figure 4. The abstract semantics are constructed by taking the collecting semantics and finitizing the address sets. Propagating this change through the other domains ensures that the entire state space is finite, and thus computable. A key change is that values abstract to sets of closures: since the address space is no longer infinite, addresses may be reused; instead of updating the store, the store is extended using a join operator combining the old and new values.

The particular choice of abstract addresses is not a crucial detail and does not affect correctness [Gilray et al. 2018], so we do not define it; when an address is needed for a binding we simply assume we have a function that allocates an address. A common choice is 0-CFA [Shivers 1991], which takes $\widetilde{Addr} = Var$ with the address of a binding being the same as the variable, while other choices, such as 1-CFA or the stack-precise choice described in Gilray et al. [2016], could result in better precision at the cost of time-complexity.

Abstract Semantics Transition. The abstract transition relation is found in Figure 5. Argument evaluation remains the same in the abstraction: $\widetilde{\mathcal{A}}$ evaluates variables using the (abstract) environment and store, and λ-abstractions are evaluated to closure values (in the abstract semantics, a singleton set). When calling a function, the function and arguments are first evaluated. Following this, since finitization means there may be multiple function values, the abstract state chooses one of the closures associated with the function to transition to (when taking a fixpoint, all choices are explored). Next, abstract addresses for the new bindings are allocated, the environment updated, and the store extended. The extension of the abstract store uses a join operator $\sqcup$: if a new address does not exist in the store, it is added as normal; if the address exists already, then the two closure sets of the two stores are combined, to preserve over-approximation.

$$(\widetilde{\sigma} \sqcup \widetilde{\sigma}')\, \widetilde{a} = \begin{cases} \widetilde{\sigma}\, \widetilde{a} & \text{if } \widetilde{a} \in \mathrm{dom}(\widetilde{\sigma}) \land \widetilde{a} \notin \mathrm{dom}(\widetilde{\sigma}') \\ \widetilde{\sigma}'\, \widetilde{a} & \text{if } \widetilde{a} \notin \mathrm{dom}(\widetilde{\sigma}) \land \widetilde{a} \in \mathrm{dom}(\widetilde{\sigma}') \\ \widetilde{\sigma}\, \widetilde{a} \cup \widetilde{\sigma}'\, \widetilde{a} & \text{otherwise.} \end{cases}$$

Finally, the sharing information is updated using $\widetilde{updateSharing}$, which is nearly identical to its concrete counterpart. This is because the key properties of sharing information (closures always sharing with the environment that creates them, transitivity, and preserving sharing through flows) all abstract well, meaning that these properties correctly act on abstract sharing information in a way that preserves the soundness between the collecting and abstract semantics.

The key difference lies in how sharing information is combined, which is a result of finitizing the address sets. Just like in stores, information on addresses may already exist in the sharing maps, so it must be combined with a join operator. In the case of sharing information, the join operator intersects the sets of variables that appear in both maps:

$$(\widetilde{sh} \sqcup \widetilde{sh}')\, \widetilde{a} = \begin{cases} \widetilde{sh}\, \widetilde{a} & \text{if } \widetilde{a} \in \mathrm{dom}(\widetilde{sh}) \land \widetilde{a} \notin \mathrm{dom}(\widetilde{sh}') \\ \widetilde{sh}'\, \widetilde{a} & \text{if } \widetilde{a} \notin \mathrm{dom}(\widetilde{sh}) \land \widetilde{a} \in \mathrm{dom}(\widetilde{sh}') \\ \widetilde{sh}\, \widetilde{a} \cap \widetilde{sh}'\, \widetilde{a} & \text{otherwise.} \end{cases}$$

The reason for using intersections is that the information lattice is "upside down" — for sharing, a smaller set of variables means less precise information; *e.g.*, the abstraction cannot find more sharing than the concrete. This contrasts with abstract values, where a larger set means less precise information. Using this join operator preserves soundness of the abstract semantics, which we discuss in the following subsection.

Soundness. To prove the soundness of the analysis, it suffices to show that the abstraction *simulates* the collecting semantics; together with the fact that the witnesses produced by the collecting semantics are valid, we show that the sharing witnesses gathered from the abstraction is a sound approximation of the true runtime behavior [Cousot and Cousot 1977]. We formalized and verified the simulation relation in Rocq.

The simulation relation utilizes an *abstraction map* [Might and Manolios 2008], $\alpha : \mathit{Addr} \rightharpoonup \widetilde{\mathit{Addr}}$, that maps addresses from the collecting semantics to abstract addresses. Given such a map, we can define what it means for a state in the collecting semantics to be appropriately abstracted (*syn.* simulated) by an abstract state, written as $\varsigma \lesssim_\alpha \widetilde{\varsigma}$. A stepwise simulation lemma can be stated as follows: if a collecting-semantics state steps to a result state, a corresponding abstract state can always step to a corresponding result state.

We define the simulation relation formally.

Definition 3.2 (Simulation)**.** Simulation is defined between every component of the semantics.

$$\langle e, \rho, \sigma, sh\rangle \lesssim_\alpha \langle e, \widetilde{\rho}, \widetilde{\sigma}, \widetilde{sh}\rangle \triangleq (\rho \lesssim_\alpha \widetilde{\rho}\,) \wedge (\sigma \lesssim_\alpha \widetilde{\sigma}\,) \wedge \left(sh \lesssim_\alpha \widetilde{sh}\right)$$

$$\begin{array}{lll} \text{where } \rho \lesssim_\alpha \widetilde{\rho} & \triangleq & \mathrm{dom}(\rho) = \mathrm{dom}(\widetilde{\rho}\,) \wedge \forall x\ a\ \widetilde{a} \,.\, (\rho\, x = a \implies \widetilde{\rho}\, x = \widetilde{a} \implies \alpha\, a = \widetilde{a}) \\ \sigma \lesssim_\alpha \widetilde{\sigma} & \triangleq & \forall a\ d \,.\, \left(\sigma\, a = d \implies \exists\, \widetilde{d} \,.\, \widetilde{\sigma}\,(\alpha\, a) = \widetilde{d} \implies d \lesssim_\alpha \widetilde{d}\right) \\ \langle lam, \rho\rangle \lesssim_\alpha \widetilde{d} & \triangleq & \exists\, \widetilde{\rho} \,.\, \langle lam, \widetilde{\rho}\rangle \in \widetilde{d} \wedge \rho \lesssim_\alpha \widetilde{\rho} \\ sh \lesssim_\alpha \widetilde{sh} & \triangleq & \forall a\ x \,.\, \left(x \in \widetilde{sh}\,(\alpha\, a) \implies x \in sh\, a\right) \end{array}$$

For most components of a state, the simulation relation $\lesssim_\alpha$ translates a concrete address to an abstract address straightforwardly using the abstraction map α: simulation for environments occus if they have the same domain and map to related addresses. Simulation for stores occurs if every value binding in the concrete store is simulated by the corresponding binding in the abstract store; simulation for closure values means that the abstraction of the closure is in the value set. For sharing information, the simulation relation means that the bindings shared between some pair of concrete addresses must be a superset of the bindings shared between the abstract counterparts of the addresses; the abstract semantics cannot find that more sharing occurs than the collecting semantics does.

Abstractions often look like the opposite: if some behavior occurs in the concrete transition, it must be observed in the abstract. The abstraction for sharing information occurs in the opposite direction because the lattice is upside-down, as described above with join being intersection. From another perspective, the sharing information lattice describes a *must* or *always* property (these bindings are *always* shared) as opposed to a *may* property (*e.g.*, a closure for a particular λ *may* flow to this variable).

3.4 Proofs

We have developed a mechanized proof of the correctness of our analysis in the Rocq theorem prover (the proofs are included as an accompanying artifact). In this section, we give an overview of these proofs. The overall correctness of the analysis consists of the two statements: that sharing propagation in the collecting semantics preserves the sharing invariants (Definition 3.1), and that if a concrete state transitions to another, any abstract state that simulates it (Definition 3.2) also transitions to a state that simulates the result.

These statements alone lack the necessary invariants to be proven directly. Instead, we prove a stronger statement: we assume that the starting state satisfies additional well-formedness conditions and show that these conditions are also preserved by the transition. In effect, we strengthen the premises in the inductive proof. Because the two statements rely on the properties of different parts of a state, we define two well-formedness properties to support each case.

Definition 3.3 (Sharing Well-formedness). Well-formedness holds when (1) all addresses appearing in all reachable environments are present in the store (*i.e.*, no "dangling pointers") and (2) the store and the sharing map have identical domains. This correspondence is particularly useful when we reason about fresh addresses allocated in a transition. In particular, when we obtain a fresh address from the store, the second statement immediately implies that it is not already present in *sh*:

$$\begin{aligned}&wf_{shr}\,\langle \_, \rho, \sigma, sh\rangle \triangleq (\forall \rho \in envs \,.\, \mathrm{im}(\rho) \subseteq \mathrm{dom}(\sigma)) \wedge (\mathrm{dom}(\sigma) = \mathrm{dom}(sh))\\ &\qquad \text{where } envs = \{\rho\} \cup \{\rho' \mid \langle lam, \rho'\rangle \in \mathrm{im}(\sigma)\}\end{aligned}$$

Using this well-formedness property, we state the sharing-invariant theorem below.

