Flow-directed Space-efficient Closure Conversion

Bangyuan “Byron” Zhong
byronzhong@cs.uchicago.edu
University of Chicago

December 10, 2024

Abstract

In higher-order languages, a function captures its free variables in a closure, often implemented as a flat record. While this simple structure is effective, more sophisticated representations can significantly reduce overhead. To manage the multiplicity of possible closure representation decisions, this paper introduces a model that represents closure decisions and a transformation that applies these decisions to a program. This paper presents several optimization passes that rewrite the closure representations, each of which greedily minimizes the dynamic allocation required for closures. Higher-order control-flow analysis provides more information about the flows of function values. This information is used both to justify the soundness for some of the proposed rewriting passes and to provide heuristics for others. This paper presents an efficient implementation of 0-CFA with supports for ref-cells and separate compilation. This novel approach is implemented in the Standard ML of New Jersey compiler and evaluated against the existing implementation. The results show that this approach is scalable to large programs and effective in reducing dynamic allocations.

Contents

1 Introduction

Closure conversion is a standard technique for implementing nested functions in a higher-order language with static scoping. In such a language, a function may refer to a variable defined in its outer scope, and when the function is invoked, it must use the value of that variable at definition time to evaluate its body. A function definition therefore introduces not only the code but also a *binding environment* that captures the values of its free variables. This pair, consisting of a code pointer and a binding environment, is referred to as a *closure* and is typically used to represent a higher-order function (Landin, 1964).

All optimizing compilers exploit the following principle in various forms: if both the introduction site and all elimination sites for a value are known, the compiler can choose any representation it prefers, provided that the representations for all potential values from the same elimination site remain consistent. Otherwise, the compiler must fall back to using some uniform representation, such as tagged data and boxed values. The quality of the generated code largely depends on the compiler's ability to choose the most efficient representation for a given value. Closure conversion is no exception. The introduction site of a closure is the corresponding function's definition, and the elimination sites are the call sites. If all call sites of a function can be identified, the compiler can make a decision on the representation of its closure and transform the program such that the call sites expect the specialized representation.

As a closure is made up of a code pointer and an environment, the representations of the two components can be specialized independently. While many existing works examine the specialization of code pointers through *defunctionalization* (Bell et al., 1997; Cejtin et al., 2000; Dimock et al., 2001; Contractor and Fluet, 2020; Brandon et al., 2023), this paper focuses on the specialization of environments. In the semantics of most higher-order functional languages, the binding environment is defined simply as an abstract map from a variable to a value. Computer hardware, however, does not have such an abstract map, so compilers must choose a specific data structure to represent it during closure conversion. A common choice is to use flat records, where the binding of each free variable occupies a field. If a closure uses a flat record as its binding environment, it is called a flat closure (Cardelli, 1983). Flat-closure conversion is straightforward to implement, but the converted program often incurs high memory traffic.

To illustrate different environment representations, we use the following code snippet as an example.

```
fix f(a, b, c) =
  fix g1(x) =
    fix g2(y) =
      fix g3(z) =
        fix h() = (x + y + z) * (a + b + c) in
        h in
      g3 in
    g2 in
  g1
```

This code defines a function f, which returns another function g_1 when applied. Following the same pattern, g_1 returns g_2, which in turn returns g_3. Finally, g_3 returns the function h, which performs some computation using all the variables in scope. This pattern is common for curried functions. Each of these nested functions captures the variables from its surrounding scope, and each closure progressively adds more captured variables.

Using a flat record to represent each closure means that the values of a, b, and c have slots in the closures for g_1, g_2, g_3, and h. Not only does the flat closure representation have high space usage, but it also incurs high memory traffic. For example, to create the closure for g_2 in the body of g_1, all three bindings must be copied from g_1's closure to the newly allocated closure for g_2. This operation is repeated

for all the intermediary functions until the variables are used in the computation in h. We can express the results from the flat-closure conversion in the same language by explicitly constructing closure records and passing the closures as additional arguments. The converted code is listed below, where `let` x = (...) `in` e denotes a record allocation and $x.i$ denotes a 1-based field selection.

```
fix f'(clo_f, a, b, c) =
  fix g'_1(clo_1, x) =
    fix g'_2(clo_2, y) =
      fix g'_3(clo_3, z) =
        fix h'(clo) = (clo.2 + clo.3 + clo.4) * (clo.5 + clo.6 + clo.7) in
        let h = (h', clo_3.2, clo_3.3, clo_3.4, clo_3.5, clo_3.6, z) in
        h in
      let g_3 = (g'_3, clo_2.2, clo_2.3, clo_2.4, clo_2.5, y) in
      g_3 in
    let g_2 = (g'_2, clo_1.2, clo_1.3, clo_1.4, x) in
    g_2 in
  let g_1 = (g'_1, a, b, c) in
  g_1 in
let f = (f') in
f
```

In languages where higher-order functions are used infrequently, the simplicity of flat closures can be appealing, and the inefficiencies they introduce are often accepted as the natural cost of supporting higher-order functions. In languages, such as Standard ML, where higher-order functions are a fundamental feature and their performance is critical, this trade-off becomes less desirable. To this end, more sophisticated closure models have been developed.

An example of an improved closure model is the safely-linked-closure representation, developed by Shao and Appel (2000). Linked-closure representations exploit the fact that multiple closures may contain bindings for the same set of variables, such as a, b, and c in the above example. Instead of occupying multiple slots in each closure, the shared variables are grouped into a separate record, and all closures that contain these bindings can link to the record. As a result, the closures only need to provide one field for the shared bindings, reducing both the allocation size and memory traffic. In the converted code below, all closures share the same bindings for a, b, and c. Moving the bindings from one closure to another thus only requires one memory copy. One level of indirection is needed, however, for h to use the three variables.

```
fix f'(clo_f, a, b, c) =
  fix g'_1(clo_1, x) =
    fix g'_2(clo_2, y) =
      fix g'_3(clo_3, z) =
        fix h'(clo) = (clo.1.1 + clo.1.2 + clo.1.3) * (clo.2 + clo.3 + clo.4) in
        let h = (h', clo_3.1, clo_3.2, clo_3.3, z) in
        h in
      let g_3 = (g'_3, clo_2.1, clo_2.2, y) in
      g'_3 in
    let g_2 = (g'_2, clo_1.1, x) in
    g_2 in
  let shared = (a, b, c) in
  let g_1 = (g'_1, shared) in
  g_1 in
let f = (f') in
```

Table 1: Functions and their free variables

	FV	Flat	Linked	Safely Linked	Spread
f	$\emptyset$	(f')	(f')	(f')	f', _, _, _, _, _, _
g_1	$\{a, b, c\}$	(g_1', a, b, c)	(g_1', a, b, c)	(g_1', shared)	f', a, b, c, _, _, _
g_2	$\{a, b, c, x\}$	(g_2', a, b, c, x)	(g_2', g_1, x)	$(g_2', \mathit{shared}, x)$	f', a, b, c, x, _, _
g_3	$\{a, b, c, x, y\}$	(g_3', a, b, c, x, y)	(g_3', g_2, y)	$(g_3', \mathit{shared}, x, y)$	f', a, b, c, x, y, _
h	$\{a, b, c, x, y, z\}$	(h', a, b, c, x, y, z)	(h', g_3, z)	$(h', \mathit{shared}, x, y, z)$	f', a, b, c, x, y, z

f

There are two primary reasons why the contents of a closure are accessed. First, a field may be accessed because the function needs to use the free variable, and second, a field may be accessed because the function needs to construct another closure that refers to the same variable. In other words, in the former case, the field is a closure is used directly, and in the latter, the field is immediately moved into another closure without being used. We refer to the former as access *for use* and the latter as access *for bookkeeping*. In this example, safely-linked-closure representations reduce the bookkeeping but increase the price of use.

On the other hand, one may optimize for *use* by representing the binding environment as extra arguments that travel with the function variable. In other words, the closure can be *spread* across multiple variables. Doing so eliminates any memory traffic when the function accesses those variables, but if the function is captured in another closure, that closure requires additional fields. Importantly, when calling a function whose closure record is spread, the caller must apply the correct closure arguments; when passing such a function as an argument, the argument must be expanded by the number of closure arguments. To ensure that functions are always applied with the correct number of these closure arguments, without more precise control flow information, a compiler may choose a uniform arity for certain kinds of functions — callee-save registers for continuations is an instance of this approach (Appel and Shao, 1992). As an example, if the compiler decides that all closures are uniformly represented by six variables, the same program may be rewritten as follows.

```
fix f'(_, _, _, _, _, _, a, b, c) =
  fix g1'(a, b, c, _, _, _, x) =
    fix g2'(a, b, c, x, _, _, y) =
      fix g3'(a, b, c, x, y, _, z) =
        fix h'(a, b, c, x, y, z) =
          (a + b + c) * (x + y + z) in
        (h', a, b, c, x, y, z) in
      (g3', a, b, c, x, y, nil) in
    (g2', a, b, c, x, nil, nil) in
  (g1', a, b, c, nil, nil, nil)
```

In this example, we assume that the language supports multiple return values without the need to allocate a boxed record and that there is a constant, `nil`, serving as padding. Here, all functions are spread across seven variables (a code pointer plus six environment variables), and the calling convention for a function is to apply the code pointer with six environment variables, followed by its actual arguments.

The different closure representations are summarized in Table 1. For in-memory closures, the flat representation is most efficient for "use," and the linked representation is most efficient for "bookkeeping." While spreading a closure into multiple variables reduces both use and bookkeeping costs, doing so inflates the sizes of other closures that capture it and raises register pressure in the code that uses it.

Additionally, any concrete representation of an abstract binding environment must not increase asymptotic space complexity; in other words, the closure representation should be *safe for space* (Appel, 1992). A contrasting example of an *unsafe*-for-space closure representation is the linked closure representation (Landin, 1964), where a closure's environment always includes the values of its free variables local to the enclosing function and a pointer to the enclosing function's closure, shown in Table 1. As a result, a variable free in the enclosing function — yet unused by the child function — is still reachable from the child's closure, artificially extending its lifetime. Shao and Appel (2000) provide a concrete example showing how this pattern can asymptotically worsen the space consumption of a program, and Glasser (2013) observes the same effect in JavaScript's V8 engine, which uses linked closure representations.

Deciding on the most efficient closure representations involves choosing the optimal combination of closure sharing and closure spreading while maintaining soundness and space safety. For compiler writers who want to optimize closure representations, this complexity presents a challenge.

The key observation is that after closure conversion, closure representation becomes a problem of calling-convention specialization. After closure conversion, a function's environment is explicitly passed as an argument or a list of arguments. Those environment parameters have a special property: the body of the function can assume that the same values — the bindings of its free variables at the definition — are always passed in. This property is unusual for a function's parameters, as parameters typically exist to receive different values at different call sites. In a way, that an environment parameter should always be bound to a specific value can be viewed as part of the function's calling convention. Maintaining a function's environment is equivalent to ensuring that all potential uses of the function adhere to the same calling convention.

This paper makes the following contributions:

1. It formulates a framework for expressing different environment representation decisions and presents a flow-directed transformation that executes these decisions (Section 4).
2. It presents an efficient high-order control-flow analysis and several related analyses that leverage its results to support more informed decision-making (Section 5).
3. It develops a novel approach to closure representation decisions by starting with flat closure representations and incrementally rewriting them to greedily optimize for space efficiency (Section 6).
4. It implements and evaluates this framework in the Standard ML of New Jersey (SML/NJ) compiler, showing the effectiveness and scalability of this approach (Section 7).

2 Background

2.1 Standard ML of New Jersey

Although the method that this paper describes is applicable to any functional language, its implementation is developed and evaluated in the Standard ML of New Jersey (SML/NJ) system (Appel and MacQueen, 1991). SML/NJ is a Standard ML implementation that supports interactive programming through a Read-Evaluate-Print Loop (REPL) and incremental recompilation via a compilation manager (CM). A compilation unit, which can be as small as a line of code or as large as a source file, can contain free references to variables bound in previous compilation units and may introduce new bindings for a later one. SML/NJ represents a compilation unit as a *closed* λ-abstraction that takes a record of values representing its dependencies as arguments and returns a record storing the bindings that it introduces. Consequently, linking is implemented as an ordinary function application without concerns for OS-specific object-file linkage.

SML/NJ uses a continuation-passing-style (CPS) intermediate representation (IR) in its "middle-end" pipeline, bridging the semantic gap between the higher-order IR used by the FLINT front-end (Shao, 1997a)

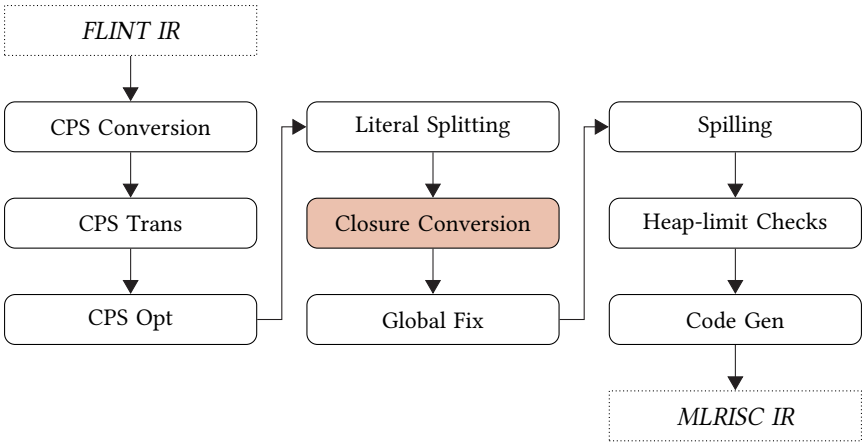

Figure 1: CPS pipeline.

and the low-level IR used by the MLRISC code generator (George et al., 1994). The phases in the CPS pipeline are outlined in Figure 1.

- After CPS conversion, the *CPS Trans* phase ensures that each function's arguments will fit in the available machine registers.
- The *CPS Opt* phase performs various common optimizations, including inlining, dead-variable elimination, and primitive-operator simplification.
- The *Literal Splitting* phase lifts string, floating-point literals, and data structures containing only literals out of the compilation unit and replaces them with a reference to the same value constructed by the runtime system.
- The *Closure Conversion* phase is the focus of this paper, which is a source-to-source transformation that explicitly allocates a closure for each function.
- With all functions now closed, the *Global Fix* phase lifts nested functions out of their scopes, transforming the compilation unit into a single mutually recursive function definition with no nested functions.
- The *Spilling* phase ensures the amount of live data at any given point does not exceed the runtime system's spill-area size.
- The *Heap-limit Checks* phase inserts garbage-collection checkpoints.
- Finally, the *Code Gen* phase lowers the first-order CPS program into MLRISC code.

