Functioning without Closure: Type-Safe Customized Function Representations for Standard ML

Allyn Dimock[*†]
Harvard University
dimock@das.harvard.edu

Ian Westmacott[*†]
Boston University
ianw@bu.edu

Robert Muller[‡†§]
Boston College
muller@cs.bc.edu

Franklyn Turbak[‡†]
Wellesley College
fturbak@wellesley.edu

J. B. Wells[*¶‡†‖]
Heriot-Watt University
http://cee.hw.ac.uk/~jbw/

ABSTRACT

The CIL compiler for core Standard ML compiles whole ML programs using a novel typed intermediate language that supports the generation of type-safe customized data representations. In this paper, we present empirical data comparing the relative efficacy of several different flow-based customization strategies for function representations. We develop a cost model to interpret dynamic counts of operations required for each strategy. In this cost model, customizing the representation of closed functions gives a 12–17% improvement on average over uniform closure representations, depending on the layout of the closure. We also present data on the relative effectiveness of various strategies for reducing *representation pollution*, i.e., situations where flow constraints require the representation of a value to be less efficient than it would be in ideal circumstances. For the benchmarks tested and the types of representation pollution detected by our compiler, the pollution removal strategies we consider often cost more in overhead than they gain via enabled customizations. Notable exceptions are *selective defunctionalization*, a function representation strategy that often achieves significant customization benefits via aggressive pollution removal, and a simple form of *flow-directed inlining*, in which pollution removal allows multiple functions to be inlined at the same call site.

[*]Partially supported by NSF CCR grant 9417382.

[†]Partially supported by Sun grant EDUD-7826-990410-US.

[‡]Partially supported by NSF EIA grant 9806745/9806746/9806747/9806835.

[§]Partially supported by a Faculty Fellowship of the Carroll School of Management, Boston College.

[¶]Partially supported by NSF CCR grant 9988529.

[‖]Partially supported by EPSRC grants GR/L 36963 and GR/L 15685.

ICFP'01, September 3-5, 2001, Florence, Italy.

1. INTRODUCTION

The efficiency of the object code generated by a compiler depends to a large extent on the compiler's ability to select efficient target language representations of values manipulated by the source program. For modern programming languages that make heavy use of functions and methods, such as Standard ML or Java, efficient representation of functions and function calling protocols is particularly important.

In this paper, we report on the performance of the CIL[1] compiler for Standard ML [14, 16]. The CIL compiler is a type- and flow-directed whole-program compiler which is designed to generate type-safe function representations that are customized for the contexts in which they are created and applied. The compiler employs a novel typed intermediate language [44] that integrates polyvariant flow information directly in the intermediate representation. The compiler generates code for the SPARC V8 architecture.

The CIL compiler has the following key features:

- It performs flow-based representation customizations in a type-directed compiler. In this paper, we restrict our attention to customizing function representations, though the CIL framework supports customizing any type of data. The intermediate representations generated within the compiler are guaranteed to be well-typed with respect to a type system that encodes flow information. Thus, we address both efficiency and reliability.
- It is parameterized with respect to a function representation strategy. We have implemented seven such strategies. In this paper, we evaluate the relative efficacy of these strategies on a variety of benchmarks based on a cost model for function representations. We focus on *selective* strategies in which closed functions (those without free variables) are represented more efficiently than open functions (those with free variables). Other customizations supported by our framework are tagged environment representations of functions (in the context of inlined open functions and defunctionalization) and flow-based optimizations of known function calls.

[1]"CIL" is an acronym for "Church Intermediate Language." The authors are members of the Church Project (`http://types.bu.edu/`). The Church Project is investigating applications of sophisticated type systems in the implementation of higher-order typed programming languages.

- It is parameterized with respect to a flow analysis. We have implemented four typed flow analyses that vary with respect to precision. All are polvariant on type and one is polyariant on variable occurrences. All of the data reported in this paper is derived from instrumented code generated from runs of the compiler using only the *typed source split* analysis. This is our most accurate analysis that is polyvariant on types but not on variable occurrence. It is comparable to a typed 0CFA on monomorphized code [4] restricted by a shallow subtyping rule [44].