Theorem 3.4 (Sharing Invariants Preserved). For all concrete states ς and ς',

$$validShare\,\varsigma \wedge wf_{shr}\,\varsigma \wedge \varsigma \rightsquigarrow_{State} \varsigma' \implies validShare\,\varsigma' \wedge wf_{shr}\,\varsigma'$$

The transition relation preserves sharing invariants and the well-formedness conditions stated above. This theorem is proven via transitivity and preservation as outlined in the previous section. Specifically, for addresses that are not bound in this transition, we show that *updateSharing* does not affect those addresses, and thus the sharing information is still valid. For any newly allocated address, its sharing information is computed by *sharingBetween*, which either removes elements from valid sharing sets or constructs a new set by including variables from the current scope. In both cases, the resulting sharing sets remain valid.

For simulation, we use a separate well-formedness property.

Definition 3.5 (Simulation Well-formedness). Simulation well-formedness holds when (1) the sharing map and the abstraction map have the same domain, (2) the state has no dangling pointers, and (3) that all concrete environments are injective. Injectivity establishes the fact that adding fresh addresses to the abstract map does not interfere with existing bindings:

$$\begin{aligned}&wf_{sim}\,\alpha\,\langle \_, \rho, \sigma, sh\rangle \triangleq\\ &\quad (\mathrm{dom}(\alpha) = \mathrm{dom}(sh)) \wedge (\forall \rho \in envs \,.\, \mathrm{im}(\rho) \subseteq \mathrm{dom}(\sigma)) \wedge (\forall \rho \in envs \,.\, \rho\, x = \rho\, x' \implies x = x')\\ &\quad \text{where } envs = \{\rho\} \cup \{\rho' \mid \langle lam, \rho'\rangle \in \mathrm{im}(\sigma)\}\end{aligned}$$

Now, we can define our simulation theorem.

Theorem 3.6 (Simulation). For all concrete states ς and ς', and abstract states $\widetilde{\varsigma}$, if

$$(\varsigma \lesssim_{\alpha} \widetilde{\varsigma}) \wedge (wf_{sim}\,\alpha\,\varsigma) \wedge (\varsigma \rightsquigarrow_{State} \varsigma')$$

then there exists an abstract state $\widetilde{\varsigma}'$ and an updated abstract map α' such that

$$(\widetilde{\varsigma} \rightsquigarrow_{\widetilde{State}} \widetilde{\varsigma}') \wedge (\varsigma' \lesssim_{\alpha'} \widetilde{\varsigma}') \wedge (wf_{sim}\,\alpha'\,\varsigma')$$

The simulation theorem establishes that for any simulating states, ς and $\widetilde{\varsigma}$, if ς is well-formed and steps to a resulting state ς', then $\widetilde{\varsigma}$ can always take a step resulting in $\widetilde{\varsigma}'$, and $\widetilde{\varsigma}'$ simulates ς'; furthermore, the well-formedness is preserved during the transition. To show simulation after a transition, we first construct an extended abstract map α' by mapping the newly allocated concrete addresses to corresponding abstract addresses. The remainder of the simulation proof proceeds by unfolding definitions through the abstract transition.

Since every transition in the collecting semantics is simulated by a corresponding abstract transition, together with a trivial property that the initial abstract state simulates the initial concrete state (also proven in Rocq), it follows by induction that for every reachable concrete state, there is a reachable abstract state that simulates it. Therefore, the sharing information identified by all reachable abstract states over-approximates the sharing information collected by the concrete semantics. Complete proofs of Theorem 3.4 and Theorem 3.6 are provided in our Rocq mechanization.

3.5 Extracting Analysis Results

From the sharing information $\widetilde{sh}$, which tracks *address* information, we can extract sharing information for *variables* into the analysis result, $\mathcal{F}_V : \mathit{Var} \to \mathcal{P}(\mathit{Var})$: given a variable x, the sharing that *always* occurs between the values that flow to x and the environment that binds x is the intersection of the sharing information over all addresses x is bound to, or equivalently looking at state $(\_, \widetilde{\rho}, \_, \widetilde{sh})$ reachable from the initial $\widetilde{\varsigma}_0$ and examining the sharing for that state for x:

$$\mathcal{F}_V(x) = \bigcap \left\{ \widetilde{sh}(\widetilde{\rho}(x)) \mid \widetilde{\varsigma}_0 \rightsquigarrow^*_{\widetilde{State}} (\_, \widetilde{\rho}, \_, \widetilde{sh}) \text{ and } x \in \operatorname{dom}(\widetilde{\rho}) \right\}$$

This result is an over-approximation of every reachable abstract state, and thus it is an over-approximation of every single concrete state — no matter the state, this sharing information is valid.

In Section 5, we use the $\mathcal{F}_V$ information to make decisions on how to transform the program to take advantage of caller-provided environments.

3.6 Store-Widening and Complexity Analysis

The analysis maintains a per-state value store $\widetilde{\sigma}$ and sharing information $\widetilde{sh}$. *Store-widening* is the process of making the store global, shared and updated by all states, which makes the size of the state-space exponentially smaller and more feasible to compute. Similarly, since we already inspect and intersect each abstract state's sh when extracting the analysis results, we can make the sharing information global as well. This transformations the state-space lattice:

$$\mathcal{P}(\mathit{Exp} \times \widetilde{\mathit{Env}} \times \widetilde{\mathit{Store}} \times \widetilde{\mathit{Share}}) \rightsquigarrow \mathcal{P}(\mathit{Exp} \times \widetilde{\mathit{Env}}) \times \widetilde{\mathit{Store}} \times \widetilde{\mathit{Share}}$$

To compute a fixpoint in the widened state-space lattice, we transition each configuration $\langle e, \widetilde{\rho} \rangle$ under the global store $\widetilde{\sigma}$ and sharing information $\widetilde{sh}$. This results in (1) new pairs $\langle e', \widetilde{\rho}' \rangle$ that are added to the accumulated set, (2) updates to the store, which are included into the global store, and (3) updates to the sharing information, which are included into the global sharing information. This process continues until a fixpoint is reached.

Using n as the size of the program, the standard worst-case complexity for monovariant store-widened 0-CFA (where abstract environments are completely determined for each expression e and function) is $O(n^3)$ [Gilray et al. 2016]. This property follows from the fact that the store's size is bounded by $O(n^2)$ and the worst case occurs when bindings in the store are found one at a time, each time visiting a different one of $O(n)$ configurations.

By a similar argument, the addition of sharing information does not increase the complexity of the underlying analysis: the sharing information has height $O(n^2)$ like the store, and worst-case occurs when visiting every configuration, for an upper bound of $O(n^3)$. In practice, we do not observe cubic behavior in the calculation of the sharing information: the analysis time seems to increase linearly with program size for example programs. The analysis cost can be made even cheaper by only tracking variables that are free in *at least* one function, as variables that are never free would never be provided.

Extending the Analysis. The presented analysis can be extended to track additional equalities. For example, our mechanization additionally tracks and proves sharing between pairs of addresses, proving that the closures referred to by those addresses will always share bindings. In practice, we did not find that this additional algorithmic complexity resulted in more precise sharing information.

One could also generalize the sharing to relate *pairs* of variables, where we track equality between a variable in the current environment with a *different* variable in the environment of a closure (or, different variables in different closures). This mechanism can be used to implement higher-order rematerialization [Might 2010]. For example, at a call $(f\ arg\ x)$, if x is a variable and arg shares x with the current environment, then when calling f and binding a new variables y and x', we know the values associated with y (*i.e.*, the values of arg) package x, which refers to the same value of x' in the new current environment.

Finally, we believe it is possible to use a different underlying base analysis. Our analysis is developed on top of a finite-state machine (all k-CFAs are finite-state machines), but pushdown analyses corresponding to pushdown automata typically provide better precision [Darais et al. 2017; Earl et al. 2012; Gilray et al. 2016; Van Horn and Might 2010; Vardoulakis and Shivers 2011; Wei et al. 2018]. We believe our general approach of deriving proofs of equalities is amenable to such pushdown analyses, as the three properties we use — reflexivity, transitivity, and propagation through calls — are still available.