The results of the program at different stages are represented using the same data type, with evolving invariants at each stage. Both the input and the output of the closure converter are CPS programs, but an additional invariant is enforced — after closure conversion, all functions are *closed* (*i.e.*, they have no free variables). This transformation is accomplished by explicitly allocating and passing a closure structure for each function, including continuations. The result of closure conversion is a first-order CPS IR, referred to as *continuation-passing closure-passing-style* IR (CPCPS) (Appel and Jim, 1989; Shao and Appel, 2000; Appel, 1992). At this stage, a "function" becomes a code address in memory, resembling more closely the notion of functions in lower-level languages like C.

SML/NJ uses a stackless runtime model called the CMachine (Appel and Jim, 1989; Appel, 1990, 1992), where calls and returns are not managed by a traditional stack. In a CPS IR, when calling a function, the caller passes a continuation, which is invoked with the function's return value. After closure conversion, a continuation closure stores the state to resume execution, such as the live variables across a function call, just like a stack frame. Instead of using a stack to push and pop, each continuation closure holds a reference to the next "frame" (that is, the continuation above it), effectively creating a linked structure that replicates the behavior of a stack. This design greatly simplifies the interaction between the runtime system and the garbage collector, as calculating the roots for reachable values no longer requires scanning a stack. Without a stack to manage calls and returns, however, this runtime model places additional pressure on the closure conversion algorithm.

2.2 Closure Conversion in SML/NJ

SML/NJ implements a sophisticated closure conversion algorithm, described in the paper by Shao and Appel (2000). In essence, the closure converter first tries to reduce the size of a closure by employing callee-save registers and lifting free variables for known functions. When free-variable allocations are necessary, the closure converter performs *closure sharing*.

Extended call graph. The heuristics used in the closure converter are supported by a *syntactic* approximation of the runtime call graph, named an *extended* call graph. A function f is said to *directly call* g if there is a path that ends with a call to the known function g. In the extended call graph, there is an edge between the functions f and g if f directly calls g or if f directly calls some other function with g as its return continuation. The closure converter estimates loop levels and variable lifetimes using the extended call graph.

Callee-save registers. As a consequence of the stackless model, all registers are initially caller-save. When a function returns, *i.e.*, when its return continuation is applied, the function does not restore any of the caller's registers, and it is the duty of the return continuation to capture all live variables across the function call in its closure. In other words, the caller allocates a record, in the form of a closure, holding all variables that are live across the function call, and upon return, these variables are reloaded from the record.

The calling conventions in most traditional compilers designate a set of registers to be callee-save (Chow, 1988); that is, the callee is responsible for preserving the values in these registers across the call. When the callee does not need those registers, the contents in the callee-save registers need not be moved during the function's execution. Appel and Shao observe that a similar approach can be applied in the context of CPS (Appel and Shao, 1992). Specifically, a fixed number of extra parameters can be added after the continuation parameter, and when the continuation is invoked, these extra parameters are passed back to it as part of the function call. These bindings are preserved across a function call, making them effectively *callee-save*. In practice, SML/NJ uniformly designates three callee-save registers for each continuation.

This technique is useful when a continuation's closure environment can fit entirely into the set of callee-save parameters — evaluating such a continuation requires no allocation. The technique is particularly effective when the callee is a leaf function, that is, a function that does not make any further function calls. In such cases, because it is the final point in a call chain, the callee does not need to save and restore the content of the registers before invoking its return continuation.

Known free-variable lifting. Another optimization implemented by SML/NJ is the lifting of free variables for first-order functions, as is commonly applied in other optimizing compilers (Adams et al., 1986;

$$
\begin{array}{rcl}
P \in \mathit{Prog} & ::= & \texttt{program}\ f(x_1, \ldots, x_n) = e \\
e \in \mathit{Exp} & ::= & \texttt{fix}\ f(x_1, \ldots, x_n) = e_1\ \texttt{in}\ e_2 \\
& | & \texttt{let}\ y = (a_1, \ldots, a_n)\ \texttt{in}\ e \\
& | & \texttt{let}\ y = x.i\ \texttt{in}\ e \\
& | & \texttt{if}\ a\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\
& | & f(a_1, \ldots, a_n) \\
a \in \mathit{Atom} & ::= & x \mid c \\
x, y, f, g, h, k \in \mathit{Var} & = & \text{unique identifiers} \\
\texttt{tt}, \texttt{ff}, \texttt{nil}, c \in \mathit{Const} & = & \text{constants} \\
i \in \mathit{Index} & = & \mathbb{N}
\end{array}
$$

Figure 2: The abstract syntax of the CPS IR.

Keep et al., 2012). If a function is never passed as an argument to another function or a data constructor, satisfying the $\mathcal{S}$ property in Serrano's taxonomy (Serrano, 1995), the function's environment can be passed as additional arguments at all of its call sites rather than packaged in a closure. This technique is only sound when the compiler can ascertain both of the following properties for a given function:

1. All call sites of the functions are known.
2. All call sites have access to the environment at the function's definition sites.

Since the current implementation of SML/NJ does not perform any higher-order flow analysis, this optimization is only applied to syntactically known functions, which trivially satisfy these properties.

Closure sharing. After applying the above optimizations, any remaining free variables are grouped by their lifetimes and allocated into separate, shareable records. These records are accessible to nested functions within the current function's body and to the functions defined later.

2.3 Notations

The paper uses $\langle x_1, \ldots, x_n \rangle$ to denote an ordered sequence, $[n]$ to refer to the set $\{1 \ldots n\}$, and $A \rightharpoonup B$ to represent a partial map from the set A to B.

3 CPS

This section presents the syntax and the semantics of a simplified higher-order CPS IR, based on the one used internally by SML/NJ. The analyses in the paper operate on this IR, and the closure-conversion is a source-to-source transformation, consistent with the structure of the SML/NJ compiler. This section also defines control-flow "facts" in this IR, which play a crucial role in the closure-conversion transformation described in Section 4.

3.1 Syntax

The abstract syntax of the IR is presented in Figure 2. A compilation unit (*Prog*) is a top-level function that, when invoked, calls its return continuation with a record of exported functions. The IR contains five types of expressions (*Exp*), listed in the order that they appear: recursive function definitions, record constructions, record selections, conditional branches, and tail calls. In this IR, all sub-expressions are

$$
\begin{array}{rcl}
l \in \mathit{Loc} & = & \text{an infinite set} \\
v \in \mathit{Val} & ::= & l \mid c \mid f(x_1, \ldots, x_n) = e \\
\rho \in \mathit{Env} & = & \mathit{Var} \rightharpoonup \mathit{Val} \\
o \in \mathit{Obj} & ::= & \texttt{Record}(v_1, \ldots, v_n) \mid \texttt{Clo}(f(x_1, \ldots, x_n) = e, \rho) \\
H \in \mathit{Heap} & := & \mathit{Loc} \rightharpoonup \mathit{Obj} \\
\varsigma \in \mathit{State} & = & \mathit{Exp} \times \mathit{Heap} \times \mathit{Env}
\end{array}
$$

Figure 3: Semantic domains of the CPS IR.

bound to unique names, and all calls are tail calls. The language also contains some unspecified set of constants to simulate the IR used by a real compiler. The only assumption that this paper makes is that among the constants, there is a true value `tt`, a false value `ff`, and a non-pointer constant `nil` that can be used as a placeholder.

For simplicity, the IR's syntax makes no distinction between user-defined functions and continuations introduced during CPS conversion. Depending on the context, however, a function is either a function or a continuation and the usual restrictions apply — *i.e.*, continuations do not have continuation arguments.

We also define the predicate $e \in P$, meaning that the expression e appears in the program P as follows.

$$
\frac{}{e \in [\![\texttt{program}\ f(x_1, \ldots, x_n) = e]\!]} \qquad \frac{[\![\texttt{fix}\ f(x_1, \ldots, x_n) = e_1\ \texttt{in}\ e_2]\!] \in P}{e_1 \in P}
$$

$$
\frac{[\![\texttt{fix}\ f(x_1, \ldots, x_n) = e_1\ \texttt{in}\ e_2]\!] \in P}{e_2 \in P} \qquad \frac{[\![\texttt{let}\ y = x.i\ \texttt{in}\ e]\!] \in P}{e \in P} \qquad \frac{[\![\texttt{let}\ y = (x_1, \ldots, x_n)\ \texttt{in}\ e]\!] \in P}{e \in P}
$$

$$
\frac{[\![\texttt{if}\ a\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!] \in P}{e_1 \in P} \qquad \frac{[\![\texttt{if}\ a\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!] \in P}{e_2 \in P}
$$

3.2 Operational Semantics

The CPS IR is untyped and uses standard call-by-value semantics. The semantic domains are presented in Figure 3. Notably, the program does not use a stack to record calls and returns — a program state consists of the current expression *Exp*, a heap *Heap*, and a binding environment *Env*. There are three kinds of values: heap locations (l), constants (c), and code ($f(x_1, \ldots, x_n) = e$). The heap stores two kinds of objects: n-field records ($\texttt{Record}(v_1, \ldots, v_n)$) and closures, which consist of a code pointer and an environment.

The operational semantics is presented in Figure 4 as a transition system. A compilation unit, which takes a return continuation for the exported functions and the record of the functions that it depends on ($[\![\texttt{program}\ f(k, x) = e]\!]$), evaluates to a closure with empty environment $\texttt{Clo}(f(k, x) = e, \emptyset)$. The semantics use the following auxiliary function to evaluate an atomic value.

$$
\mathcal{A}(\rho, x) = \rho(x) \qquad\qquad \mathcal{A}(\rho, c) = c
$$

A variable evaluates to its binding in ρ while a constant is simply evaluated to itself.

When evaluating a function definition, we allocate a fresh heap address and bind it to a closure containing the function's code and the current environment, restricted to the free variables in the body minus its arguments. At an application site, we retrieve the closure from the heap and transition to its body, evaluating it in the closure's environment augmented with parameter bindings. Record introduction and elimination follow a similar pattern: we allocate a fresh address and bind it to a record object on the heap

$$\frac{l \text{ fresh} \qquad H' = H[l \mapsto \texttt{Clo}(f(x_1, \ldots, x_n) = e', \rho|_{fv(e') \setminus \{x_1, \ldots, x_n\}})]}{\langle [\![\texttt{fix } f(x_1, \ldots, x_n) = e' \texttt{ in } e]\!], H, \rho \rangle \rightsquigarrow \langle e, H', \rho[f \mapsto l] \rangle}$$

$$\frac{H(l) = \texttt{Clo}(f(x_1, \ldots, x_n) = e', \rho') \qquad v_i = \mathcal{A}(\rho, a_i), \forall i \in [n] \qquad \rho(g) = l \qquad \rho'' = \rho'[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n, f \mapsto l]}{\langle [\![g(a_1, \ldots, a_n)]\!], H, \rho \rangle \rightsquigarrow \langle e', H, \rho'' \rangle}$$

$$\frac{v_i = \mathcal{A}(\rho, a_i), \forall i \in [n] \qquad l \text{ fresh} \qquad H' = H[l \mapsto \texttt{Record}(v_1, \ldots, v_n)] \qquad \rho' = \rho[y \mapsto l]}{\langle [\![\texttt{let } y = (a_1, \ldots, a_n) \texttt{ in } e']\!], H, \rho \rangle \rightsquigarrow \langle e', H', \rho' \rangle}$$

$$\frac{l = \rho(x) \qquad H(l) = \texttt{Record}(v_1, \ldots, v_i, \ldots, v_n)}{\langle [\![\texttt{let } y = x.i \texttt{ in } e']\!], H, \rho \rangle \rightsquigarrow \langle e', H, \rho[y \mapsto v_i] \rangle} \qquad \frac{\mathcal{A}(\rho, a) = \texttt{tt}}{\langle [\![\texttt{if } a \texttt{ then } e_1 \texttt{ else } e_2]\!], H, \rho \rangle \rightsquigarrow \langle e_1, H, \rho \rangle}$$

$$\frac{\mathcal{A}(\rho, a) = \texttt{ff}}{\langle [\![\texttt{if } a \texttt{ then } e_1 \texttt{ else } e_2]\!], H, \rho \rangle \rightsquigarrow \langle e_2, H, \rho \rangle}$$

Figure 4: Small-step Operational Semantics for CPS IR

during the introduction, and we access a field by looking it up in the object. For an if-expression, if the condition evaluates to `tt`, we transition to the first branch; other we transition to the second.

3.3 Control Flows and Webs

The "control flow" of a program refers to the order in which expressions are evaluated at runtime. Simple expressions, like record creation and selection, have trivial orderings. For an if-expression, the control flow is syntactically limited to the two branches. For a function application, the subsequent expression depends on the function variable's dynamic bindings. The dynamic nature introduces complexity because variable bindings both influence and result from function applications. A control-flow analysis (Shivers, 1991; Midtgaard, 2012) offers a static approximation of the dynamic bindings of first-class functions and, consequently, the control flow of the program.

This paper uses a monovariant, flow-graph-based formulation of control-flow analysis (Heintze and McAllester, 1997; Midtgaard and Van Horn, 2009; Adsit and Fluet, 2014). The analysis produces four kinds of "flow facts" about a program P, written as logical relations and summarized in Figure 5. The first relation, $P \vdash \hat{v} \longrightarrow x$, indicates that a value $\hat{v}$ flows to x. The second relation, $P \vdash x \rightarrowtail y$, signifies that if a value flows to x, it must also flow to y; this relation encapsulates transitivity. The other two relations address escaping and unknown values, which are necessary for separate compilation. The unary relation $P \vdash\uparrow \hat{v}$ shows that $\hat{v}$ is an escaping value, meaning that its use sites are unknown and require a standard calling convention. Finally, given a variable x, $P \vdash\downarrow x$ signifies that any value that flows to x escapes.