4 Implementation and Evaluation of the Analysis

To evaluate the effectiveness of our analysis, we implemented the environment-sharing analysis in a prototype that supports a substantial subset of Standard ML.[6] The implementation parses and type-checks the source program, producing an ANF-style intermediate representation (IR). The compiler then performs several simple, syntax-based optimizations on this IR, such as argument flattening, inlining, constant folding, uncurrying, and dead code elimination. Afterward, the program is converted to a full-featured CPS IR. During this conversion, we remember whether a CPS function was produced from a source language function (a *user* function), or if it was introduced to handle control flow (a *continuation* function). Without call-with-current-continuation (call/cc) or other control primitives, continuation functions can easily store their environments on a stack. Since most compilers, including CPS-based compilers, use a stack to manage calls and returns, the environments of the continuation functions do not require explicit management. In the discussion below, we only compare environment-sharing results of user functions.

We present the semantics of our analysis in Section 3 as if the discovery of possible program states and the sharing propagation take place simultaneously. In our implementation, we use store widening (maintaining one global abstract store for all abstract states) and additionally separate the computation of the program state-space from the pass that performs the environment-sharing analysis, and thus only propagate sharing information across an already-computed state-space. Our analysis computes the abstract state-space using the incremental approach of Quiring and Van Horn [2024], propagating changes to the closure sets associated with addresses through previously explored abstract states. To ensure the smallest number of abstract states are re-explored due to an update, each abstract address is associated with the set of explored abstract states that depends on it (*i.e.*, looks it up in the global abstract store). Additionally, the abstract interpretation tracks typing information to filter impossible execution paths [Fluet 2013]. Our implementation of the sharing analysis uses a naïve fixpoint to iteratively narrow the valid sharing.

As we discuss in Section 5, there are many possible optimizations that utilize the data obtained from our analysis. In addition, there are many other variables (*e.g.*, the quality of code generation and the design of the garbage collector) that affect the results. It is hard to isolate the quality of our analysis from the numerous decisions made in a particular transformation with an end-to-end performance evaluation. Instead, we present counting results since they are repeatable and specific to the analyses being compared. We believe the best way to evaluate the quality of an analysis is through these detailed counting statistics.

[6]Our implementation handles most of the SML core language, including references, recursion, datatypes, and exceptions, and supports SML structures (modules), but not signatures or functors.

Table 1. Summary of benchmarks used to evaluate the analysis

Program	**LOC**	**# λs**	**Description**
nucleic	3326	69 (38)	The SML version of the "Pseudoknot" program [Hartel et al. 1996].
boyer	819	22 (16)	The Boyer-Moore benchmark from SML/NJ.
k-cfa	591	77 (67)	A SML port of Matt Might's k-CFA reference implementation.
ratio-regions	545	90 (60)	An image segmentation benchmark from MLton.
mc-ray	487	35 (26)	A SML port of the "Weekend Raytracer" [Shirley 2020].
knuth-bendix	436	99 (71)	The Knuth-Bendix benchmark from SML/NJ.
raytracer	335	32 (30)	The Ray tracer from the Impala benchmarks.
s-n-f	291	29 (21)	The Smith-Normal-Form benchmark from MLton.
cps-convert	210	22 (13)	Danvy-Filinski CPS conversion [Danvy and Filinski 1992].
parser-comb	185	83 (25)	Using parser combinators to parse a simple expression syntax.
json-decode	183	38 (12)	JSON decoding using Elm-style combinators [Elm 2021].
interpreter	165	17 (13)	An interpreter for a simple language.
twenty-four	162	35 (16)	A backtracking solver for the 24 puzzle[7] written in CPS.
derivative	123	12 (8)	A SML port of the symbolic derivation benchmark from Larceny.
life	122	31 (16)	The Life benchmark from SML/NJ.
streams	117	11 (3)	A program that manipulates and combines push and pull streams.
tardis	104	40 (7)	A SML port of the example in the Haskell tardis package.

Our analysis computes what variable bindings are common between the environment of a call site and the environments of the closures it invokes. To evaluate the analysis, we propose the following metric:

For a function, how many of its free variables can be provided at all call sites?

Ordinarily, a closure allocates a record on the heap storing the bindings of all its free variables. If a free variable can be provided at all call sites, the compiler may choose to exclude that variable from the closure. As the analysis identifies more such variables, the landscape of safe sharing opportunities expands, providing greater flexibility for the transformation.

To contextualize our results among benchmark programs in different sizes and styles, we implement a baseline analysis that relies solely on syntactic properties to discover the same sharing facts. The syntactic analysis works as follows. If a variable f is syntactically bound to a function and all uses of f are function applications (*i.e.*, f is used in a first-order way), then all free variables of f can be provided at all the call sites. All other functions are simply marked as unknown; no variables can be shared at any call sites potentially calling an unknown function. The syntactic analysis provides a good baseline since if we think of a higher-order program as consisting of a first-order "skeleton," on top of which some higher-order "ornaments" reside, the syntactic analysis isolates the ornaments from the skeleton, focusing the evaluation on functions with non-obvious behaviors. In addition, the syntactic approach is found in a number of compilers [Appel 1992; Keep et al. 2012] and serves as a representative for such systems.

As another point of comparison, we have also implemented lightweight-closure conversion (LWCC) [Steckler and Wand 1997], which performs an analysis that can also be used to enable caller provided environment. To our knowledge, this is the first empirical evaluation of lightweight-closure conversion. This implementation is an improvement over the original formulation, as it uses the same control-flow analysis as our environment sharing analysis. Our environment sharing analysis and our LWCC implementation both perform type-based filtering and dead-code elimination, which are not done in the original LWCC paper. We discuss LWCC further in Section 7.

Table 2. Summary of results for the benchmarks

Program	**#FV**	**Syn.**	**Env. Sh.**	**LWCC**	**Interp.**	**E.S. / S.**	**E.S. / I.**	**Time (ms)**
nucleic	124	49	120	75	120	2.45	1.00	14 + 8
boyer	30	17	27	23	27	1.59	1.00	11 + 8
k-cfa	189	169	182	176	112 + 70	1.08	1.00	5 + 5
ratio-regions	478	311	447	334	447	1.44	1.00	4 + 4
mc-ray	92	63	83	79	24 + 68	1.32	0.90	1 + 1
knuth-bendix	304	239	279	251	273 + 6	1.17	1.00	4 + 4
raytracer	76	75	76	75	76	1.01	1.00	5 + 1
s-n-f	106	82	89	87	89	1.09	1.00	1 + 4
cps-convert	50	18	23	23	23	1.28	1.00	1 + 1
parser-comb	150	26	93	50	93	3.58	1.00	2 + 2
json-decode	80	11	54	26	54	4.91	1.00	1 + 1
interpreter	26	19	22	19	21 + 1	1.16	1.00	1 + 1
twenty-four	70	16	38	31	38	2.38	1.00	1 + 1
life	53	26	41	27	41	1.58	1.00	3 + 1
streams	10	3	6	4	6	2.00	1.00	1 + 0
derivative	11	9	11	10	11	1.22	1.00	1 + 1
tardis	99	8	50	41	57	6.25	0.88	1 + 1

Additionally, we use an instrumented interpreter based on the collecting semantics to gather precise sharing results. At a call site, when the interpreter transitions from the environment of the caller to the environment of the callee, the instrumentation gathers what bindings in the two environments are shared. As the interpreter only collects data from one specific execution trace, it is an *under-approximation* of the true sharing facts; *i.e.*, it does not guarantee soundness over all possible executions. This information nonetheless provides a useful *upper-bound* for how much the analysis precision could be improved — if the numbers are close, a more precise analysis, such as 1-CFA, is unlikely to discover significantly more sharing opportunities in most cases.

Table 1 lists the benchmark programs used in the evaluation in descending order of their sizes. For each program, we show the number of lines of code, the number of user functions after CPS conversion and optimizations, and a short summary of its content. In each program, the numbers of user functions that are syntactically known is shown in parentheses. The lower the number of syntactically known functions is, the more "higher-order" the benchmark program is.

Table 2 summarizes the results, presenting the data for a syntactic baseline (**Syn.**), the analysis presented in this paper (**Env. Sh.**), an implementation of lightweight-closure conversion's analysis (**LWCC**), and the results of running an interpreter (**Interp.**). For each program, we show the total number of free variables (#**FV**), calculated by summing the number of free variables in each function. Each column under an analysis's name shows the total number of providable free variables, that is, free variables of a function group that can be provided at all call sites of the function based on the facts discovered by the respective analysis. The closer the number of providable free variables is to the total number of free variables, the fewer variables an average closure needs to package.[8] Since the interpreter may not visit all functions in a program, the interpreter's column sometimes shows two numbers: the number of providable free variables of visited functions, and the number

[7]Given n numbers, the objective of the 24 puzzle is to find an arithmetic expression that uses all n numbers exactly once and evaluates to 24.