An efficient algorithm that gathers these facts is presented in Section 5. For now, we assume that a *sound* result set is available via some means. As an example, a syntactic approach to gathering the flow information, such as the one implemented in SML/NJ, can be expressed as follows. In this analysis, the set

$\hat{v} \in \widehat{Val}$	(Abstract values)
$P \vdash \hat{v} \rightarrow x$	(An abstract value flows to a variable)
$P \vdash x \rightarrowtail y$	(A variable flows to another variable)
$P \vdash\uparrow \hat{v}$	(An abstract value escapes)
$P \vdash\downarrow x$	(A variable is passed to an escaping function)

Figure 5: Monovariant Control Flow Results.

of abstract values is the set of code pointers: $\widehat{Val} ::= f(x_1, \ldots, x_n) = e$.

$$\frac{[\![\texttt{fix}\ f(x_1, \ldots, x_n) = e_1\ \texttt{in}\ e_2]\!] \in P}{P \vdash f(x_1, \ldots, x_n) = e_1 \rightarrow f}$$

$$\frac{[\![\texttt{fix}\ f(x_1, \ldots, x_n) = e_1\ \texttt{in}\ e_2]\!] \in P \qquad [\![\texttt{let}\ y = (a_1, \ldots, f, \ldots, a_n)\ \texttt{in}\ e]\!] \in P}{P \vdash\uparrow f(x_1, \ldots, x_n) = e_1}$$

$$\frac{[\![\texttt{fix}\ f(x_1, \ldots, x_n) = e_1\ \texttt{in}\ e_2]\!] \in P \qquad [\![g(a_1, \ldots, f, \ldots, a_n)]\!] \in P}{P \vdash\uparrow f(x_1, \ldots, x_n) = e_1} \qquad \frac{[\![\texttt{fix}\ x(\ldots) = e_1\ \texttt{in}\ e_2]\!] \notin P}{P \vdash\downarrow x}$$

Every λ-abstraction only flows to the syntactically bound variable. When any function variable is used in a higher-order way, the function is marked as escaping. Finally, any variable that is not the name of a function is also marked as escaping.

A data-flow web, $w \in Web = \mathcal{P}(Fun) \times \mathcal{P}(Var)$, is defined as the transitive closure of *du*- and *ud*-chains (Muchnick, 1997). Specifically, given a result from control-flow analysis, $U(f)$ identifies all variables that the function f can flow to, and $D(x)$ indicates all functions that the variable x can be bound to.

$$U(f) = \{x : Var \mid P \vdash f \rightarrowtail x\} \qquad D(x) = \{f : Fun \mid P \vdash f \rightarrowtail x\}$$

Starting from a singleton set of variables and an empty set of function definitions, the function $web(x)$ constructs a web by iteratively combining definitions that share a common use. Since both *Fun* and *Var* are finite, this iteration is guaranteed to terminate at a fixed point. A web can be defined recursively as follows.

$$\begin{aligned} web(x) &= \langle defs, uses \rangle \\ \text{where} \quad defs &= \textstyle\bigcup_{x \in uses} D(x) \\ uses &= \{x\} \cup \textstyle\bigcup_{f \in defs} U(f) \end{aligned}$$

By calculating the web for each function, we identify all webs in the program. These webs form a partition over the entire sets of *Fun* and *Var*. This means that each function and variable in the program is uniquely assigned to exactly one web.

Because any call site calling a variable in *uses* might invoke any function in *defs*, all functions and call sites must agree on a calling convention. This calling convention, in the context of closure conversion, includes the environment's arity (how many "roots" each function has to access its environment) and the placement of the code pointer (where to find the code pointer from the roots). Within a web, any calling convention can be chosen as long as there is agreement, and functions outside the web are not constrained by this choice. In other words, we can choose a calling convention independently for each web.

There is, however, an exception to this independence. Because of separate compilation, some webs may contain an escaping function whose call site can appear outside the current compilation unit. Similarly, a

call site may call an unknown function from other units. We call such a web an *escaping* web. Formally, an escaping web is defined as follows:

$$escapingWeb(\langle defs, uses\rangle) \iff (\exists f \in defs, P \vdash\uparrow f(x_1, \ldots, x_n) = e) \vee (\exists v \in uses, P \vdash\downarrow v)\,.$$

Since an escaping web needs to interface with the outside world, all functions need to follow the language's standard calling convention.

4 Closure Conversion

This section introduces a way to specify closure decisions, which is derived from the concrete machine state presented in Section 3. It then describes a closure-conversion transformation that rewrites the CPS program according to given closure decisions. The section concludes with an example of such conversion.

4.1 Representation of Closures

In CPS, the runtime system does not need a traditional stack, requiring a closure to capture the entire environment that a function depends on except for its parameters. This property is evident in the operational semantics: at an application site, when the control transfers to the callee's body, the environment is constructed by simply extending the closure environment with parameter bindings. Since CPS functions do not "return" in the traditional sense, the combined environment effectively captures all bindings necessary for the remainder of the program's execution. Phrased differently, in CPCPS, where closures are explicitly passed as parameters, everything a function requires is reachable from the parameters.

This property provides a high degree of flexibility in structuring the parameters. As long as the bindings reachable from the parameters stay the same, the parameters can be rearranged — spread, nested, or otherwise reorganized — without affecting correctness. This flexibility has been studied and exploited in "unboxed" representations for parameter passing (Ziarek et al., 2008; Bergstrom and Reppy, 2009).

While a closure record is syntactically the same as a user-defined record, its usage is much more regular. Closure records are abstract outside their owning functions, akin to values with existential type (Tarditi et al., 1996). Only the owning function can construct its closure, and all selections from the closure record occur within the function. The closure's value is not only known but also has a statically unique introduction site. The static uniqueness ensures that within a function, there is only one possible layout for the environment. If a field in this layout is spread, the indices of all other fields can be adjusted accordingly. In contrast, user-defined records lack this guarantee. Even in a typed language, after datatypes are desugared into records, a record may have different layouts depending on the constructors. This variability makes it challenging to update selection indices consistently, especially if a spread field appears in some, but not all, possible layouts.

The representation of a closure, therefore, depends only on the individual function. Externally, only the calling convention of the closure needs to be known. The domains of closure representation decisions are defined in Figure 6. To refer to the functions defined in a program, we define *Fun* as the subset of variables statically bound to function values. For each function, we can choose one of the following two representations: `Spread` and `Boxed`. As Section 1 alludes to, the spread representation is a generalization of using callee-save registers for continuations (Appel and Shao, 1992). Unlike the callee-save-register approach, which only applies to continuations and imposes a uniform number of parameters to represent a spread closure, the `Spread` representation applies to all non-escaping functions and continuations, allowing each web to customize its arity. On the other hand, the `Boxed` representation encodes the uniform heap-allocated closure, where the first element is the code pointer.

A function's closure can be allocated multiple times during execution, and each allocation receives a fresh address, but all allocations of a closure for a given function necessarily share the same layout.

$$
\begin{array}{rcl}
\mathit{Fun} & = & \{f : \mathit{Var} \mid \texttt{fix}\ f(x_1, \ldots, x_n) = e\ \texttt{in}\ e' \in P\} \\
\mathit{env} \in \mathit{EnvID} & = & \text{a set of identifiers} \\
\Gamma \in \mathit{Clos} & = & \mathit{Fun} \rightarrow \mathit{Repr} \\
L \in \mathit{Layout} & = & \mathit{EnvID} \rightarrow \mathit{Slot}^* \\
\Phi \in \mathit{Allocator} & = & \mathit{Fun} \rightarrow \mathit{EnvID}^* \\
\mathit{Repr} & ::= & \texttt{Spread}(f_{code}, s_1, \ldots, s_n) \mid \texttt{Boxed}(\mathit{env}) \\
s \in \mathit{Slot} & ::= & \texttt{Var}(v) \mid \texttt{Code}(f) \mid \texttt{EnvID}(\mathit{env}) \mid \texttt{Nil} \mid \texttt{Expand}(v, i) \\
\mathit{Decision} & ::= & \langle \Gamma, L, \Phi \rangle
\end{array}
$$

Figure 6: Decision Representation Domains.

Therefore, we use the set *EnvID* to represent all concrete locations allocated for a given closure record, and we use *Layout* to indicate the layout of the record. Additionally, *EnvID* captures closure sharing. When two closures contain the same *EnvID*, at runtime the two closures point to the same heap object. The allocator map Φ identifies the function responsible for allocating an environment record, tracking closure creation.

The contents of a record are represented using *Slot*. A slot s abstracts the possible concrete values that a record field can hold, and it has the following five variants. $\texttt{Code}(f)$ indicates that the field can only be a function pointer to f. $\texttt{EnvID}(\mathit{env})$ indicates that the field can only point to another known record, typically another closure. `Nil` indicates that the field can only be the special constant `nil`. If the field in a record may not be unique during execution, we represent it as $\texttt{Var}(v)$, meaning the field contains any values that flow to v. Finally, $\texttt{Expand}(v, i)$ holds the place for the expansion of v during transformation.

As an example, an entry $\Gamma(f) = \texttt{Boxed}(\mathit{env}_f)$ indicates that we use a simple flat closure record env_f to represent f's environment. If $\Gamma(f) = \texttt{Spread}(f_{code}, s_1, \ldots, s_n)$, it means that we decide to use a flat representation of f's environment, expanding f to $n + 1$ variables. All higher-order uses of f will be expanded to the sequence $\langle f_{code}, s_1, \ldots, s_n \rangle$, and when f is applied, the slots $s_1, \ldots, s_n$ are expected to be passed as arguments.

One can view Γ and L as monovariant views of the binding environment and the heap, respectively, that a function expects on entry; *i.e.*, they encode the calling convention of a function. At the most basic level, a function expects that all of its formal arguments be bound to the values of the actual arguments in the environment; after flat-closure conversion, for example, a function additionally expects that the first argument be bound to a heap location that contains the closure tuple. In this model, a function can expect specific bindings to be installed at specific arguments.

4.2 Transformation

A triplet $\langle \Gamma, L, \Phi \rangle$ thus specifies closure-representation decisions for a program P. This section introduces a transformation that executes the decisions. To summarize, Γ indicates how the environment of a closure is represented; they are "roots" to a graph containing all bindings needed in the evaluation of the function's body; L indicates what is stored at each of the entry points; Φ indicates what environments a function is responsible for allocating.

The decisions are *global* in that they do not change when the transformation steps into a new scope. For clarity, the rest of this section omits the decision maps in the definitions of various relations, and it is implicit that all transformations are defined in the context of a particular decision triplet $\langle \Gamma, L, \Phi \rangle$.

To execute the decisions, the transformation uses three environments for bookkeeping: α, ϕ, and σ. The domains for these environments are listed in Figure 7. The access map α tells for each non-local variable what is the access path for that variable given the new closure decision map Γ. The protocol

$$
\begin{array}{rcl}
w \in \mathit{Web} & = & \mathcal{P}(\mathit{Fun}) \times \mathcal{P}(\mathit{Var}) \\
\alpha \in \mathit{Access} & = & \mathit{Var} \rightharpoonup \mathit{Path} \\
\phi \in \mathit{Protocols} & = & \mathit{Var} \to \mathit{Protocol} \\
\sigma \in \mathit{InScope} & = & \mathcal{P}(\mathit{Var}) \\
\mathit{Protocol} & ::= & \texttt{Special}(f_{code}, s_1, \ldots, s_n) \mid \texttt{Uniform} \\
\mathit{Header} \ni \mathcal{E} & ::= & \texttt{let } y = (a_1, \ldots, a_n) \texttt{ in } \mathcal{E} \\
& \mid & \texttt{let } y = x.i \texttt{ in } \mathcal{E} \\
& \mid & [\cdot] \\
\mathit{path} \in \mathit{Path} & := & x \mid \mathit{path}.i
\end{array}
$$

Figure 7: Transformation domains.

map ϕ indicates what a variable should be replaced with according to the protocol decided for a web to which the variable belongs. Finally, a set σ indicates if a non-local variable is already in scope so as not to generate duplicate bindings. A header $\mathcal{E}$ is an expression with exactly one hole; we write $\mathcal{E}[e]$ to replace the hole with an expression e. Since the header is used to generate access headers for a variable or to allocate environment tuples, only relevant expressions are included.

Although the source program in the higher-order IR has the invariant that every variable has a unique name, we relax this invariant for the resulting code to simplify the transformation. The relaxed invariant is that there is no rebinding in the lexical scope of a variable; in other words, no variable binding *shadows* another. This relaxation is particularly useful because when multiple functions close over the same variable, we would like to reuse the same name after closure conversion. Since each function has its own access path for the non-local variable, the variable must be renamed differently in each function to maintain the stronger invariant. The relaxed invariant eliminates the need for tracking these new names and updating the decision maps to reflect the new names. A simple renaming pass can restore the stronger invariant after the transformation.

The auxiliary relations used in the transformation are listed in Figure 8 with short descriptions of their intended use. As discussed in Section 3.3, functions belonging to the same web need to agree on a common convention. Since the transformation frequently refers to the protocol of the web that a variable belongs to, we define the following function *webproto* that returns the common calling convention.

$$
\frac{\mathit{escapingWeb}(\mathit{web}(x))}{\mathit{webproto}(\Gamma, x) = \texttt{Uniform}} \qquad \frac{\langle \emptyset, \{x\} \rangle = \mathit{web}(x)}{\mathit{webproto}(\Gamma, x) = \texttt{Uniform}}
$$

$$
\frac{\langle \mathit{defs}, \mathit{uses} \rangle = \mathit{web}(x) \qquad \mathit{defs} \neq \emptyset \qquad \Gamma(f) = \texttt{Boxed}(\mathit{env}_f), \forall f \in \mathit{defs}}{\mathit{webproto}(\Gamma, x) = \texttt{Uniform}}
$$

$$
\frac{\langle \mathit{defs}, \mathit{uses} \rangle = \mathit{web}(x) \qquad \mathit{defs} \neq \emptyset \qquad \Gamma(f) = \texttt{Spread}(f_{code}, f_1, \ldots, f_n), \forall f \in \mathit{defs} \qquad x_1, \ldots, x_n \text{ fresh}}{\mathit{webproto}(\Gamma, x) = \texttt{Special}(x_{code}, x_1, \ldots, x_n)}
$$

This function is well-defined only if the functions in the web indeed have a common environment size.

At a high level, the transformation expands each higher-order variable to a sequence of variables that represent the function's environment and inserts a sequence of formal parameters (of the same length) at the function's definition to receive the environment. At the same time, the transformation needs to fix the access path of each non-local variable that a function captures. Since this process transforms the program from one closure representation to another, it also takes the previous representation into account.

$H, path \vdash_\phi s \xrightarrow{\text{path}} \alpha$	(α is the access map reachable from the slot.)
$H \vdash_\phi \overrightarrow{s} \xrightarrow{\text{access}} \alpha$	(α is the access map reachable from a list of slots.)
$\phi \vdash xs \xrightarrow{\text{expand}} ys$	(ys is the expanded variables of xs according to a protocol.)
$\alpha, \sigma \vdash as \rhd \mathcal{E}, \sigma'$	($\mathcal{E}$ is an access header for a list of atomic values.)
$\alpha, \sigma \vdash envs \xrightarrow{\text{allocate}} \mathcal{E}, \sigma$	($\mathcal{E}$ is the allocation header for a list of environment IDs, $envs$.)
$\phi \vdash s \xrightarrow{\text{realize}} a, \sigma$	(An atomic value a is the realization of a slot in the source language.)
$x \vdash path \xrightarrow{\text{select}} \mathcal{E}$	($\mathcal{E}$ is the access header for x using the path $path$.)
$\alpha, \phi, \sigma \vdash e \rightsquigarrow e'$	(e' is the closure converted expression of e.)

Figure 8: Summary of Auxiliary Relations.

To transform a list of formal parameters, we expand each variable according to the protocol in context.

$$\frac{}{\phi \vdash [\,] \xrightarrow{\text{expand}} [\,]} \qquad \frac{\phi(x) = \texttt{Uniform} \qquad \phi \vdash xs \xrightarrow{\text{expand}} ys}{\phi \vdash x :: xs \xrightarrow{\text{expand}} x :: ys}$$

$$\frac{\phi(x) = \texttt{Special}(x_{code}, x_1, \ldots, x_n) \qquad \phi \vdash xs \xrightarrow{\text{expand}} ys}{\phi \vdash x :: xs \xrightarrow{\text{expand}} x_{code} :: x_1 :: \cdots :: x_n :: ys}$$

The relation $L, path \vdash_\phi s \xrightarrow{\text{path}} \alpha$ specifies the access map α containing all variables reachable from a given slot s. This access map is used to generate access paths for non-local variables.

$$\frac{}{L, path \vdash_\phi \texttt{Nil} \xrightarrow{\text{path}} \{\}} \qquad \frac{}{L, path \vdash_\phi \texttt{Code}(f) \xrightarrow{\text{path}} \{\}} \qquad \frac{}{L, path \vdash_\phi \texttt{Var}(v) \xrightarrow{\text{path}} \{v \mapsto path\}}$$

$$\frac{L(env) = \langle s_1, \ldots, s_n \rangle \qquad L, path.i \vdash_\phi s_i \xrightarrow{\text{path}} \alpha_i \qquad i = 1 \ldots n}{L, path \vdash_\phi \texttt{EnvID}(env) \xrightarrow{\text{path}} \{env \mapsto path\} \uplus \alpha_1 \uplus \ldots \uplus \alpha_n}$$

$$\frac{\phi(v) = \texttt{Special}(v_{code}, w_1, \ldots, w_i, \ldots, w_n)}{L, path \vdash_\phi \texttt{Expand}(v, i) \xrightarrow{\text{path}} \{w_i \mapsto path\}}$$

Because of closure sharing, a variable can sometimes can be reached via multiple paths. In this case, the shortest path is preferred as defined by the operator $\uplus$.

$$\begin{array}{rlll}
\alpha_1 \uplus \alpha_2 & = & \{v \mapsto \alpha_1(v) \mid v \in \mathrm{dom}(\alpha_1) \wedge v \notin \mathrm{dom}(\alpha_2)\} & \\
& \cup & \{v \mapsto \alpha_2(v) \mid v \notin \mathrm{dom}(\alpha_1) \wedge v \in \mathrm{dom}(\alpha_2)\} & \\
& \cup & \{v \mapsto \mathit{shorterOf}(\alpha_1(v), \alpha_2(v)) \mid v \in \mathrm{dom}(\alpha_1) \wedge v \in \mathrm{dom}(\alpha_2)\} & \\
\text{where} & \mathit{shorterOf}(path_1, path_2) & = & path_1 \qquad \text{if } \mathit{length}(path_1) \leqslant \mathit{length}(path_2) \\
& \mathit{shorterOf}(path_1, path_2) & = & path_2 \qquad \text{otherwise} \\
& \mathit{length}(v) & = & 0 \\
& \mathit{length}(path.i) & = & \mathit{length}(path) + 1
\end{array}$$

For convenience, we also define a similar rule that specifies the access paths of all reachable variables from a sequence of slots. At the same time, a fresh name z is generated to refer to each environment in the

list. This rule is used at a function's definition site, and the fresh names are additional formal arguments receiving the environment.

$$\frac{}{L \vdash [\,] \xrightarrow{\text{access}} [\,], \{\}} \qquad \frac{z \text{ fresh} \qquad L, z \vdash s \xrightarrow{\text{path}} \alpha \qquad L \vdash \overrightarrow{s} \xrightarrow{\text{access}} zs, \alpha'}{L \vdash s :: \overrightarrow{s} \xrightarrow{\text{access}} z :: zs, \alpha \uplus \alpha'}$$

Given an access map, α, which maps a non-local variable to a path, *path*, the following rules specify an access header for a list of variables. No access header needs to be generated for an argument if any of the following is true:

1. the argument is a constant;
2. the argument is a local variable; *or*
3. the argument's access header has been generated.

Otherwise, the rule defines a selection expression and records the fact that the access header has been generated. The left-hand side of the selection expression uses the same name as the variable to simplify bookkeeping and the environment σ ensures that the non-local variable has a unique local binding site — in other words, a variable's access header would not be generated more than once.

$$\frac{}{\alpha, \sigma \vdash [\,] \rhd [\cdot], \sigma} \qquad \frac{\alpha, \sigma \vdash as \rhd \mathcal{E}, \sigma'}{\alpha, \sigma \vdash c :: as \rhd \mathcal{E}, \sigma'} \qquad \frac{x \notin \text{dom}(\alpha) \qquad \alpha, \sigma \vdash as \rhd \mathcal{E}, \sigma'}{\alpha, \sigma \vdash x :: as \rhd \mathcal{E}, \sigma'}$$

$$\frac{x \in \sigma \qquad \alpha, \sigma \vdash as \rhd \mathcal{E}, \sigma'}{\alpha, \sigma \vdash x :: as \rhd \mathcal{E}, \sigma'} \qquad \frac{\alpha(x) = path \qquad x \notin \sigma \qquad x \vdash path \xrightarrow{\text{select}} \mathcal{E}' \qquad \alpha, \sigma \cup \{x\} \vdash as \rhd \mathcal{E}, \sigma'}{\alpha \vdash x :: as \rhd \mathcal{E}' \left[\mathcal{E}[\cdot]\right], \sigma' \cup \{x\}}$$

The following relation $\xrightarrow{\text{select}}$ specifies the selection header for an access path. Specifically, the relation $x \vdash path \xrightarrow{\text{select}} \mathcal{E}$ signifies that $\mathcal{E}$ is a selection header that puts the content following the path *path* into x. Since the IR does not have a syntactic form for renaming, if the selection path is the trivial path, we require the base of the path has the same name as the destination.

$$\frac{}{x \vdash x \xrightarrow{\text{select}} [\cdot]} \qquad \frac{}{x \vdash y.i \xrightarrow{\text{select}} \texttt{let}\ x = y.i\ \texttt{in}\ [\cdot]} \qquad \frac{y \text{ fresh} \qquad y \vdash path.i \xrightarrow{\text{select}} \mathcal{E}}{x \vdash path.i.j \xrightarrow{\text{select}} \mathcal{E}[\texttt{let}\ x = y.j\ \texttt{in}\ [\cdot]]}$$

We also define the following relation $\xrightarrow{\text{realize}}$ that reflects an abstract slot into an atomic value in the program. We assume that there is a mapping from each environment ID *env* to a fresh variable, rewritten as x_{env}.

$$\frac{\phi(x) = \texttt{Uniform}}{\phi \vdash \texttt{Var}(x) \xrightarrow{\text{realize}} x} \qquad \frac{\phi(x) = \texttt{Special}(f_{code}, x_1, \ldots, x_n)}{\phi \vdash \texttt{Var}(x) \xrightarrow{\text{realize}} f_{code}} \qquad \frac{}{\phi \vdash \texttt{EnvID}(env) \xrightarrow{\text{realize}} x_{env}}$$

$$\frac{}{\phi \vdash \texttt{Nil} \xrightarrow{\text{realize}} \texttt{nil}} \qquad \frac{}{\phi \vdash \texttt{Code}(f) \xrightarrow{\text{realize}} f_{code}}$$

$$\frac{\phi(v) = \texttt{Special}(f_{code}, s_1, \ldots, s_i, \ldots, s_n) \qquad \phi \vdash s_i \xrightarrow{\text{realize}} v}{\phi \vdash \texttt{Expand}(x, i) \xrightarrow{\text{realize}} v}$$

A function is sometimes responsible for allocating an environment structure as a part of the closure decision Φ. The following rule specifies the execution of this decision. For a list of environments to allocate, the rule defines a header constructing a record containing the content indicated by L. If the construction of some record fields requires non-local variables, their access headers are also included.

$$\frac{}{\alpha, \sigma \vdash [\,] \xrightarrow{\text{allocate}} [\cdot], \sigma}$$

$$\frac{\begin{array}{c} L(env) = \langle s_1, \ldots, s_n \rangle \\ \phi \vdash s_i \xrightarrow{\text{realize}} a_i, \forall i \in 1 \ldots n \qquad \alpha, \sigma \vdash [a_1, \ldots, a_n] \rhd \mathcal{E}, \sigma'' \qquad \alpha, \sigma' \vdash envs \xrightarrow{\text{allocate}} \mathcal{E}', \sigma'' \end{array}}{\alpha, \sigma \vdash env :: envs \xrightarrow{\text{allocate}} \mathcal{E}[\texttt{let } x_{env} = (a_1, \ldots, a_n) \texttt{ in } \mathcal{E}'], \sigma''}$$

Finally, the following auxiliary function is defined to return the name of an atomic value. If the value is already a variable, we simply return it; if not, we return a fresh name.

$$\frac{}{getVar(v) = v} \qquad \frac{w \text{ fresh}}{getVar(c) = w}$$

The transformation rules are defined by the relation $\rightsquigarrow$ in Figure 9. The relation operates in the context of an access map α, a protocol environment ϕ, a set of non-local variables that are in scope σ, and specifies the transformation of an expression.

4.3 An Example

We use the following (contrived) program to demonstrate the transformation. On the left, we present the CPS-converted version of the program, and on the right, we show the original source code for comparison.

```
program p(k') =
  fix f(k, n) =
    if n = 0
      then k(0)
      else fix k1(res) = k( res + 1) in
           f(k1, n − 1) in
  fix k0(res) = k'(res) in
  f(k0, 100)
```

```
program p() =
  fix f(n) =
    if n = 0
      then 0
      else f(n − 1) + 1 in
  f(100)
```

This program defines a recursive function f that, given a non-negative integer, performs some computation and ultimately returns the same value. It applies f to the integer 100 and returns the result.

There are three distinct webs in this program.

$$w_1 = \langle \{k_0(res) = k'(res), k_1(res) = k(res + 1)\}, \{k, k_0, k_1\} \rangle$$
$$w_2 = \langle \{f(k, n) = \ldots\}, \{f\} \rangle$$
$$w_3 = \langle \{unknown\}, \{k'\} \rangle$$

The recursive function f has no free variables. The continuation k_0, used in the "entry" call to f, has the top-level continuation k' as its free variable. Meanwhile, the continuation k_1, used in the recursive call to f, has a free variable k, which belongs to the same web as k_1. A potential closure representation decision

$$\frac{\begin{array}{c}\Gamma(f) = \texttt{Boxed}(env) \qquad \phi' = \phi[x_i \mapsto p_i \mid webproto(\Gamma, x_i) = p_i, x_i \in \vec{x}][f \mapsto \texttt{Boxed}(env)] \\ L \vdash_\phi [f_{env}] \xrightarrow{\text{access}} \alpha' \qquad \phi' \vdash \vec{x} \xrightarrow{\text{expand}} \vec{x'} \\ \alpha, \sigma \vdash \Phi(f) \xrightarrow{\text{allocate}} \mathcal{E}, \sigma' \qquad \alpha', \phi', \emptyset \vdash e_1 \rightsquigarrow e_1' \qquad \alpha, \phi[f \mapsto \texttt{Boxed}(env)], \sigma' \vdash e_2 \rightsquigarrow e_2'\end{array}}{\alpha, \phi, \sigma \vdash \texttt{fix}\ f(\vec{x}) = e_1\ \texttt{in}\ e_2 \rightsquigarrow \texttt{fix}\ f_{code}(f_{env} \cdot \vec{x'}) = e_1'\ \texttt{in}\ \mathcal{E}[e_2']}$$

$$\frac{\begin{array}{c}\Gamma(f) = \texttt{Spread}(f_{code}, s_1, \ldots, s_n) \\ \phi' = \phi[x_i \mapsto p_i \mid webproto(\Gamma, x_i) = p_i, x_i \in \vec{x}][f \mapsto \texttt{Special}(f_{code}, s_1, \ldots, s_n)] \\ \phi \vdash s_i \xrightarrow{\text{realize}} a_i, \forall i \in 1 \ldots n \\ L \vdash_\phi [s_0, \ldots, s_n] \xrightarrow{\text{access}} \alpha' \qquad \phi' \vdash \vec{x} \xrightarrow{\text{expand}} \vec{x'} \qquad z_i = getVar(a_i), \forall i \in 1 \ldots n \\ \alpha, \sigma \vdash \Phi(f) \xrightarrow{\text{allocate}} \mathcal{E}, \sigma' \qquad \alpha', \phi', \emptyset \vdash e_1 \rightsquigarrow e_1' \qquad \alpha, \phi[f \mapsto \texttt{Special}(f_{code}, s_1, \ldots, s_n)], \sigma' \vdash e_2 \rightsquigarrow e_2'\end{array}}{\alpha, \phi, \sigma \vdash \texttt{fix}\ f(\vec{x}) = e_1\ \texttt{in}\ e_2 \rightsquigarrow \texttt{fix}\ f_{code}(z_1, \ldots, z_n \cdot \vec{x'}) = e_1'\ \texttt{in}\ \mathcal{E}[e_2']}$$

$$\frac{\phi(f) = \texttt{Uniform} \qquad \alpha, \sigma \vdash f :: \vec{a} \rhd \mathcal{E}, \rho'}{\alpha, \phi, \sigma \vdash f(\vec{a}) \rightsquigarrow \mathcal{E}[\texttt{let}\ f_{code} = f.1\ \texttt{in}\ f_{code}(f, \vec{a'})]}$$

$$\frac{\begin{array}{c}\phi(f) = \texttt{Special}(f_{code}, s_1, \ldots, s_n) \\ \phi \vdash s_i \xrightarrow{\text{realize}} s_i', \forall i \in 1 \ldots n \qquad \alpha, \sigma \vdash f_{code} :: s_i' :: \cdots :: s_n' :: \vec{a} \rhd \mathcal{E}, \rho'\end{array}}{\alpha, \phi, \sigma \vdash f(\vec{a}) \rightsquigarrow \mathcal{E}[f_{code}(s_i' :: \cdots :: s_n' :: \vec{a})]}$$

$$\frac{\alpha, \sigma \vdash [a_1, \ldots, a_n] \rhd \mathcal{E}, \sigma' \qquad \alpha, \phi, \sigma' \vdash e \rightsquigarrow e'}{\alpha, \phi, \sigma \vdash \texttt{let}\ y = (a_1, \ldots, a_n)\ \texttt{in}\ e \rightsquigarrow \mathcal{E}[\texttt{let}\ y = (a_1, \ldots, a_n)\ \texttt{in}\ e']}$$

$$\frac{x = baseOf(path) \qquad \alpha, \sigma \vdash [x] \rhd \mathcal{E}, \sigma' \qquad \alpha, \phi[y \mapsto \texttt{Uniform}], \sigma' \vdash e \rightsquigarrow e'}{\alpha, \phi, \sigma \vdash \texttt{let}\ y = path\ \texttt{in}\ e \rightsquigarrow \mathcal{E}[\texttt{let}\ y = path\ \texttt{in}\ e']}$$

$$\frac{\alpha, \sigma \vdash [a] \rhd \mathcal{E}, \sigma' \qquad \alpha, \phi, \sigma' \vdash e_1 \rightsquigarrow e_1' \qquad \alpha, \phi, \sigma' \vdash e_2 \rightsquigarrow e_2'}{\alpha, \phi, \sigma \vdash \texttt{if}\ a\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightsquigarrow \mathcal{E}[\texttt{if}\ a\ \texttt{then}\ e_1'\ \texttt{else}\ e_2']}$$