[8]More precisely: the more opportunities that a decision procedure has available to reduce the size of a closure based on these facts, the fewer variables an average closure needs to package.

of free variables of unvisited functions. The sum of the two numbers provides an upper-bound of the analysis, assuming that all free variables can be provided for a function that the interpreter did not call. Finally, the columns labeled "**E.S. / S.**" and "**E.S. / I.**" include the factor of improvement of our analysis compared to the syntactic analysis and interpreter results respectively. (*i.e.*, the **Env. Sh.** column divided by the **Syn.** and **Interp.** columns). Because the interpreter result only represents a single execution trace, it cannot perfectly describe the execution of all programs. The comparison with the interpreter gives an unreachable upper-bound for the analysis.

For programs that primarily include first-order functions, the syntactic analysis provides a cost-effective approximation of necessary facts for the decision process. As higher-order functions become more prevalent in the input program, the gap between our analysis and the syntactic analysis widens. For instance, in the `tardis` program, the analysis discovers 6 times as many providable free variables as the syntactic analysis. In most cases, our analysis improves significantly over the syntactic baseline.

While LWCC uncovers certain sharing opportunities beyond what the syntactic approach can identify, as the program exhibits more higher-order behaviors, our analysis reveals more additional opportunities. For the `parser-comb` program, as an example, while our analysis identifies all variables that are safe to provide, the lightweight closure conversion misses a significant portion of these sharing opportunities. Our analysis often does better than LWCC, although for a few benchmarks, LWCC finds more variables than our analysis does. LWCC never performs strictly better than our analysis — in the cases where it finds more variables in total, there are other variables that the environment sharing analysis discovers that are not detected by LWCC.

The last column (**Time**) shows the time our analysis takes in milliseconds; the number on the left shows the time for state-space discovery and the one on the right shows sharing information propagation. These numbers are the average wall-clock times of 10 runs and were collected on a MacBook M3 Air using MLton (version 20241230) to compile our implementation. This data demonstrates the scalability of our analysis; in the worst case, a 3300+ line program takes less than 0.03 seconds to analyze.

Our analysis is close to "optimal," for our benchmark programs. Even for the higher-order ones, every single fact found by the interpreter is also found by the analysis. For these programs, a more precise sharing analysis, or a more precise underlying CFA, would not yield more facts.

It is worth reiterating that the analysis only determines what is sound to do, not what must happen. The evaluation of our analysis also does not factor in the web constraint at a call site — if a call site calls several potential callees, even though the analysis concludes that for each of the callees, the call site can provide some of its environment, a sound transformation requires all callees to agree on a list of provided variables. In Section 5, we outline two policies to decide when to provide environments and align the calling convention for all callees in a web, as well as a transformation that executes the decision. We provide an evaluation of our proposed strategies in Section 6.

5 Applications

This paper is focused on our analysis; it is about discovering precise facts about environment sharing. To motivate the usefulness of this information, we now describe and evaluate a potential compiler optimization based on this information. In addition, we discuss several other applications of the information that we extract from our analysis. As we discuss below, these optimizations depend on the results of our sharing analysis and on the call-graph that we can extract from the underlying 0-CFA.

5.1 Replacing Closures with Arguments

The optimization is to replace closure allocations with function arguments; it is a more precise version of lightweight closure conversion (LWCC) [Siskind 1999; Steckler and Wand 1997].[9] If we know all the call sites for a given function, often times the free variables of that function can be provided as a function parameter instead of through an allocated closure. The knowledge required before this rewrite can occur is that the variable binding corresponding to the definition site is available at the call site, which is answered by the environment sharing analysis. Before a compiler can perform this transformation, it must ensure that the modification would not break any program invariants, and it needs to be confident that it is not de-optimizing the program.

In higher-order languages, the same call site may call to multiple different program locations at runtime. Therefore each of those functions must agree on a calling convention, which includes arguments as well as caller-provided environments. This is not the entire story: those same functions can be called from other call sites and thus agree with those as well. This process can continue, as each of those call sites may themselves have additional functions, and every call site and function must have consistent handling of arguments and environments. This interdependence between function definitions and calls is captured by the notion of a *call web* [Quiring et al. 2025].[10] The call web for a given function or call consists of all the other functions or calls that must agree on calling convention (what Serrano [1995] defines as the $\mathcal{T}$ property).

A call webs must over-approximate the runtime program behavior to be useful for compiler optimization. We represent these call webs with the following definition:

Definition 5.1. The function $web : (Exp \cup Lam) \to \mathcal{P}(Exp \cup Lam)$ maps a λ or *call* (using ℓ to denote either) to a set of λs and *calls*, subject to:

- $\ell \in web(\ell)$,
- *call* possibly invokes *lam* at runtime $\implies$ $lam \in web(call) \wedge call \in web(lam)$, and
- for all ℓ, ℓ' we have $\ell' \in web(\ell) \implies web(\ell) = web(\ell')$.

A *web* function is a sound over-approximation of runtime behavior, and will be used to encompass the program elements that must be transformed consistently. This information can be easily extracted from many CFAs [Serrano 1995; Shivers 1991], including our abstract interpretation.

We now lift the results of the sharing analysis $\mathcal{F}_V : Var \to \mathcal{P}(Var)$ to the function/call level, which we use to ensure that transformations respects the analysis: environments cannot be provided by the caller if its binding does not match that of the callee.

Definition 5.2. The map $\mathcal{F} : (Exp \cup Lam) \to \mathcal{P}(Var)$ describes the sharing available on a per-call and per-function basis. We use the syntax $X \cap Exp$ to select the expressions from $X \subseteq (Exp \cup Lam)$.

$$\mathcal{F}(\ell) = \begin{cases} availVars(f) & \text{if } \ell = (f\ arg_1\ \dots\ arg_n) \text{ and } f \in Lam \\ \mathcal{F}_V(f) & \text{if } \ell = (f\ arg_1\ \dots\ arg_n) \text{ and } f \in Var \\ \bigcap \{\mathcal{F}(call) \mid call \in web(\ell) \cap Exp\} & \text{if } \ell \in Lam \end{cases}$$

For each call *call*, $\mathcal{F}(call)$ is the set of bindings that available to share with the callee and for each function *lam*, $\mathcal{F}(lam)$ is the set of bindings that are always available to be shared at all of its calls.

To illustrate the above definitions, Figure 6 presents two small examples that are modifications of Listing 1. For each example, the code is on the left and includes labels to identify the subterms, while the web and $\mathcal{F}$ information are given on the right. In the first example, the free variable `n` of `f1` and `f2` is available at the call sites and can be caller-provided. In the second example, however, `n` cannot be shared because it is not available at C_1.

[9]We compare our optimization to the prior work on LWCC in Section 7.
[10]The term "web" was coined by Muchnick [1997].

```
let
  val n  = 5
  val f1 = [F1] fn x => x + n
  val f2 = [F2] fn y => y * n
  val h  = [F3] fn g => [C1] g 5
in
  [C2] (h f1) + [C3] (h f2)
end
```

$$\begin{tikzcd}
& \mathcal{F}(C_1) = \{\mathtt{n}\} \arrow[dl, dash] \arrow[dr, dash] & \\
\mathcal{F}(F_1) = \{\mathtt{n}\} & & \mathcal{F}(F_2) = \{\mathtt{n}\} \\
\mathcal{F}(C_2) = \{\mathtt{n}\} \arrow[dr, dash] & & \mathcal{F}(C_3) = \{\mathtt{n}\} \arrow[dl, dash] \\
& \mathcal{F}(F_3) = \{\mathtt{n}\} &
\end{tikzcd}$$

```
let
  val h1 = [F1] fn g1 => [C1] g1 5
  val n  = 5
  val f  = [F2] fn x => x + n
  val h2 = [F3] fn g2 => [C2] g2 6
in
  [C3] (h1 f) + [C4] (h2 f)
end
```

$$\begin{tikzcd}
\mathcal{F}(C_1) = \{\} \arrow[dr, dash] & & \mathcal{F}(C_2) = \{\mathtt{n}\} \arrow[dl, dash] \\
& \mathcal{F}(F_2) = \{\} &
\end{tikzcd}$$

$$\begin{tikzcd}
\mathcal{F}(C_3) = \{\mathtt{n}\} \arrow[d, dash] & \mathcal{F}(C_4) = \{\mathtt{n}\} \arrow[d, dash] \\
\mathcal{F}(F_1) = \{\} & \mathcal{F}(F_3) = \{\mathtt{n}\}
\end{tikzcd}$$

Fig. 6. Two examples with their corresponding call-web and $\mathcal{F}$ definitions

To describe the possible changes that an optimization will make on a program, we formally define the notation of a *decision policy.* A decision defines the additional variables that are passed by a given call. Decisions about caller-provided environment are defined as the following:

Definition 5.3. A *decision policy* for environment sharing is a function $\mathcal{D} : (Exp \cup Lam) \to \mathcal{P}(Var)$ mapping every call and function definition to a set of variables that can be provided by the caller (the set of variables is the environment-sharing *decision* for the term).