Figure 9: Transformation.

is shown below.

$$\begin{aligned}
\Gamma = \{ \quad & f \mapsto \texttt{Boxed}(f_{env}), \\
& k_0 \mapsto \texttt{Spread}(k_{0\,code}, \texttt{Var}(k')), \\
& k_1 \mapsto \texttt{Spread}(k_{1\,code}, \texttt{EnvID}(k_{1\,env})) \quad \} \\
L = \{ \quad & f_{env} \mapsto \langle f_{code} \rangle, \\
& k_{1\,env} \mapsto \langle \texttt{Var}(k), \texttt{Expand}(k, 1) \rangle \quad \} \\
\Phi = \{ \quad & f \mapsto \langle f_{env} \rangle, \\
& k_0 \mapsto \langle \rangle, \\
& k_1 \mapsto \langle k_{1\,env} \rangle \quad \}
\end{aligned}$$

This decision is valid because the functions in the same web consistently use either the `Boxed` representation or the `Spread` representation with the same arity. Additionally, for any given function, all free variables are reachable from its representation.

To illustrate the mechanics of the transformation, the table below presents the contexts at each function definition. We show the states of the access map, α, and the protocol map, ϕ, both within the function body and immediately after the definition. Since $k1$ is defined in f, its entry is indented. Additionally, because the protocol map used by k_1 extends f's protocol map, the common parts are omitted for clarity. The in-scope map, σ, is not included in the table, as it remains empty at all points except at the call site in the body of k_1. The in-scope map is only populated with non-local variables selected from k_1's closure.

	α	ϕ
f (body)	$\{\}$	$\{k' \mapsto \texttt{Uniform}, f \mapsto \texttt{Uniform}, k \mapsto \texttt{Special}(k_{code}, x_0)\}$
k_1 (body)	$\{k_{code} \mapsto k_{1\,env}.1, x_0 \mapsto k_{1\,env}.2\}$	$\{\cdots, k_1 \mapsto \texttt{Special}(k_{1\,code}, \texttt{EnvID}(k_{1\,env}))\}$
k_1 (after)	$\{\}$	$\{\cdots, k_1 \mapsto \texttt{Special}(k_{1\,code}, \texttt{EnvID}(k_{1\,env}))\}$
f (after)	$\{\}$	$\{k' \mapsto \texttt{Uniform}, f \mapsto \texttt{Uniform}\}$
k_0 (body)	$\{k' \mapsto k'\}$	$\{k' \mapsto \texttt{Uniform}, f \mapsto \texttt{Uniform}, k_0 \mapsto \texttt{Special}(k_{0\,code}, \texttt{Var}(k'))\}$
k_0 (after)	$\{\}$	$\{k' \mapsto \texttt{Uniform}, f \mapsto \texttt{Uniform}, k_0 \mapsto \texttt{Special}(k_{0\,code}, \texttt{Var}(k'))\}$

The result of the closure conversion is shown in Figure 10.

5 Analyses

This section presents the analyses that Section 6 requires for making closure decisions. First, a simple syntactic analysis is performed to gather basic program information, such as variable use counts, free variables for each function, and the nesting depth of expressions. Since such an analysis is standard, we do not describe it in detail. Next, a control-flow analysis is performed to collect the "flow facts" and provide the basis for flow-web calculation as outlined in Section 3.3. Using the control-flow information, an approximated call graph is constructed, enabling the application of several traditional algorithms. Additionally, a greedy, bottom-up sharing analysis determines which free-variable bindings can be shared across closures.

5.1 Flow-based Control-flow Analysis

This section presents an efficient implementation of a control-flow analysis that drives the subsequent closure decisions. This analysis is developed based on a flow-based specification of control-flow analysis, conducive to a propagator-based implementation (Adsit and Fluet, 2014; Heintze and McAllester, 1997; Midtgaard and Van Horn, 2009), rather than an abstract-machine-based formulation more commonly seen in the literature (Van Horn and Might, 2010), for which a propagator-based implementation is not immediately obvious. Like most of the flow-based specifications, this analysis merges all bindings of the same

```
program p(k′) =
  fix f_code(f_env, k_code, x_0, n) =
    if n = 0
      then k_code(x_0, 0)
      else fix k_1code(k_1env, res) =
             let k_code = k_1env.1 in
             let x_0 = k_1env.2 in
             k_code(x_0, res + 1) in
           let k_1env = (k_code, x_0) in
           let f_code = f_env in
           f_code(f_env, k_1code, k_1env, n − 1) in
  let f_env = (f_code) in
  fix k_0(k′, res) =
    let k′_code = k′.1 in
    k′_code(k′, res) in
  let f_code = f_env.1 in
  f_code(f_env, k_0, k′, 100)
```

Figure 10: Closure conversion example.

$$\hat{v} \in \widehat{Val} \quad ::= \quad f(x_1, \ldots, x_n) = e \mid \widehat{\texttt{Record}}(i, x) \mid \widehat{\texttt{RecordAny}}(i) \mid \widehat{\texttt{Any}}$$

Figure 11: Abstract values and domains.

variable and all calling contexts of the same function; *i.e.* it is monovariant. To make it modular and scalable, the analysis tracks escaping functions in a novel way and introduces a new abstract representation for tuples to speed up convergence by an order of magnitude.

The abstract domain used by the analysis is shown in Figure 11. A "closure," represented simply as $f(x_1, \ldots, x_n) = e$, is the primary form of an abstract value. Since the analysis is monovariant, the abstract closure does not include an environment; only the code pointer is used to represent the closure. Another form of an abstract value is $\widehat{\texttt{Record}}(i, x)$, which indicates that x flows into the i-th field of a record. Similarly, $\widehat{\texttt{RecordAny}}(i)$ indicates that an unknown value flows into the i-th field of the record value. These two record value forms may seem peculiar, and their usage is explained below. The final form of an abstract value is $\widehat{\texttt{Any}}$, which represents an unknown value. This value is used when no specific information about the value is available.

The first set of rules in Figure 12 concerns the control transfers and the propagation of values. The reflexivity (Refl) rule states that when a function is introduced in the program, its closure flows to its name. The transitivity rule (Trans) ensures that if two variables are connected by the relation $\rightarrowtail$, any value flowing to the first variable also flows to the second. The call rule (Call) establishes connections between variables. At each application site, if an abstract closure flows to the function variable being applied, the formal parameters are connected with their corresponding actual arguments. Since this analysis does not track the flows of constants, if an actual argument is a constant, it is ignored. The remaining rules in this set handle escaping functions and unknown values. For an escaping function, the source (Src) rule declares that an abstract value $\widehat{\texttt{Any}}$ flows to all of its parameters, signifying that the escaping function is applied with an arbitrary value. Complementing this rule, the source rule asserts if a value flows to a variable marked as "sink," the value escapes. If a call site invokes an unknown value, all variables used as

$$\textsc{Refl}\ \frac{[\![\texttt{fix}\ f(x_1,\ldots,x_n) = e\ \texttt{in}\ e']\!] \in P}{P \vdash f(x_1,\ldots,x_n) = e \to f} \qquad \textsc{Trans}\ \frac{P \vdash \hat{v} \to x \qquad P \vdash x \rightarrowtail y}{P \vdash \hat{v} \to y}$$

$$\textsc{Call}\ \frac{P \vdash g(x_1,\ldots,x_n) = e \to f \qquad [\![f(a_1,\ldots,y_i,\ldots,a_n)]\!] \in P}{P \vdash y_i \rightarrowtail x_i} \qquad \textsc{Src}\ \frac{P \vdash\uparrow f(x_1,\ldots,x_i,\ldots,x_n) = e}{P \vdash \widehat{\texttt{Any}} \to x_i}$$

$$\textsc{Sink}\ \frac{P \vdash \hat{v} \to x \qquad P \vdash\downarrow x}{P \vdash\uparrow \hat{v}} \qquad \textsc{XCall}\ \frac{P \vdash \widehat{\texttt{Any}} \to f \qquad [\![f(a_1,\ldots,x_i,\ldots,a_n)]\!] \in P}{P \vdash\downarrow x_i}$$

Figure 12: Flow-based specification of CFA

actual arguments at the call site are marked as sinks.

The second set of rules in Figure 13 specifically processes record introduction and elimination. These specialized rules significantly reduce the number of generated "facts." Ordinarily, an abstract record value may be represented a tuple of abstract addresses, *e.g.* $\widehat{\texttt{SmplRecord}}(x_1,\ldots,x_n)$. A rule that directly mimics the semantic definitions would look as follows.

$$\textsc{SimpleRecord}\ \frac{[\![\texttt{let}\ y = (x_1,\ldots,x_n)\ \texttt{in}\ e]\!] \in P}{P \vdash \widehat{\texttt{SmplRecord}}(x_1,\ldots,x_n) \to y} \qquad \textsc{SimpleSelect}\ \frac{P \vdash \widehat{\texttt{SmplRecord}}(x_1,\ldots,x_i,\ldots,x_n) \to y \qquad [\![\texttt{let}\ y = x.i\ \texttt{in}\ e]\!] \in P}{P \vdash x_i \rightarrowtail y}$$

It seems that this representation results in a more compact set of facts. For each $\widehat{\texttt{SmplRecord}}(x_1,\ldots,x_n) \to y$ in the alternative analysis, the proposed rules instead generate n separate facts: $\widehat{\texttt{Record}}(i,x_i) \to y$ for $i = 1\ldots n$. In practice, however, functions typically only flow to a few fields in a record. Therefore, the analysis opts for a sparse representation of a closure; we only conclude such a record flow if the input is a function closure (RecordF) or another record (RecordR), and all other values are ignored. Furthermore, if a record contains an unknown value ($\widehat{\texttt{Any}}$) and that record flows to a field in another record, the analysis behaves as if an unknown value flowed directly to the field. Doing so greatly reduces the number of facts but also makes the analysis less precise. This imprecision, however, hardly affects the downstream decisions because we do not use a specialized calling convention for functions that flow into a data structure.

5.2 Web Analysis

One use of the CFA results is to discover call webs that must share the same calling convention (see Section 4). Since a web calculates the maximal union of *du*- and *ud*-chains, the web is obtained by leveraging a union-find data structure (Tarjan, 1983).

The high-level algorithm is listed in Algorithm 1. The union-find data structure provides two primary operations, *union* and *find*. The implementation details of union-find are omitted, and only their behaviors, specialized to web analysis, are defined. The expression $find(x)$ finds the web to which x belongs; in other words, $find(x)$ finds a unique web w such that $w = web(x)$ as defined in Section 3.3. For convenience, $find(x)$ is also written as w_x. We note that using this definition, w_x and w_y may be different aliases of the same web even when $x \neq y$. Initially, for each function f in the program $w_f = \langle\{f(x_1,\ldots,x_n) = e\},\{f\}\rangle$. For all other variables, $w_v = \langle\emptyset,\{v\}\rangle$. The expression $union(x,y)$ combines the two webs to which x

$$\textsc{RecordF}\quad \frac{P \vdash f(x_1, \ldots, x_m) = e \rightarrow x_i \qquad [\![\texttt{let } y = (a_1, \ldots, x_i, \ldots, a_n) \texttt{ in } e]\!] \in P}{P \vdash \widehat{\mathsf{Record}}(i, x_i) \rightarrow y}$$

$$\textsc{RecordR}\quad \frac{P \vdash \widehat{\mathsf{Record}}(j, z) \rightarrow x_i \qquad [\![\texttt{let } y = (a_1, \ldots, x_i, \ldots, a_n) \texttt{ in } e]\!] \in P}{P \vdash \widehat{\mathsf{Record}}(i, x_i) \rightarrow y}$$

$$\textsc{RecordA}\quad \frac{P \vdash \widehat{\mathsf{Any}} \rightarrow x_i \qquad [\![\texttt{let } y = (a_1, \ldots, x_i, \ldots, a_n) \texttt{ in } e]\!] \in P}{P \vdash \widehat{\mathsf{RecordAny}}(i) \rightarrow y}$$

$$\textsc{RecordRA}\quad \frac{P \vdash \widehat{\mathsf{RecordAny}}(j) \rightarrow x_i \qquad [\![\texttt{let } y = (a_1, \ldots, x_i, \ldots, a_n) \texttt{ in } e]\!] \in P}{P \vdash \widehat{\mathsf{RecordAny}}(i) \rightarrow y}$$

$$\textsc{SelectR}\quad \frac{P \vdash \widehat{\mathsf{Record}}(i, z) \rightarrow y \qquad [\![\texttt{let } x = y.i \texttt{ in } e]\!] \in P}{P \vdash z \rightarrowtail x} \qquad \textsc{SelectA}\quad \frac{P \vdash \widehat{\mathsf{Any}} \rightarrow y \qquad [\![\texttt{let } x = y.i \texttt{ in } e]\!] \in P}{P \vdash \widehat{\mathsf{Any}} \rightarrow x}$$

$$\textsc{SelectRA}\quad \frac{P \vdash \widehat{\mathsf{RecordAny}}(i) \rightarrow y \qquad [\![\texttt{let } x = y.i \texttt{ in } e]\!] \in P}{P \vdash \widehat{\mathsf{Any}} \rightarrow x}$$

Figure 13: Flow-based specification of CFA (cont.)