We constrain the policy $\mathcal{D}$ by the sharing analysis $\mathcal{F}$ (the decision made should be semantically sound) and the web constraints (the decision made should be uniform for the web).

Definition 5.4. An environment-sharing decision policy $\mathcal{D}$ is *valid* if it

- respects the environment sharing information: $\mathcal{D}(\ell) \subseteq \mathcal{F}(\ell)$, and
- is consistent across call webs: $\ell' \in web(\ell) \implies \mathcal{D}(\ell) = \mathcal{D}(\ell')$.

Given a valid policy $\mathcal{D}$, we are now able to define a whole-program transformation $\mathcal{T}[\![\cdot]\!]$ that passes the specified variables as additional function parameters.

Definition 5.5. The transformation $\mathcal{T}$ operates recursively over the term structure, adding the variables for the decisions as arguments and new parameters (shadowing the previous free variables).

$$\begin{aligned}
\mathcal{T}[\![\ell = (f\ arg_1\ \dots\ arg_n)]\!] &= (\mathcal{T}[\![f]\!]\ \mathcal{T}[\![arg_1]\!] \dots \mathcal{T}[\![arg_n]\!]\ y_1\ \dots\ y_k) && \mathcal{D}(\ell) = \{y_1, \dots, y_k\} \\
\mathcal{T}[\![\ell = (\lambda\ (x_1, \dots, x_n)\ e)]\!] &= (\lambda\ (x_1\ \dots\ x_n\ y_1\ \dots\ y_k)\ \mathcal{T}[\![e]\!]) && \mathcal{D}(\ell) = \{y_1, \dots, y_k\} \\
\mathcal{T}[\![x]\!] &= x
\end{aligned}$$

We assume an ordering on variables so that the arguments and parameters line up properly.

We can now define a naïve, yet valid decision policy that always passes environment from caller to callee, only limited by the requirements that the environment is available and shares the binding.

Definition 5.6. The *maximal* decision policy is defined as follows:

$$\mathcal{D}_{max}(\ell) = \bigcap \{\mathcal{F}(call) \mid call \in web(\ell) \cap Exp\}$$

This maximal decision policy has some major issues. The first is that it does not consider whether the environments provided from caller to callee are ever used. It is possible that providing an environment to a function that does not reference it directly is optimal, as that function can then provide it to its callees. But without any additional checks, a caller-provided environment could go entirely unused. For the best assurances on program efficiency and memory usage, we can conservatively say that it is a good idea to provide the environment across a call if that environment is guaranteed to be used: *i.e.*, all program traces result in the environment being accessed. We call a decision with this property *use-conservative*.

The other issue lies in how memory allocations can be shifted by the transformation. Converting closed-over environment into caller-provided environment shifts the location in which that variable is stored, and depending on the program it can result in worse performance. Suppose we have a call, *call*, with *lam* as the enclosing function ($lam = enclosingFn(call)$), and that *call* always jumps to the function lam'. Assume that lam' shares its environment with the environment of *call*, and thus we would like to arrange for *call* to provide that environment instead of requiring lam' to package it. There is situation in which this could be a bad idea. Even though the environment is available at *call*, that part of the environment may not have been free in *lam* to begin with. That is, the variables being passed to lam' may be *in scope* at *call*, but might not be *lexically referenced* by *lam*. When the transformation takes place, the compiler must modify *lam*'s closure to reference the environment, which increases the size of *lam*'s closure. Shifting allocations in this way may not save work, and if *lam* is constructed more times than lam' it may result in more allocation.

This second situation is challenging to handle optimally, and a compiler will need to utilize heuristics to effectively decide when the shifting of allocations is acceptable. Thus for our evaluation, we avoid transformations that could enlarge the size of closures entirely. Our requirement is that closures do not grow larger than their traditional, closure-converted versions. That is, we limit the set of variable available to provide to those that would be packaged due to lexical reference. We call this property *package-conservative*, since it limits the amount that we can package.

Definition 5.7. We define the *conservative* decision policy that satisfies both the *use-conservative* and *package-conservative* properties as follows.

$$\begin{aligned}
\mathcal{D}_{cons}(lam) \quad = \quad & \mathcal{P} \cap (freeVars(lam) \cup \mathcal{U}) \\
& \text{where } \mathcal{P} = \bigcap_{lam' \in web(lam)} (\mathcal{F}(lam')) \\
& \text{and } \mathcal{U} = \bigcap_{lam' \in web(lam)} \; \bigcup_{call \in callsOf(lam')} \; \bigcap_{\ell \in web(call)} (\mathcal{D}_{cons}(\ell) \cap \mathcal{F}(\ell))
\end{aligned}$$

5.2 Higher-Order Displays

While the use- and package-conservative properties limit avoid general pitfalls in caller-provided environments, an implementation should also take available machine resources into account. For example, the number of caller-provided variables might exceed the available argument registers, which means that they would have to either be passed on the stack. Thus, the per-definition memory-traffic cost of allocating the closure is replaced by a per-call cost of calling the function. Since most functions are called more often than they are defined, this trade-off is a net loss.

In practice, however, we can take advantage of the fact that the free variables of a function are bound in outer functions and can be saved in the outer stack frames. Furthermore, in the situation where there are a large number of free variables, it is likely that they are bound in a much smaller number of outer functions. Thus, rather than provide the environment as individual variables, we can provide pointers to the *frames* where those variables are bound.

```
fun top x = let
    fun h () = x
    fun g () = h () + x
  in g end
```

(a) First-order

```
fun top x = let
    fun h () = x
    fun f k = let
        fun g () = k () + x
      in g end
  in f h end
```

(b) Safe higher-order

```
fun top (x, y) = let
    fun h () = x
    fun f k = let
        fun g () = k () + x
      in g end
  in
    case y
     of NONE => h
      | SOME h => f h
  end
val h = top (100, NONE)
val g = top (200, SOME h)
```

(c) Unsafe higher-order

Listing 3. Closure sharing examples

The motivation behind using frames is that the time and space the program spends manipulating and managing environment is decreased. For example, the large set of free variables postulated above might all be bound two or three frames, meaning that there would be sufficient registers to implement the free variables as caller-provided environment.

Such an approach is reminiscent of the use of *displays* to implemented lexically scoped functions in Algol 60, Pascal, and similar languages [Dijkstra 1960; Nawrocki and Koster 1990; van den Hove 2017]. In these languages, functions can be passed down as arguments, but not returned as results (*i.e.*, known as *downward funargs*). In our setting, we can track function values in and out of other functions, and thus be able to use displays for higher-order functions in some cases. We intend to investigate this implementation technique in the 3CPS compiler in the future.

5.3 Flow-Directed Closure Sharing

Our analysis has applications beyond enabling caller-provided environments. Even when the compiler decides to package all free variables into a flat closure, the analysis can help identify redundancies within a closure. Consider the three examples in Listing 3. In the first program (Listing 3(a)), the function h closes over an integer x, and the function g closes over both h and x. Since h is syntactically bound to a function, it is easy to discover that after closure conversion, h refers to a closure containing x. A naïve closure converter would allocate a two-field record for g's closure environment — one for h and another for x. A more sophisticated converter may decide to only allocate one field storing h.[11] When g needs to access x, it can do so via h's closure. This optimization is performed in the Standard ML of New Jersey compiler [Shao and Appel 2000].

The closure-sharing optimization relies on knowing what closure flows to a variable. The first example uses syntactic flow information. With higher-order flow information, we might want to extend this technique. In the second example (Listing 3(b)), g no longer captures a syntactically bound function but rather a higher-order function parameter k. The flow analysis nevertheless discovers that there is only one possible value that flows to k, which is the closure of h. Since the closure of h contains x, it appears that g could reuse h's closure just as in Listing 3(a).

Although it is indeed correct to share the closure environment between h and g in Listing 3(b), it is not generally true that two variables with the same name necessarily refer to the same binding. Listing 3(c) gives such a counterexample. Here, the function top is called first to obtain h, and

[11]Or better yet, g can just reuse h's environment, saving an allocation.

```
let
  val a = 5
  fun f x = 2 * (x + a)
  val g = ···
in
  g (7 - a)
end
```

(a) Original code

```
let
  val a = 5
  fun f x = 2 * x
  val g = ···
in
  g ((7 - a) + a)
end
```

(b) Moving "_ + a" to the call

```
let
  val a = 5
  fun f x = 2 * x
  val g = ···
in
  g 7
end
```

(c) Simplification

Listing 4. Movement of code example. The variable g is eventually bound to the closure over f.

called a second time to obtain g, passing h as an argument. The two invocations of top result in two separate bindings for the variable x. In other words, the variable x in h's closure, referred to in g via k, is different from the variable x closed over by g. Because the two instances of x refer to different bindings, it is incorrect for g to access x from h's closure.