$$\frac{P \vdash f(x_1, \ldots, x_n) = e \rightarrow g}{P \vdash \mathit{union}(f, g)} \qquad \frac{P \vdash x \rightarrowtail y}{P \vdash \mathit{union}(x, y)} \qquad \frac{P \vdash \downarrow x}{P \vdash \mathit{unionStd}(x)}$$

$$\frac{[\![g(a_1, \ldots, x_i, \ldots, a_n)]\!] \in P \qquad \nexists \hat{v}, P \vdash \hat{v} \rightarrow g}{P \vdash \mathit{unionStd}(x_i)} \qquad \frac{[\![\texttt{let } y = (a_1, \ldots, x_i, \ldots, a_n) \texttt{ in } e]\!] \in P}{P \vdash \mathit{unionStd}(x_i)}$$

$$\frac{[\![\texttt{fix } f(x_1, \ldots, x_n) = e_1 \texttt{ in } e_2]\!] \in P}{P \vdash \mathit{find}(f) = \langle \{f(x_1, \ldots, x_n) = e\}, \{f\} \rangle} \qquad \frac{[\![\texttt{fix } f(x_1, \ldots, x_n) = e_1 \texttt{ in } e_2]\!] \notin P}{P \vdash \mathit{find}(f) = \langle \{\emptyset\}, \{f\} \rangle}$$

$$\frac{P \vdash \mathit{find}(x) = \langle \mathit{defs}, \mathit{uses} \rangle \qquad P \vdash \mathit{find}(y) = \langle \mathit{defs}', \mathit{uses}' \rangle \qquad P \vdash \mathit{union}(x, y)}{P \vdash \mathit{find}(x) = \langle \mathit{defs} \cup \mathit{defs}', \mathit{uses} \cup \mathit{uses}' \rangle}$$

$$\frac{P \vdash \mathit{find}(x) = \langle \mathit{defs}, \mathit{uses} \rangle \qquad P \vdash \mathit{find}(y) = \langle \mathit{defs}', \mathit{uses}' \rangle \qquad P \vdash \mathit{union}(x, y)}{P \vdash \mathit{find}(y) = \langle \mathit{defs} \cup \mathit{defs}', \mathit{uses} \cup \mathit{uses}' \rangle}$$

$$\frac{P \vdash \mathit{find}(x) = \langle \mathit{defs}, \mathit{uses} \rangle \qquad w_{std} = \langle \mathit{defs}', \mathit{uses}' \rangle \qquad P \vdash \mathit{unionStd}(x)}{P \vdash w_{std} = \langle \mathit{defs} \cup \mathit{defs}', \mathit{uses} \cup \mathit{uses}' \rangle}$$

Algorithm 1: Web analysis

and y belong respectively. Additionally, we define a special web containing escaping functions w_{std} and $\mathit{unionStd}(x)$ merges w_x with w_{std}.

The web analysis has the following components. First, for all function values that flow to a variable, we add those values to the variable's web. Then, if a variable x flows to another variable y, we unite the two webs since x and y must share the same calling convention. By the transitivity rule in CFA, all function values that flow to x necessarily flow to y. This step in the web calculation sets up the other direction — functions that flow to y, but not to x, must also be a consideration for deciding the calling convention for x. Finally, if a variable escapes or is used as a parameter to a function that is never bound, the variable is merged into the escaping web and has to use the standard calling convention.[1]

5.3 Control-flow Graph and Nesting Analysis

Another use of the control-flow results is to construct a static call graph. Since function application is an essential source of control transfer, having the flow information significantly improves the precision of the control-flow graph. Similar to the approach in Shao and Appel (2000), the decision process uses the control-flow graph to decide which free variables are prioritized.

Because of separate compilation, some call sites have unknown callees, and some functions escape the current compilation unit, making them callable from any location, including those unknown call sites in the same compilation unit. A realistic construction of a call graph may be to connect an "entry" node to all escaping functions and to place an edge from any call site potentially calling unknown functions to all

[1] If there is a call site that uses a variable that no values flow to, it means that the call site is unreachable. As an example, if there is a function that is never applied, the call to the function's return continuation will be such an unreachable call site. In these cases, the branch that ends with an unreachable call site may be removed, but doing so requires substantial transformation. For simplicity, we vacuously treat such a call site as if it called an unknown function.

escaping functions. This construction, although sound, obfuscates the true control-flow structure because it is highly unlikely that an escaping function is called from all unknown call sites. Since the purpose of this control-flow graph is to provide heuristics, we choose an alternative approach.

Similar to the extended call graph used in the current implementation (Shao and Appel, 2000), the *flow-direct extended* call graph is constructed as follows. The nodes in the call graph consist of *blocks* and a special node called "entry." Each block represents a maximal region in the IR without control transfer, a block is terminated by either a call or a branch. There is an edge between two blocks, b_1 and b_2, if any of the following conditions is true.

1. b_1 ends with a branch where b_2 is one of the successors.
2. b_1 calls x, $P \vdash f(x_1, \ldots, x_n) = e \longrightarrow x$, and b_2 is the first block in the function f.
3. b_1 calls a function x with k as its return continuation, no known function flows to x, and b_2 is the first block in the continuation k' where $P \vdash k'(x_1, \ldots, x_n) = e \longrightarrow k$.

Additionally, after applying the above three criteria, if a function f does not have an incoming edge, we set up an edge between the special "entry" node to the first block in f. The third criterion assumes that if one calls an unknown function, that function returns. Because SML/NJ supports first-class continuations in the form of call/cc, this assumption is not necessarily true. A function could discard its own return continuation and invoke an arbitrary one. Nonetheless, this behavior is rare in practice.

Two important traditional analyses are implemented on this *flow-directed extended* call graph. Using this construction, all nodes except the "entry" node have at least one incoming edge, and all nodes are reachable from the "entry" node. That is, the call graph is a single-entry graph, but since the control transfer is arbitrary, this graph is not guaranteed to be reducible. Therefore, Havlak's algorithm is used for loop nesting analysis (1997). In addition, at each branch, we use static branch prediction to analyze which branch is more likely to be executed (Wu and Larus, 1994).

5.4 Sharing Analysis

The current implementation performs sharing analysis *just in time* during the transformation. When emitting code for closure allocation, the transformation analyzes the approximated lifetime of each free variable. If there are more than a given number[2] of free variables whose lifetimes end together, it allocates a separate record for such a variable group. Since the sharing analysis occurs after the flattening is performed and callee-save registers are assigned, the sharing analysis only concerns the variables that have to be spilled. Another characteristic of the current sharing analysis is that the decisions are made *top-down*. The outer functions decide which variables to share, and the inner functions utilize the existing shareable records.

In the flow-directed approach, however, because the number of spread variables (*i.e.*, the arity of the environment) has to be negotiated among functions in the same web, it is beneficial for each function to perform sharing first, obtaining the "minimum required" arity for each web, before spreading decisions are made. In addition, the *top-down* approach often misses sharing opportunities due to the ambiguity of an integer-valued lifetime used by the current implementation.

Thus, we use the alternative approach shown in Algorithm 2. The new sharing analysis uses a *bottom-up* approach, processing the leaf functions in the abstract syntax tree first and propagating the decisions up. The analysis takes a program `program` $p(x_1, \ldots, x_n) = e$ and produces, a sharing-decision map $\mathcal{S} : \mathit{Fun} \longrightarrow \mathcal{P}(\mathit{Group}) \times \mathcal{P}(\mathit{Var})$, where $\mathit{Group} = \mathcal{P}(\mathit{Var})$. For each function f, $\mathcal{S}(f)$ indicates what variables

[2]The number currently used is 3.

```
fun analyze (⟦program p(x_1, ..., x_n) = e⟧) = inner e
and shares (f(x_1, ..., x_n) = e) =
  let val F = freevar (f(x_1, ..., x_n) = e)
      val I = inner e
      val I' = filter (fn G' => ∀v ∈ G', v ∈ F) I
      val (G, V) = cover (I', F)
      val (G', V') = group (V, depth f)
  in  S ← S[f ↦ (G' ∪ G, V)];
      G' ∪ G
  end
and inner (⟦g(a_1, ..., a_n)⟧) = {}
  | inner (⟦let y = x.i in e⟧) = inner e
  | inner (⟦let y = (a_1, ..., a_n) in e⟧) = inner e
  | inner (⟦if a then e_1 else e_2⟧) = inner e_1 ∪ inner e_2
  | inner (⟦fix f(x_1, ..., x_n) = e in e'⟧) =
      { shares (f(x_1, ..., x_n) = e) } ∪ inner e'
and group (V, d_0) =
  let val G = groupBy { (v, depth v) | v ∈ V }
      fun choose ((V, d), (G_0, V_0)) =
        if |V| ⩽ szthres orelse (d − d_0) ⩽ dpthres then
          (G_0, V ∪ V_0)
        else
          ({V} ∪ G_0, V)
  in  fold choose (∅, ∅) G
  end
```

Algorithm 2: Sharing Analysis.

are grouped together into a shareable record and which variables should stay in the function's immediate closure.

The sharing analysis assumes that the following two pieces of syntactic information are available:

- $\mathit{freevar} : \mathit{Fun} \to \mathcal{P}(\mathit{Var})$, which returns the set of free variables of a given function, *and*
- $\mathit{depth} : \mathit{Var} \to \mathbb{N}$, which returns the function-nesting depth of the binding site of a variable.

Furthermore, the analysis uses the following standard operations:

- $\mathit{groupBy} : \mathcal{P}(\mathit{Var} \times \mathbb{N}) \to \mathcal{P}(\mathit{Group} \times \mathbb{N})$, which, given a set of variable-integer pairs, groups the variables by the corresponding integers,
- *fold* and *filter*, defined in the standard way.

The function $\mathit{cover} : \mathcal{P}(\mathit{Group}) \times \mathcal{P}(\mathit{Var}) \to \mathcal{P}(\mathit{Group}) \times \mathcal{P}(\mathit{Var})$ takes a set of groups and a set of variables and *tiles* the set of variables with the set of groups. It returns the set of groups used and the remaining variables that cannot be covered by the chosen groups.

The helper function *inner* calls *shares* on all immediate inner functions and collects the results. For a leaf function, since *inner* returns an empty set, $\mathcal{I} = \mathcal{I}' = \mathcal{G} = \emptyset$ and $\mathcal{V} = \mathcal{F}$. It then invokes *group* with all of its free variables. The *group* function groups the free variables based on the nesting depths of their definition site, and removes the groups that are either too small or too close to the current function to be worth sharing. It uses the configurable parameter **szthres** to determine the minimum-size threshold for a shareable closure and the parameter **dpthres** for the minimum-distance threshold from the current function.

For a non-leaf function, *shares* gathers all the records that functions from a lower level would like to share and tries to propagate these shareable records up. It first disqualifies the groups that cannot be propagated any further; *i.e.*, the groups that contain a variable bound by the current function. For the remaining groups, it uses best-fit *cover*-ing heuristics to select the groups that tile the set of free variables. Resuming the same process as a leaf function, *shares* finally returns the union of the newly created groups and the propagated shareable groups.

After the analysis, an additional pruning phase is performed to remove the groups that are selected by too few functions, as determined by another configurable parameter: **usethres**.

6 Space-efficient Closure Decisions

The previous sections identify the landscape of sound closure decisions and facts about the input program, but how does one come up with a closure decision that satisfies the invariants and takes advantage of the gathered information? Although it is plausible to implement some procedure that calculates the triple $\langle \Gamma, L, \Phi \rangle$ from scratch, this section describes another approach. One can start with a starting decision known to be sound and safe for space — flat closures — and through a series of rewrites, iteratively improve the closure decisions.

At a high level, the decisions are rewritten to minimize the allocations for closure at each step in the pipeline. Starting from an initial, flat closure representation, the static sizes of all allocated closures are first reduced, using the results from the closure sharing analysis. Then, we aggressively spread heap-allocated closures into registers, as long as doing so does not increase the size of those that must be allocated. Each phase performs a simple pass over each of the three decision maps, and updates the decisions greedily — it does not backtrack a rewrite performed in a previous step even if it might be advantageous to do so. A diagrammatic summary of the decision pipeline and the analyses each stage depends on is shown

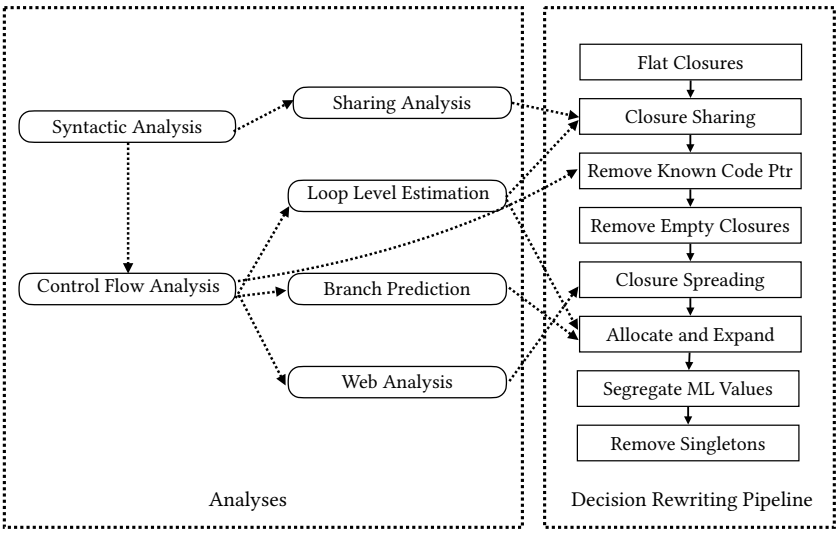

Figure 14: Analyses and decision rewriting pipeline.

in Figure 14. The dashed lines represent the dependencies between the analyses and each stage of the pipeline.

In this pipeline, the flow information is used in two related but distinct ways. First, it serves as heuristics. The flow analysis induces an approximated call graph, and loop nesting analysis and static branch prediction guide the decision pipeline in prioritizing certain variables. Second, and more importantly, the flow information ensures soundness. As the flow analysis provides an over-approximation of all functions that can be called at a call site, it can be used to enforce a consistent calling convention for all potential callees.

Flat Closures The start of the pipeline is a simple flat-closure decision. Each function represents its closure as a `Boxed` value, containing the function's code pointer followed by all of its free variables. Concretely, the following rule specifies the initial closure decision.

$$\frac{\texttt{fix}\ f(x_1,\ldots,x_n) = e\ \texttt{in}\ e' \in P \qquad \mathcal{F} = \langle \texttt{Var}(v) \mid v \in \mathit{freevar}(f(x_1,\ldots,x_n) = e)\rangle \qquad \mathit{env}\ \text{fresh}}{\vdash \langle \Gamma[f \mapsto \texttt{Boxed}(\mathit{env})], L[\mathit{env} \mapsto \texttt{Code}(f) :: \mathcal{F}], \Phi[f \mapsto \langle \mathit{env}\rangle]\rangle\ \textbf{initial}}$$

At the initial stage, each function allocates exactly one record and no sharing takes place.