The syntactic approach does not need to consider bindings because if a function g closes over a syntactically bound function h within its scope, then g's binding environment must be an extension to h's. In other words, for the variables that are available at g's binding site, no rebinding can occur between g and h. Therefore, there is a one-to-one correspondence between the variable names and their bindings for those variables.

Lightweight closure conversion [Steckler and Wand 1997] addresses a similar issue by identifying a set of variables for which there is only one live binding that can exist at a time, named the *invariant set*. For such variables, the one-to-one relationship still exists, and variables with the same name cannot refer to different bindings. Our analysis solves this problem more directly by identifying which bindings can be shared between values. The result extracted from our analysis, $\mathcal{F}_V(k)$, calculates the bindings that remain unchanged between the binding environments of the closures that can flow to k and k's own binding environment. When g closes over k, since g's binding environment is an extension to k's, the set of variables that g can safely access via the closures that flow to k is $\mathcal{F}_V(g) \cap \mathcal{F}_V(k)$.

5.4 Super-β Optimization

One of the original applications of 0-CFA is "super-β" optimization [Shivers 1991]. If at a call $(f\ arg_1\ \ldots\ arg_n)$ the value of the f is always a closure over a single λ; that is, it always calls one function (with possibly different packaged environments), then that λ can be inlined at that call, effectively performing a β-reduction enabled by the analysis's semantic information. Because the inlined λ may package its environment, the closure object cannot simply be deleted; instead, the environment from the closure is extracted into let-bindings for the free variables of the λ. Our analysis can discover that this environment is shared between the callee and the caller, so that when inlining fewer extractions from the closure's environment are needed. In the best case, all free variables of the λ are shared, and the resulting code will not refer to the closure object.

One way to view super-β is that it moves the computation of the function body directly to the call. More generally, Quiring et al. [2025] introduce a flow-directed "code movement" transformation where fragments of computation are moved from function definition to function call, which is enabled by the semantic understanding provided by call webs. For example, consider Listing 4(a). It binds a variable a, captures it in the function f, performs some computation that eventually binds f to g, and then invokes g. The key is that the argument to g subtracts a, while f immediately adds it back. Webs enable moving code from function definition to call, but if that computation

Table 3. Allocation savings for the benchmarks

Program	**#Closure Allocs**	**#FV Allocs**	**Syntactic Saved**	**Conservative Saved**	**Conservative Δ Syn.**	**Maximal Saved**	**Maximal Δ Syn.**
nucleic	44,074	85,685	2.6%	51.4%	48.8%	51.4%	48.8%
boyer	1,417	2,121	98.8%	99.5%	0.8%	99.5%	0.8%
k-cfa	330	528	78.0%	82.6%	4.5%	91.1%	13.1%
ratio-regions	15,747	248,572	6.5%	75.5%	69.0%	75.5%	69.0%
mc-ray	1,654	7,633	2.1%	30.8%	28.8%	64.2%	62.1%
knuth-bendix	671,920	1,583,051	68.0%	68.3%	0.3%	69.5%	1.5%
raytracer	195	734	98.6%	98.6%	0.0%	100.0%	1.4%
s-n-f	105	467	53.7%	64.7%	10.9%	64.7%	10.9%
cps-convert	51	134	63.4%	63.4%	0.0%	67.2%	3.7%
parser-comb	1,066	2,130	17.9%	24.4%	6.5%	50.5%	32.6%
json-decode	183	314	24.5%	46.8%	22.3%	49.0%	24.5%
interpreter	135	143	30.1%	48.3%	18.2%	51.0%	21.0%
twenty-four	1,441	3,567	13.2%	20.9%	7.7%	35.3%	22.1%
life	5,584	6,295	88.6%	88.6%	0.0%	94.4%	5.8%
streams	22	37	35.1%	37.8%	2.7%	56.8%	21.6%
derivative	508	501	81.6%	100.0%	18.4%	100.0%	18.4%
tardis	483	1,566	1.0%	16.9%	15.8%	20.9%	19.9%

is not comprised of only constants or the parameters to the function, namely, if it contains other free variables, then the transformation may not preserve the program semantics. The key piece of information we need to generalize this technique to accomodate extra free variables, like in the example above, is our environment sharing analysis: it tells us that the environment of the call and the environment of the closure over `f` invoked at the call share `a`, meaning we can freely move references to `a` across this semantic boundary. The example Listing 4(b) moves the addition to the call, and Listing 4(c) eliminates the redundant computation.

6 Evaluation of Caller-Provided Environments

While our analysis can be applied more broadly, its primary — and perhaps the most impactful — use is to support the caller-provided-environment transformation introduced in Section 5. We focus our evaluation on this application to demonstrate how the analysis informs transformation decisions that could lead to practical improvements.

In our prototype compiler, we implemented the transformation with the decisions induced by both maximal and conservative polices. We ran an instrumented interpreter on the resulting program and collected **dynamic** allocation counts attributed to environment management.

Table 3 summarizes our results. To provide context, we present two baseline metrics: the number of allocated closures (**#Closure Allocs**) and the number of allocated free-variables stored in these closures (**#FV Allocs**).[12] Functions without free variables are only counted in the first of these.

The **Saved** columns show the percentage of free-variable allocations avoided using three different methods: syntactic analysis and our higher-order analysis with either the **Conservative** ($\mathcal{D}_{cons}$) or **Maximal** ($\mathcal{D}_{max}$) policies. A higher percentage represents a greater reduction in dynamic allocations.

[12]One should think of **#Closure Allocs** as the number of allocated objects and **#FV Allocs** as the number of allocated words.

The **Syntactic Saved** column shows the effect of a standard optimization for eliminating closure allocation that is enabled by a simple syntactic analysis [Keep et al. 2012; Shao and Appel 2000]. Specifically, if a function is never passed as an argument or stored in a data structure (called a *known function*), then its free variables are provided as arguments at all call sites. For example, 2.6% of free variable allocations are avoided by this optimization for the `nucleic` benchmark. For this analysis, there is no difference between the two policies.

In addition to the percentage of allocations saved for the two policies, we also present the improvement over the **Syntactic Saved** column. The Δ **Syn.** columns give the percentage of free-variable allocations saved above the **Syntactic Saved** results. For example, the $\mathcal{D}_{cons}$ policy saves 51.4% of the free variable allocations, which is an additional 48.8% over the syntactic savings.

Consistent with our findings in Section 4, for programs dominated by first-order functions, such as `raytracer` and `boyer`, the syntactic approach is effective. For higher-order programs, our approach — using either policy — uncovers many additional saving opportunities. While noticeable in certain programs, the difference between the maximal and conservative polices is typically small.

We note that although our transformation reduces total allocations, this reduction may not translate to overall performance improvements. For example, the evaluation does not reflect register pressure (see Section 5.2) or consider the interactions with other optimizations such as stack-allocated closures. For example, the conservative policy might produce better performance than the maximal policy because of reduced register spilling. We expect that more nuanced policies will be needed to improve overall performance in practice. Nevertheless, this evaluation shows that the traditional, syntactic method leaves many sharing opportunities unexploited, which the caller-provided-environment transformation uncovers when driven by our high-quality analysis.

7 Related Work

Lambda Lifting [Johnsson 1984] is a transformation that "lifts" nested functions from their scopes by converting free variables into parameters. Since it applies only to syntactically known functions, the environment-sharing problem is trivially solved, similar to our syntactic baseline analysis, and it does not consider higher-order functions. In Section 4, we demonstrate that our analysis uncovers many more sharing opportunities. Selective Lambda Lifting [Graf and Jones 2019] further examines the circumstances under which lambda lifting is beneficial. Specifically, it provides heuristics for evaluating the impact of using caller-provided environments on the callers' closures, which can help inform our decision policies (see Section 5).

Lightweight closure conversion (LWCC) [Siskind 1999; Steckler and Wand 1997] is perhaps the closest work to this paper, as it describes a program transformation that could be seen as a form of caller-provided environment. The condition it uses, however, is more limited than our environment sharing: a variable must have at most one binding at any time; it has to be "single." At a high level, the observation that LWCC makes is that if a function `f` closes over a variable `x` and `x` is never rebound before all uses of `f`, then `f` can be transformed to accept `x` as an additional argument. By doing so, `x` is no longer a free variable within `f`, eliminating the need for `f`'s closure to capture `x`. Because the transformation is source-to-source, the variable `x` must also be in scope at the call to `f`. Ensuring that a variable is never rebound is a sufficient, but not wholly necessary, condition to perform caller-provided environment — the caller can still pass `x` to `f`, provided that at the point of application, `x` retains the same binding it had at `f`'s point of definition. For example, consider the function in Listing 5. In this example, the function `number` takes parameters `idx` and `strings`. The `idx` parameter is used in the nested function named `prefix`, which is then called by the function `addPrefix`. The function `number` inspects its second argument and either returns or performs a recursive call followed by call to `addPrefix`. Our analysis is able to determine that the binding of `index` can be provided to `addPrefix`, and then to `prefix`, because the binding of `index`

```
fun number (idx : int, strings : string list) : string list = let
    fun prefix s = s ^ Int.toString idx
    fun addPrefix s = prefix s ^ s
  in
    case strings
     of [] => []
      | str :: rest => let
          val xs = number (idx + 1, rest)
        in addPrefix str :: xs end
  end
```

Listing 5. An example where our analysis outperforms LWCC

at `addPrefix`'s call site is the same as its definition site. In contrast, neither LWCC nor the 3CPS register-extent analysis is able to detect any caller-provided environment opportunities, because the recursive call to `number` rebinds `index` and thus `index` does not have register extent.

In addition, the analysis for LWCC operates under the assumption of a protocol-agreement constraint. This constraint dictates that all functions called from the same call site must use the same calling convention. In contrast, our analysis opts not to impose this constraint, instead deferring the decision to subsequent compiler passes as discussed in Section 5. Applying this constraint prematurely precludes some optimization strategies, including call-site splitting and defunctionalization, which could potentially enhance performance [Dimock et al. 2001].

The unchanged variable analysis presented by Bergstrom et al. [2014] calculates a similar property to LWCC's invariant set. Using an already computed approximate control-flow graph, the analysis checks if a variable is rebound at any point in the control path between the creation of the closure and its call. Even though this equivalence is not explicitly noted by the authors, we believe that the unchanged variables share the same property as LWCC, meaning the counterexample mentioned above is applicable.

Must-alias analyses [Germane and McCarthy 2021; Jagannathan et al. 1998] attempt to improve the analysis of the original Lightweight Closure Conversion paper and handle the prior code example. To do this, they utilize the notion of "singleness." A variable binding is *single* in a state if all reachable instances of that variable from the current environment have the same value. Thus, if an abstract address is known to be single and it is referenced in two live environments, the bindings in those environments must be the same. To obtain sufficient precision, they rely on abstract garbage collection [Might and Shivers 2006], which prunes the bindings that are not reachable from the current abstract state. This technique makes analyzing mutable cells more precise, at the cost of maintaining a per-state store — store-widening is not possible. This choice appears to have a significant cost in analysis time; in Germane and McCarthy [2021], the "boyer" example took 2 seconds to analyse for a monovariant version. While Jagannathan et al. include a partial complexity analysis, Germane and McCarthy do not examine the complexity of their implementation using heap fragments.

Our prior work on the 3CPS compiler [Quiring et al. 2022a,b] defines a notion of *register extent*. Variables that have register extent can be stored in the register set (or in global variables), so closures do not need to package them. The free references are resolved by hoisting the variables' definition site to the top level. Unlike that of this paper, there is the need to handle uninitialized values, as the register's scope lives longer than the scope of the bindings. The advantage, however, is that variables no longer need to be kept in-scope as they are passed from definition to use.

This approach to optimizing closure sizes complements our approach because they have different runtime behaviors and requirements; one technique may be able to optimize one allocation away, while the other eliminates a different one.

In the limited cases where a continuation's lifetime can be syntactically predicted, the ORBIT compiler [Kranz et al. 1986] employs stack allocation for its free variables, reducing the overall heap impact of closures (though not reducing their overall size). Appel and Shao [1992] optimize the case where a continuation literal is passed to a function in CPS: using the analogue of callee-saved registers, the continuation's variables can be passed as extra arguments to the receiving function, which can then relay them back to the continuation. Both of these techniques effectively eliminate the need for heap-allocated closures by ensuring that the continuation has no heap-allocated environment.

In situations where caller-provided environment is not possible and free variables need to be encapsulated in closures, several closure representations are proposed in literature, each with its own set of trade-offs [Appel and Jim 1988; Keep et al. 2012; Shao and Appel 2000]. Fundamentally, the problem of caller-provided environment is distinct from how environment is represented in a closure, whether the representation is flat, linked [Kranz 1988], or some other choice.

To our knowledge, the use of displays [Nawrocki and Koster 1990; Randell and Russell 1964] to represent environment structure has not been done in higher-order languages. The ORBIT Scheme compiler uses so-called "lazy displays" to refer to the elision of common access paths to variables inside deeply linked closures [Kranz 1988], which is different from our interpretation of displays as caller-provided environment.

Finally, it is worth discussing the relationship between our methods and defunctionalization [Brandon et al. 2023; Cejtin et al. 2000; Dimock et al. 2001; Pottier and Gauthier 2006; Tolmach and Oliva 1998]. The problem of defunctionalization is orthogonal to that of environment representation — while defunctionalization replaces a higher-order function value with a tag,[13] which can be used for dispatch at call sites, the environment of the function still needs to be captured, which is usually done by packaging the tag and the environment into a datatype constructor. A choice of the environment representation still needs to be made after defunctionalization. Defunctionalization does change the *web* information of the program (each web only contains a single function due to the dispatching), however, which affects the decisions about which variables to package and which to provide.

8 Conclusion

Effective environment management is one of the key aspects of implementing higher-order functional languages. One way to reduce the space and time overhead associated with environment management is by reusing existing environment structure. In this paper, we have presented an analysis that discovers environment-sharing information, outlined transformations that take advantage of the sharing information, and discussed factors that a compiler writer needs to consider in making transformation decisions. Our results are firmly grounded in a formal semantics (the Rocq proofs are submitted as a supplement) and have been evaluated on a number of benchmark programs, demonstrating the scalability and effectiveness of the analysis, as well as its potential to greatly reduce closure-related allocation.

Data Availability Statement

Our implementation in the 3CPS compiler containing our environment sharing analysis, experiments, and Rocq proofs of soundness are available as an artifact via Zenodo [3CPS 2025]. The

[13]Polyvariant defunctionalization can replace a function with multiple tags.

artifact contains instructions on how to reproduce our results and documentation relating the paper to the implementation. There is a separate `README.md` for the Rocq proofs.

Acknowledgments

This material is based upon work partially supported by the National Science Foundation under grant numbers 2212537 and 2212538. We thank the anonymous reviewers for their insightful feedback of this work.

References

3CPS. 2025. *Artifact for "Environment-Sharing Analysis and Caller-Provided Environments for Higher-Order Languages".* doi:10.5281/zenodo.15708994

Andrew W. Appel. 1992. *Compiling with Continuations.* Cambridge University Press, Cambridge, England, UK.

Andrew W. Appel and Trevor T.Y. Jim. 1988. *Optimizing Closure Environment Representations.* Technical Report CS-TR-168-88. Department of Computer Science, Princeton University. https://www.cs.princeton.edu/research/techreps/TR-168-88

Andrew W Appel and Zhong Shao. 1992. Callee-save registers in continuation-passing style. *Lisp and Symbolic Computation* 5 (Sept. 1992), 191–221. doi:10.1007/BF01807505

Lars Bergstrom, Matthew Fluet, Matthew Le, John Reppy, and Nora Sandler. 2014. Practical and effective higher-order optimizations. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 81–93. doi:10.1145/2628136.2628153

William Brandon, Benjamin Driscoll, Frank Dai, Wilson Berkow, and Mae Milano. 2023. Better Defunctionalization Through Lambda Set Specialization. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 977–1000. doi:10.1145/3591260

Luca Cardelli. 1983. *The Functional Abstract Machine.* Technical Report TR-107. AT&T Bell Laboratories. Available at http://lucacardelli.name/Papers/FAM.pdf.

Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. 2000. Flow-Directed Closure Conversion for Typed Languages. In *Proceedings of the 9th European Symposium on Programming Languages and Systems (ESOP '00) (Lecture Notes in Computer Science, Vol. 1782)*. Springer, Berlin, Heidelberg, 56–71. doi:10.1007/3-540-46425-5_4

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages (POPL '77)* (Los Angeles, CA, USA), Ravi Sethi (Ed.). Association for Computing Machinery, New York, NY, USA, 238–252. doi:10.1145/512950.512973

Olivier Danvy and Andrzej Filinski. 1992. Representing Control: A study of the CPS transformation. *Mathematical Structures in Computer Science* 2, 4 (Dec. 1992), 361–391. doi:10.1017/S0960129500001535

David Darais, Nicholas Labich, Phúc C. Nguyen, and David Van Horn. 2017. Abstracting Definitional Interpreters (Functional Pearl). *Proc. ACM Program. Lang.* 1, ICFP (Aug. 2017), 12:1–12:25. doi:10.1145/3110256

E. W. Dijkstra. 1960. Recursive Programming. *Numer. Math.* 2, 1 (Dec. 1960), 312–318. doi:10.1007/BF01386232

Allyn Dimock, Ian Westmacott, Robert Muller, Franklyn Turbak, and J. B. Wells. 2001. Functioning without Closure: Type-Safe Customized Function Representations for Standard ML. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming* (Florence, Italy) *(ICFP '01)*. Association for Computing Machinery, New York, NY, USA, 14–25. doi:10.1145/507635.507640

Christopher Earl, Ilya Sergey, Matthew Might, and David Van Horn. 2012. Introspective Pushdown Analysis of Higher-Order Programs. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*. Association for Computing Machinery, New York, NY, USA, 177–188. doi:10.1145/2364527.2364576

Elm 2021. *JSON in Elm.* Elm. Accessed: 2024-07-11.