Sharing After the initial stage, the sharing stage applies the sharing analysis result $\mathcal{S}$ as is, except for tail-recursive, leaf functions. For each function f, if the analysis indicates that its free variables can be grouped into shareable sets $\mathcal{G}$ with additional variables $\mathcal{V}$, the closure for f is represented using the groups $\mathcal{G}$, followed by the remaining variables $\mathcal{V}$. Each group is represented as an environment ID env, whose contents in the heap are the variables in the group. If the environment ID has not been allocated by f's enclosing function, f is responsible for its allocation. The sharing results are not applied to tail-recursive leaf functions because these functions presumably access their environment frequently. Additionally, since there are no nested functions in their bodies, they do not need to create new closures. Thus, having shareable records in such functions provides little benefit.

Remove Known Code Pointers If a non-escaping web contains only one function, there is no ambiguity at any call site about which function is being invoked. As a result, the closure for this function need not store its code pointer. During the transformation, the code pointer can be supplied directly whenever needed. This approach generalizes a common first-order optimization, where the code pointer is removed from the closure of a *first-order* function (Keep et al., 2012). Since the web of a first-order function always contains only one function, the generalized step subsumes the first-order optimization.

Remove Empty Closures After the removal of code pointers, a known function with no free variables results in an empty closure. In such cases, the function variable itself can be eliminated entirely from the program, akin to how functions are treated in first-order languages. This optimization reduces the sizes of every closure that closes over known, closed functions. Moreover, since the analysis of closure arities is performed at the level of webs, this optimization is applicable to higher-order functions as well. If a known, closed function is passed as an argument, the argument is removed and all relevant call sites are updated. This optimization implements the same ideas introduced by Dimock et al. (2001), who demonstrate the benefits of closure removal for closed, higher-order functions.

Analyze Arities and Spread After applying sharing and removing unnecessary components, at this point in the pipeline, closures have been reduced to their minimal size. The next step determines, for each web, whether the functions should use a spread representation and, if so, the number of variables that the closure can expand to.

This implementation takes a conservative approach to deciding when to spread closures. If a function f is free in another function g, g's closure must contain f. Spreading f increases the arity of g's closure, which can introduce additional overhead. Consequently, the analysis only decides to spread f if g can be spread as well. To extend this intuition for functions to webs, for any web $w = \langle \mathit{defs}, \mathit{uses} \rangle$, the functions in *defs* can use a spread representation if, for any $v \in \mathit{uses}$, all functions that close over v can be spread. Concretely, given a web w and a set of spreadable webs $\mathcal{W}$, the conservative spreadable criterion is defined as follows:

$$\mathit{spreadable}(w, \mathcal{W}) = w \in \mathcal{W} \vee \mathit{freeInSpreadable}(w)$$
$$\text{where } \mathit{freeInSpreadable}(\langle \mathit{defs}, \mathit{uses} \rangle) = \forall v \in \mathit{uses}, \forall f \in \{f \mid v \in \mathit{freevar}(f)\}, w_f \in \mathcal{W}$$
$$\mathit{spreadableWebs}(\mathcal{W}) = \{w \mid \mathit{spreadable}(w, \mathcal{W})\}$$

The set of spreadable webs is a fixed point of *spreadableWebs*. Two fixed points of this function are worth considering: the least fixed point and the greatest fixed point. The distinction arises when a variable in *uses* is free in a function in *defs* of the same web. This pattern occurs naturally in any non-tail-recursive functions. For instance, consider the example at the end of Section 4. The web containing the continuation variable k, $w_k = \langle \{k_0, k_1\}, \{k, k_0, k_1\} \rangle$, where k appears free in k_1. The least fixed-point solution avoids spreading w_k, while the greatest fixed point allows it. Spreading this web may eliminate the allocation for k_0 but can increase the live variables in k', potentially causing register spilling. The optimal answer likely lies between these extremes, balancing the allocation reduction with spill minimization as outlined by Chow (1988).

Once the analysis determines that a web should be spread, it must determine the exact arity for the web's closures. This calculation considers the size of each closure in the web, and the call sites where these variables must be passed. Because the runtime system of SML/NJ always passes arguments in registers, the number of available registers constrains the arity. Two policies are implemented for evaluation, a liberal policy, which uses the size of the largest closure in the web as its arity, and a conservative policy, using the minimum. The arity is then constrained by the number of available registers at all call sites where any variable in *uses* may be passed.

Once the spread arity for each web is determined, we trivially update the closure decisions to align with the arities. For a web that needs to be spread to k, the decisions are rewritten as follows. For each $f \in \mathit{defs}$, assuming its representation, $\Gamma(f)$, is $\texttt{Boxed}(\mathit{env})$, we pad the representation to k with `Nil` slots:

$$\texttt{Flat}(f_{\mathit{code}}, \mathit{env}, \overbrace{\texttt{Nil}, \ldots, \texttt{Nil}}^{k-1}).$$

For each $v \in \mathit{defs}$, if v appears in a record, we put the expansion slots in the record:

$$\begin{array}{ll} & H[\mathit{env} \mapsto \langle s_1, \ldots, \texttt{Var}(v), \ldots, s_n \rangle] \\ \longrightarrow & H[\mathit{env} \mapsto \langle s_1, \ldots, \texttt{Var}(v), \texttt{Expand}(v, 1), \ldots, \texttt{Expand}(v, k), \ldots, s_n \rangle]. \end{array}$$

Allocate and Expand After the simple spreading pass, this step addresses the allocation and expansion of variables within the spread representation. Spread closures initially contain placeholders (*i.e.*, `Nil`s) that are not directly useful until populated with meaningful values. For each function, this pass selects variables from the initial closure to fill these slots, prioritizing variables with a higher frequency of usage based on branch prediction. When multiple variables have the same frequency, it breaks ties by considering loop nesting levels, favoring variables used in deeper loops.

Additionally, for any $\texttt{Expand}(v, i)$ slots, if v is a syntactically known function, these expansion slots are replaced with the actual values that v allocates from its initial closure. This substitution can eliminate duplicates when multiple slots redundantly refer to the same variables and can remove unused slots altogether.

Segregate ML Values and Remove Singleton Records Finally, since the runtime system of SML/NJ currently does not support mixed records of uniform ML values and untagged "raw data" such as untagged integers or floating-point numbers, any raw data in a closure is separated into its own structure. For runtime systems that support mixed records, this step is unnecessary. As a last point of simplification, records containing only one slot are replaced with the slot itself, saving allocation and access indirection.

7 Evaluation

This new approach is implemented and integrated into the SML/NJ compiler, with a runtime switch that bypasses the existing closure converter. The new converter can compile a substantial amount of code, including the SML/NJ compiler itself. The configurable parameters described in Section 6 are implemented as command-line arguments, enabling easy adjustments in measurements. It is important to note that while the various analyses have been performance-tuned, the decision pipeline and the transformation are implemented naïvely. This section reports the performance of the new converter in comparison to the current converter and evaluates the quality of generated code under various configurations, including that of the current converter. The measurements were gathered on a server with a 2.60GHz Intel Xeon Gold 6142 CPU running Ubuntu with kernel version 5.15, using SML/NJ 110.99.6. To reduce noise and minimize the impact of outlier data from a cold start, all reported timing data represent the median of 10 consecutive runs.

7.1 Quality of Generated Code

The first part evaluates the quality of the generated code using two metrics: the space usage of the resulting program and its overall runtime. Four notable closure strategies are assessed, each stopping at a different point in the decision pipeline.

Table 2: Bytes allocated.

Program	Strategies: flat closure	sharing-only	liberal	conservative	**Old** (KB)
hamlet	117.95%	102.06%	100.01%	100.07%	**1,623,239**
vliw	139.59%	107.18%	105.02%	107.13%	**446,415**
nucleic	155.81%	113.09%	102.14%	126.58%	**15,682**
lexgen	145.22%	101.28%	101.59%	104.90%	**247,264**
barnes-hut	96.76%	**88.07%**	91.01%	92.07%	2,681,947
mc-ray	101.73%	100.53%	100.20%	100.32%	**42,736,228**
life	103.25%	100.61%	**77.45%**	101.91%	6,938,211
mandelbrot-rat	22.92%	10.69%	**9.52%**	38.30%	22,233,064
mandelbrot	0.00%	**0.00%**	**0.00%**	0.00%	134,228

The first one is "flat closures," which disables both closure sharing and flow-directed spreading. It is worth noting that some degree of spreading still occurs since, in SML/NJ, continuation closures uniformly use spread representations with three slots. Additionally, all strategies, including flat closures, perform known-free-variable lifting as described in Section 2, as well as known-code pointer removal and empty closure removal. The flat-closure strategy serves as a representative of a sophisticated baseline closure converter — one that uses flat closure representations but implements common, inexpensive analyses.

The second strategy, "sharing-only," enables closure sharing but does not apply any additional flow-directed spreading. This strategy corresponds to a converter that employs *safely linked* closures (Shao and Appel, 1994) with aggressive sharing.

The last two strategies perform all steps in the pipeline, including both closure sharing and flow-directed spreading. The difference between them lies in the policy used to determine the arity within a web during the spreading analysis. The "liberal" strategy picks an arity based on the function with the largest environment. This policy ensures that no functions need to spill their environments, but it may require some functions to pad their slots with placeholders, potentially increasing register usage. In contrast, the "conservative" strategy picks an arity based on the function with the smallest environment. While this policy eliminates unused slots, it may require other functions in the web to heap-allocate parts of their environments. The columns labeled **Old** represent the current closure converter for comparison.

7.1.1 Space

The first aspect of the code quality is its space efficiency, which is the focus of this thesis and the metric that the heuristics optimize for. To analyze the space usage, the garbage collector in SML/NJ was instrumented to collect the two metrics presented in this section. Before each measurement, the entire heap was garbage-collected and the counters reset. Once the program returned, the counters were read and their values were presented here.

The first metric is the number of bytes allocated by each program, presented in Table 2. To provide context, the allocation sizes for the "old" implementation are presented, in kilobytes, and the allocation sizes for other implementations are expressed as a percentage of the old implementation. The numbers in **boldface** are the lowest number among the configurations. Since SML/NJ allocates all closures on the heap, the closure representations have a direct impact on the allocation count.

The second metric is the number of bytes *promoted* to the first generation, shown in Table 3. In a generational garbage collection, objects are allocated in the zeroth generation, *a.k.a.* the nursery, a minor collection occurs when the nursery fills up. During a minor collection, live objects are promoted to the first generation. The work required during a minor collection is proportional to the volume of live data.

Table 3: Bytes promoted.

Program	Strategies: flat closure	sharing-only	liberal	conservative	**Old** (KB)
hamlet	129.99%	101.72%	99.97%	**93.64%**	60,249
vliw	141.88%	103.69%	96.88%	**93.27%**	4,454
nucleic	128.51%	112.50%	**89.57%**	110.43%	143
lexgen	107.60%	99.77%	**99.41%**	100.57%	3,585
barnes-hut	100.28%	**98.98%**	99.62%	99.53%	36,916
mc-ray	102.48%	**99.12%**	100.09%	99.51%	59,180
life	102.69%	103.19%	**84.94%**	99.59%	16,042
mandelbrot-rat	38.65%	17.00%	**15.81%**	67.26%	2,419
mandelbrot	**0.00%**	**0.00%**	**0.00%**	**0.00%**	12

Minimizing the number of bytes promoted reduces the cost of each minor collection. Conversely, promoting a large number of objects increases the workload of major garbage collection cycles, as more objects need to be scanned and forwarded.

For certain programs, such as "mandelbrot" and "mandelbrot-rat,"[3] this approach significantly reduces allocations across both metrics. The Mandelbrot examples have a very simple control-flow pattern — the main loop consists of three nested tail-recursive functions. In the floating-point-number version, the new closure conversion algorithm removes all closure allocations and makes the allocation *constant*, regardless of the problem size. In the rational-number version, only the return continuations for non-tail-recursive functions allocate closures.

The combination of aggressive sharing and flattening is effective in reducing the number of bytes promoted out of the nursery. In all programs, at least one of these strategies achieves a reduction in promoted bytes compared to the old implementation. The total number of bytes allocated using the new approach is close to, but often slightly higher than, the old implementation. The exact cause of this slight increase in allocation requires further investigation, but it is likely caused by some unintended consequence of increased register usage. For example, since the garbage collector requires the root set to be stored in registers, if there are more live data than there are registers, a heap record may be allocated to hold the excess. Such a temporary heap record contributes to the total bytes allocated but is unlikely to increase the number of bytes promoted.

The results also underline the limit of greedy approaches. Although all strategies that enable sharing outperformed the flat closure strategy, there is no one clear winner among those three.

7.1.2 Time

Another important dimension of code quality is the overall runtime of each program. Table 4 presents the execution time of each program, compiled under different configurations. The times are reported in seconds and represent the median of 10 consecutive runs. The smallest execution time for each program is highlighted in **boldface**.

The results demonstrate that the relationship between space usage and execution time is not direct. While reducing allocation typically lowers the number of garbage collection invocations, execution times depend on several other factors. These include the length of the access path for a frequently used free variable, the quality of the register allocator, and the cache locality of the generated machine code. For example, the conservative configuration produces the most performant code for "mandelbrot-rat," despite being the

[3] "mandelbrot" and "mandelbrot-rat" both calculates the Mandelbrot set. The former uses floating-point numbers and the latter uses rational number approximations represented as two integers.

Program	Strategies				Old
	flat closure	sharing-only	liberal	conservative	
hamlet	0.176	0.154	0.146	**0.142**	0.148
vliw	0.069	**0.062**	0.064	0.065	0.064
nucleic	**0.002**	**0.002**	**0.002**	**0.002**	**0.002**
lexgen	0.034	0.031	0.032	0.031	**0.030**
barnes-hut	**0.535**	0.541	0.569	0.570	0.539
mc-ray	9.559	9.612	9.614	9.624	**7.814**
life	2.839	2.949	3.013	2.676	**2.577**
mandelbrot-rat	9.613	9.441	9.585	**9.125**	10.569
mandelbrot	5.068	5.609	5.532	5.098	**5.031**

Table 4: Execution time (seconds).

configuration that allocates the most memory among the new strategies. This evaluation highlights that optimizing for the overall execution time requires decisions based on a more nuanced and comprehensive policy.