Matthew Fluet. 2013. A Type- and Control-Flow Analysis for System F. In *Implementation and Application of Functional Languages*, Ralf Hinze (Ed.). Springer, Berlin, Heidelberg, 122–139. doi:10.1007/978-3-642-41582-1_8

Kimball Germane and Jay McCarthy. 2021. Newly-Single and Loving It: Improving Higher-Order Must-Alias Analysis with Heap Fragments. *Proceedings of the ACM on Programming Languages* 5, ICFP (Aug. 2021), 1–28. doi:10.1145/3473601

Thomas Gilray, Michael D Adams, and Matthew Might. 2018. Abstract allocation as a unified approach to polyvariance in control-flow analyses. *Journal of Functional Programming* 28 (2018), 46 pages. doi:10.1017/S0956796818000138

Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. 2016. Pushdown control-flow analysis for free. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 691–704. doi:10.1145/2837614.2837631

Sebastian Graf and Simon Peyton Jones. 2019. Selective Lambda Lifting. (2019). arXiv:1910.11717 [cs.PL] https://arxiv.org/abs/1910.11717

Pieter H. Hartel, Marc Feeley, Martin Alt, Lennart Augustsson, Peter Baumann, Marcel Beemster, Emmanuel Chailloux, Christine H. Flood, Wolfgang Grieskamp, John H. G. Van Groningen, and et al. 1996. Benchmarking implementations of functional languages with 'Pseudoknot', a float-intensive benchmark. *Journal of Functional Programming* 6, 4 (1996), 621–655. doi:10.1017/S0956796800001891

E. A. Hauck and B. A. Dent. 1968. Burroughs' B6500/B7500 Stack Mechanism. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference* (Atlantic City, NJ, USA) *(AFIPS '68 (Spring))*. Association for Computing Machinery, New York, NY, USA, 245–251. doi:10.1145/1468075.1468111

Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. 1998. 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 - POPL '98*. ACM Press, San Diego, California, United States, 329–341. doi:10.1145/268946.268973

Thomas Johnsson. 1984. Efficient compilation of lazy evaluation.. In *Proceedings of the 1984 SIGPLAN symposium on Compiler construction (SIGPLAN '84)*. Association for Computing Machinery, New York, NY, USA, 58–69. doi:10.1145/502874.502880

Andrew W. Keep, Alex Hearn, and R. Kent Dybvig. 2012. Optimizing Closures in O(0) Time. In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming* (Copenhagen, Denmark) *(Scheme '12)*. Association for Computing Machinery, New York, NY, USA, 30–35. doi:10.1145/2661103.2661106

David Kranz, Richard Kesley, Jonathan Rees, Paul Hudak, Jonathan Philbin, and Norman Adams. 1986. ORBIT: An Optimizing Compiler for Scheme. In *Proceedings of the 1986 Symposium on Compiler Construction* (Palo Alto, California). Association for Computing Machinery, New York, NY, USA, 219–233. doi:10.1145/13310.13333

David A. Kranz. 1988. *ORBIT: An Optimizing Compiler for Scheme.* Ph. D. Dissertation. Computer Science Department, Yale University, New Haven, Connecticut. Research Report 632.

P. J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* 6, 4 (01 1964), 308–320. doi:10.1093/comjnl/6.4.308

Matthew Might. 2010. Shape Analysis in the Absence of Pointers and Structure. In *Verification, Model Checking, and Abstract Interpretation*, Gilles Barthe and Manuel Hermenegildo (Eds.). Springer, Berlin, Heidelberg, 263–278. doi:10.1007/978-3-642-11319-2_20

Matthew Might and Panagiotis Manolios. 2008. A Posteriori Soundness for Non-deterministic Abstract Interpretations. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation* (Savannah, GA) *(VMCAI '09)*. Springer, Berlin, Heidelberg, 260–274. doi:10.1007/978-3-540-93900-9_22

Matthew Might and Olin Shivers. 2006. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming (ICFP'06)* (Portland, Oregon). Association for Computing Machinery, New York, NY, USA, 13–25. doi:10.1016/j.tcs.2006.12.031

Steven Muchnick. 1997. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, San Francisco, CA, USA.

J.R. Nawrocki and C.H.A. Koster. 1990. On display optimization for Algol-like languages. *Computer Languages* 15, 1 (1990), 27–39. doi:10.1016/0096-0551(90)90017-J

François Pottier and Nadji Gauthier. 2006. Polymorphic typed defunctionalization and concretization. *Higher-Order and Symbolic Computation* 19 (2006), 125–162. doi:10.1007/s10990-006-8611-7

Benjamin Quiring, John Reppy, and Olin Shivers. 2022a. 3CPS: The Design of an Environment-Focussed Intermediate Representation. In *Proceedings of the 33rd Symposium on Implementation and Application of Functional Languages (IFL '21)* (Nijmegen, Netherlands). Association for Computing Machinery, New York, NY, USA, 20–28. doi:10.1145/3544885.3544889

Benjamin Quiring, John Reppy, and Olin Shivers. 2022b. Analyzing Binding Extent in 3CPS. *Proc. ACM Program. Lang.* 6, ICFP, Article 114 (Aug. 2022), 29 pages. doi:10.1145/3547645

Benjamin Quiring and David Van Horn. 2024. Deriving with Derivatives: Optimizing Incremental Fixpoints for Higher-Order Flow Analysis. *Proc. ACM Program. Lang.* 8, ICFP, Article 261 (Aug. 2024), 28 pages. doi:10.1145/3674650

Benjamin Quiring, David Van Horn, John Reppy, and Olin Shivers. 2025. Webs and Flow-Directed Well-Typedness Preserving Program Transformations. *Proc. ACM Program. Lang.* 9, PLDI, Article 177 (June 2025), 25 pages. doi:10.1145/3729280

B. Randell and L. J. Russell. 1964. *Algol 60 Implementation: The Translation and Use of Algol 60 Programs on a Computer.* Academic Press, London and New York.

Manuel Serrano. 1995. Control Flow Analysis: A Functional Languages Compilation Paradigm. In *Proceedings of the 1995 ACM Symposium on Applied Computing (SAC '95)* (Nashville, Tennessee, USA) *(SAC '95)*. ACM, New York, NY, USA, 118–122. doi:10.1145/315891.315934

Zhong Shao and Andrew W. Appel. 2000. Efficient and Safe-for-Space Closure Conversion. *ACM Transactions on Programming Languages and Systems* 22, 1 (Jan. 2000), 129–161. doi:10.1145/345099.345125

Peter Shirley. 2020. *Ray Tracing in One Weekend.* Ray Tracing in One Weekend. https://raytracing.github.io

Olin Shivers. 1991. *Control-Flow Analysis of Higher-Order Languages, or Taming Lambda.* Ph. D. Dissertation. School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania. Technical Report CMU-CS-91-145.

Jeffrey M. Siskind. 1999. *Flow-Directed Lightweight Closure Conversion.* Technical Report 99-190R. NEC Research Institute, Princeton, New Jersey.

Paul A Steckler and Mitchell Wand. 1997. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19, 1 (1997), 48–86. doi:10.1145/239912.239915

Andrew Tolmach and Dino P. Oliva. 1998. From ML to Ada: Strongly-Typed Language Interoperability via Source Translation. *Journal of Functional Programming* 8, 4 (July 1998), 367–412. doi:10.1017/S0956796898003086

Gauthier van den Hove. 2017. Dissolving a half century old problem about the implementation of procedures. *Sci. Comput. Program.* 150, C (dec 2017), 75–86. doi:10.1016/j.scico.2017.07.007

David Van Horn and Matthew Might. 2010. Abstracting Abstract Machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '10)*. Association for Computing Machinery, New York, NY, USA, 51–62. doi:10.1145/1863543.1863553

Dimitrios Vardoulakis and Olin Shivers. 2011. CFA2: A context-free approach to control-flow analysis. *Logical Methods in Computer Science* 7, 2, Article 3 (May 2011), 39 pages. doi:10.2168/LMCS-7(2:3)2011 Special issue for ESOP 2010..

Guannan Wei, James Decker, and Tiark Rompf. 2018. Refunctionalization of Abstract Abstract Machines: Bridging the Gap between Abstract Abstract Machines and Abstract Definitional Interpreters (Functional Pearl). *Kraks/RefuncAAM: Release v1.0.0* 2, ICFP (July 2018), 105:1–105:28. doi:10.1145/3236800

Received 2025-02-27; accepted 2025-06-27