7.2 Performance of the Closure Converter

The second part evaluates the compile-time impact of the new closure converter. Table 5 summarizes the results. The first column lists the names of the programs, and the second column shows the lines of code in each program, excluding blank lines and comments. The next six columns, under the header **Phases**, show the runtime of each major phase in the new converter in milliseconds. Specifically,

- "cfa" indicates the propagator-based control-flow analysis (Section 5.1);
- "web" denotes the web analysis on the results of "cfa" (Section 5.2);
- "cfg" is the combination of the extended control-flow graph construction, loop-nesting analysis, and static branch prediction (Section 5.3);
- "shr" records the bottom-up sharing analysis (Section 5.4);
- "pipe" captures the entire closure decision pipeline, including the flattening analysis (Section 6);
- "trans" represents the transformation that applies the decisions produced by the pipeline (Section 4).

The symbol ϵ is displayed if the phase takes less than one millisecond. The two columns under the header **Closure** report the time spent on the entire closure conversion; "old" corresponds to the existing closure converter while "new" refers to the approach introduced in this paper, expressed as a factor of the time spent in "old." Notably, the times reported for individual phases do not necessarily add up to the total time because additional minor passes, such as free-variable and census analyses, contribute to the overall timing. The final two columns, under the header **Total**, report the end-to-end compile time.

As expected, the new approach incurs a slowdown compared to the existing closure converter because of the additional analyses it performs, many of which are algorithmically expensive. The variability in the slowdown, however, tells a more nuanced story. First, the performance impact of this new, flow-based approach seems to correlate with the program's degree of *higher-orderness* and the complexity of its control flow. For most programs, the new implementation takes roughly twice as long as the existing one. An exception to this pattern is "mc-ray," which has a 3.7x slowdown. This program uses higher-order functions to implement an object-oriented programming pattern, likely causing more merging in the

Table 5: Performance of the new closure converter (milliseconds).

		Phases						Closure		Total	
Program	**LOC**	cfa	web	cfg	shr	pipe	trans	old	new	old	new
sml-nj	309,001	809	479	479	1,369	2,207	2,155	5,308	2.0x	71,862	1.1x
hamlet	16,936	2,841	1,635	1,708	481	510	651	1,862	4.4x	10,044	2.8x
vliw	3,032	76	17	26	46	72	88	296	1.3x	1,546	1.7x
nucleic	2,920	1	ϵ	ϵ	ϵ	2	2	3	1.9x	375	0.96x
lexgen	1,035	9	3	11	9	15	13	41	1.8x	478	1.2x
barnes-hut	937	4	1	6	8	10	8	22	2.0x	294	1.0x
mc-ray	542	1	ϵ	ϵ	1	3	4	5	3.7x	101	1.4x
life	114	ϵ	ϵ	ϵ	ϵ	1	1	2	1.7x	84	0.83x
mandelbrot-rat	95	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	2	1.8x	59	0.65x
mandelbrot	48	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	2.0x	14	1.5x

control-flow analysis. The imprecision caused by merging, in turn, results in a more complicated control-flow graph and web structure. Consequently, the convergence of fixed-point algorithms may take longer. Second, the performance impact of the new approach sometimes extends beyond the closure converter itself. For example, in the "hamlet" program, which is an implementation of Standard ML concatenated into a single file (Rossberg, 2001), the new implementation adds 6.4 seconds to closure conversion time, yet the total compilation time increases by about 18 seconds. Further analysis of the downstream phases reveals that register allocation is significantly affected: while the compiler using the old closure converter spends only 3.56 seconds in register allocation, the same back end with the new converter spends 15.13 seconds. This indicates that flattening decisions can introduce additional complexity into later stages of compilation.

On average, across the benchmark programs, the new implementation causes a 1.33x slowdown in the compiler's end-to-end performance. The slowdown remains consistent regardless of the input program size, demonstrating that the new implementation is both practical and scalable.

8 Related Work

This work is inspired by and builds upon the existing closure converter in SML/NJ (Shao and Appel, 1994, 2000), with which this paper offers many points of comparison throughout.

Control-flow analysis has been the subject of extensive investigation (Midtgaard, 2012), and is used in a number of functional compilers, such as MLton (MLton, 1999; Ziarek et al., 2008), Bigloo (Serrano, 1992), and more recently 3CPS (Quiring et al., 2022). Using higher-order control-flow information to optimize function representation is studied in the context of defunctionalization (Cejtin et al., 2000; Dimock et al., 2001; Pottier and Gauthier, 2006; Brandon et al., 2023). Defunctionalization and environment representation are orthogonal problems. A function value consists of both a tag — concretely, a code pointer — and an environment. While defunctionalization analyzes and optimizes the code used to represent a higher-order function, the environment representation optimizes the layout of the environment.

Lightweight closure conversion (Wand and Steckler, 1994; Siskind, 1999) is a well-known technique to reduce the sizes of closures. At a high level, the observation made by lightweight closure conversion is that if a function f closes over a variable x, and x is not rebound before all calls to f, f's closure need not contain x, but x can be supplied as an additional argument. Similar to this work, lightweight closure conversion takes advantage of the fact that a function's environment is passed as a parameter, if we know all call sites of a function, we can arrange the calling convention to supply the environment in customized ways. This work, however, does not attempt to reduce the sizes of closures. All closures contain the same

elements, but rather the closure "passing" is arranged in different ways.

Arity raising is a well-known technique used to flatten a boxed representation (*e.g.*, a record) into an unboxed representation (*e.g.*, multiple variables). Performing arity raising safely depends on the compiler's knowledge of the definition and use sites of each record. A compiler can obtain this knowledge by *types* (Leroy, 1992, 1997; Shao, 1997b; Peterson, 1989) or by *flows* (Ziarek et al., 2008; Bergstrom and Reppy, 2009). Callee-save registers can be seen as an instance of type-based arity raising if one considers continuations as a type separate from user-defined functions.

As discussed in Section 4.1, closure records behave much more regularly than user-defined records. It is this behavior that allows a typed IR to assign an existential type to a closure (Tarditi et al., 1996). In both Manticore (Bergstrom and Reppy, 2009) and MLton (Ziarek et al., 2008), when a flattened record is contained in another record that cannot be flattened, the flattened record is reinstated as a boxed record. By contrast, when a spread closure record is contained in a boxed closure record, as is performed in Section 6, the latter closure record can just modify its layout, because all of its use sites expect only one layout.

The correctness and space safety of closure conversion is studied by Paraskevopoulou and Appel (2019) and Sullivan et al. (2023). This paper does not offer proof of correctness or safety, but the transformation and the closure representation are designed to facilitate formal proof.

9 Conclusion and Future Work

This paper introduces a new approach to optimizing closure representations in higher-order languages. It formulates a framework for expressing environment representation decisions and provides a transformation that executes these decisions. It presents an efficient higher-order control-flow analysis, along with other related analyses, to support decision-making by leveraging flow information. It presents a decision pipeline that incrementally improves on a flat-closure decision. This approach is evaluated in the context of the SML/NJ compiler, showing the scalability and effectiveness.

This paper raises several research questions for further exploration. First, while the heuristics implemented here effectively reduce the allocation rate of programs, the evaluation also shows that more nuanced policies are necessary to enhance overall performance. One potential avenue is to make closure representation decisions based on the usage patterns of free variables. Some free variables are used directly in computations, benefiting from the immediacy of a spread representation, while others are primarily used to construct other closures, favoring a nested representation. The current greedy approach, which nests first and spreads afterward, does not differentiate between these scenarios. Another direction involves analyzing if a closure is "construction-dominant" or "use-dominant." By leveraging the approximated control-flow graph, one could estimate the relative frequency of function definitions versus function applications. A linked closure minimizes construction costs, while a flat closure is more efficient for retrieving variable bindings.

Second, the correctness and the space safety of this approach require formal proof, beginning with the verification of the transformation itself. Since the transformation follows a similar structure to the one proposed by Paraskevopoulou and Appel (2019), similar arguments may apply. The proof of correctness relies on the property that the closure decisions provide all necessary bindings for the function's body, while the proof of space safety requires that only the necessary bindings are reachable from the closure. The pipeline must also be shown to uphold these properties. An inductive argument can achieve this by showing that the flat closure decision satisfies the constraints — which is a known result — and that each rewriting step respects these constraints.

References

Norman Adams, David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, and James Philbin. 1986. Orbit: An Optimizing Compiler for Scheme. *ACM SIGPLAN Notices* 21, 7 (1986), 219–233.

Connor Adsit and Matthew Fluet. 2014. An Efficient Type- and Control-Flow Analysis for System F. In *Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages*. Association for Computing Machinery, New York, NY, USA, 1–14.

Andrew W Appel. 1990. A Runtime System. *Lisp and Symbolic Computation* 3, 4 (1990), 343–380.

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

Andrew W Appel and Trevor Jim. 1989. Continuation-passing, Closure-passing Style. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. Association for Computing Machinery, New York, NY, USA, 293–302.

Andrew W Appel and David B MacQueen. 1991. Standard ML of New Jersey. In *International Symposium on Programming Language Implementation and Logic Programming*. Springer, Springer-Verlag, New York, NY, USA, 1–13.

Andrew W Appel and Zhong Shao. 1992. Callee-save Registers in Continuation-passing Style. *Lisp and Symbolic Computation* 5 (1992), 191–221.

Jeffrey M Bell, Françoise Bellegarde, and James Hook. 1997. Type-driven Defunctionalization. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*. Cambridge University Press, Cambridge, England, UK, 25–37.

Lars Bergstrom and John Reppy. 2009. Arity Raising in Manticore. In *International Symposium on Implementation and Application of Functional Languages*. Springer, Springer-Verlag, Berlin, Heidelberg, 90–106.

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.

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 *Programming Languages and Systems: 9th European Symposium on Programming, ESOP 2000 Berlin, Germany, March 25–April 2, 2000*. Springer, Berlin Heidelberg, 56–71.

Fred C Chow. 1988. Minimizing Register Usage Penalty at Procedure Calls. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA). Association for Computing Machinery, New York, NY, USA, 85–94.

Maheen Riaz Contractor and Matthew Fluet. 2020. Type- and Control-Flow Directed Defunctionalization. In *IFL 2020: Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages (IFL '20)* (September 2–4, 2020, Canterbury, United Kingdom). Association for Computing Machinery, New York, NY, USA, 79–92.

Allyn Dimock, Ian Westmacott, Robert Muller, Franklyn Turbak, and Joseph Brian 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*. Association for Computing Machinery, New York, NY, USA, 14–25.

Lal George, Florent Guillame, and John H Reppy. 1994. A Portable and Optimizing Back End for the SML/NJ Compiler. In *International Conference on Compiler Construction*. Springer, Springer, Berlin Heidelberg, 83–97.

David Glasser. 2013. *An Interesting Kind of JavaScript Memory Leak*. Meteor. `https://blog.meteor.com/an-interesting-kind-of-javascript-memory-leak-8b47d2e7f156`

Paul Havlak. 1997. Nesting of Reducible and Irreducible Loops. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19, 4 (1997), 557–567.

Nevin Heintze and David McAllester. 1997. Linear-time Subtransitive Control Flow Analysis. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming Language Design and Implementation* (Las Vegas, NV, USA). Association for Computing Machinery, New York, NY, USA, 261–272.

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). Association for Computing Machinery, New York, NY, USA, 30–35.

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

Xavier Leroy. 1992. Unboxed Objects and Polymorphic Typing. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. Association for Computing Machinery, New York, NY, USA, 177–188.

Xavier Leroy. 1997. The Effectiveness of Type-based Unboxing. In *TIC 1997: Workshop Types in Compilation* (Amsterdam, Netherlands). HAL, Lyon, France.

Jan Midtgaard. 2012. Control-flow Analysis of Functional Programs. *ACM computing surveys (CSUR)* 44, 3 (2012), 1–33.

Jan Midtgaard and David Van Horn. 2009. Subcubic Control Flow Analysis Algorithms. *Computer Science Research Report* (2009), 1–35.

MLton 1999. *MLton, a whole program optimizing compiler for Standard ML*. MLton. `http://mlton.org`

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

Zoe Paraskevopoulou and Andrew W Appel. 2019. Closure Conversion is Safe for Space. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–29.

John Peterson. 1989. Untagged Data in Tagged Environments: Choosing Optimal Representations at Compile Time. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*. Association for Computing Machinery, New York, NY, USA, 89–99.

François Pottier and Nadji Gauthier. 2006. Polymorphic typed defunctionalization and concretization. *Higher-Order and Symbolic Computation* 19 (2006), 125–162.

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

Andreas Rossberg. 2001. *HaMLet*. Max Planck Institute for Software Systems. https://people.mpi-sws.org/~rossberg/hamlet/

Manuel Serrano. 1992. *Bigloo, a Practical Scheme Compiler*. Inria. http://www-sop.inria.fr/indes/fp/Bigloo/

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. 1997a. An Overview of the FLINT/ML Compiler. In *Proc. 1997 ACM SIGPLAN workshop on types in compilation*, Vol. 156. ACM New York, NY, USA, New York, NY, USA, 156–158.

Zhong Shao. 1997b. Flexible Representation Analysis. *ACM SIGPLAN Notices* 32, 8 (1997), 85–98.

Zhong Shao and Andrew W Appel. 1994. Space-efficient Closure Representations. *ACM SIGPLAN Lisp Pointers* 7, 3 (1994), 150–161.

Zhong Shao and Andrew W Appel. 2000. Efficient and Safe-for-space Closure Conversion. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22, 1 (2000), 129–161.

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 Mark Siskind. 1999. *Flow-directed Lightweight Closure Conversion*. Technical Report. Technical Report 99-190R, NEC Research Institute, Inc.

Zachary J Sullivan, Paul Downen, and Zena M Ariola. 2023. Closure Conversion in Little Pieces. In *Proceedings of the 25th International Symposium on Principles and Practice of Declarative Programming*. Association for Computing Machinery, New York, NY, USA, 1–13.

D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. 1996. TIL: A Type-directed Optimizing Compiler for ML. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation* (Philadelphia, Pennsylvania, USA) *(PLDI '96)*. Association for Computing Machinery, New York, NY, USA, 181–192. doi:10.1145/231379.231414

Robert Endre Tarjan. 1983. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

David Van Horn and Matthew Might. 2010. Abstracting Abstract Machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. Association for Computing Machinery, New York, NY, USA, 51–62.

Mitchell Wand and Paul Steckler. 1994. Selective and Lightweight Closure Conversion. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. Association for Computing Machinery, New York, NY, USA, 435–445. doi:10.1145/174675.178044

Youfeng Wu and James R Larus. 1994. Static Branch Frequency and Program Profile Analysis. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*. Association for Computing Machinery, New York, NY, USA, 1–11.

Lukasz Ziarek, Stephen Weeks, and Suresh Jagannathan. 2008. Flattening Tuples in an SSA Intermediate Representation. *Higher-Order and Symbolic Computation* 21 (2008), 333–358.