- It can generate customized function representations even in the presence of *representation pollution*. Pollution of function representations occurs when a function is constrained to have a less efficient representation than it otherwise would because it shares an application site with an inefficiently represented function. A complementary phenomenon occurs for function representations with pollution of application sites. Although we focus on functions here, representation pollution can occur with any data type. The CIL compiler can remove pollution by (1) generating multiple and mutually incompatible customized representations of value definition and use forms and (2) introducing sufficient "plumbing" to ensure that only compatible representations flow together.

Recently, there has been growing interest in using typed intermediate languages to ensure the integrity of complex program transformations such as closure conversion [25, 26]. Our approach is similar to the customization strategies used by type-based compilers that remove polymorphic higher-order functions via monomorphization and defunctionalization [39, 8]. These compilers maintain type correctness during closure conversion by injecting closures with different free variables that flow to the same application site into a sum-of-products datatype, and performing a case dispatch on the constructed value at the application site.[2] As in the CIL compiler, these compilers use flow analysis to customize function representations for particular application sites. These flow analyses are not integrated into their type systems, although after monomorphization and defunctionalization a flow analysis can be implicit in their types.

Customization by duplication of value construction points is sometimes called *cloning* [11]. Among other applications, it has been used for implementing lazy functional languages [17], for resolving overloading in Haskell [23], for compiling NESL [6], and particularly for optimizing method invocation in object-oriented languages [9, 10, 1, 13, 31, 30].

Our Results

The CIL compiler emits instrumented code that tracks the creation and application of functions as well as the plumbing associated with pollution removal. We have developed a cost model that assigns a dynamic cost to these points and have used it to compare the costs of our function representation strategies on a benchmark suite.[3] We have experimented with two basic closure representations: (1) *paired closures* that pair a code pointer with an environment record; and (2) *flat closures* that combine the code pointer and environment values in a single record.

We consider two function representation strategies that perform no pollution removal: (1) a *uniform* strategy in which all functions are represented by closures; and (2) a *polluted selective* strategy that represents closed functions as code pointers only when no additional plumbing is needed to do so. Under our cost model, our data show that compiling whole SML programs with the *polluted selective* strategy yields SPARC code in which function representation costs are 12% lower with flat closures, and 17% lower with paired closures, than those observed with the uniform strategy.

For the benchmarks tested and the types of representation pollution detected by our compiler, our data show that the plumbing costs associated with removing pollution often outweigh the benefits of using more efficient function representations. In particular, three pollution-removing strategies (what we call *selective source split*, *selective sink split*, and *uniform defunctionalization*) rarely perform better than — and in many cases perform significantly worse than — the *polluted selective* strategy.

In two other strategies, pollution removal often results in a net gain, sometimes significant, compared to *polluted selective*. In *selective defunctionalization*, aggressive splitting of application sites enables many customizations, and the function representation costs are often 10% or more better (and in one case 46% better) than *polluted selective*. In the *inlining* strategy, flow-directed inlining [45] allows certain functions to be represented as tagged environments and splitting of application sites allows multiple functions to be inlined at the same site. As with selective defunctionalization, the gains for inlining can be impressive. However, in some cases these two strategies have much higher costs than *polluted selective*. In general, pollution costs are relatively larger in flat than in paired closures.

The remainder of this paper is organized as follows. Section 2 provides an overview of the function representations used in the CIL compiler. Section 3 gives a brief overview of the CIL compiler. Section 4 presents our cost model. Section 5 presents run-time statistics for several standard benchmark programs. Section 6 summarizes our conclusions and describes future work.

2. REPRESENTATION CUSTOMIZATION AND POLLUTION

An essential invariant maintained by any compiler is that the representation chosen for a run-time value at its point of definition must be consistent with the representation expected at every point where that value is used. We call this the *representation invariant*. The simplest way to satisfy this invariant is to adopt a *uniform representation assumption (URA)*, under which the representation of any value is determined by the type constructor of its type and all values are accessed through a fixed-size interface (achieved by *boxing* values larger than one machine word). The URA simplifies the task of compiler writing by using a type system as a crude form of flow analysis. Type soundness guarantees that a value reaching an elimination form for a type constructor must have been defined at an introduction form for that constructor. If the uniform representations chosen for

[2] Some defunctionalizing compilers avoid this run-time cost by using the appropriate code pointer as a "tag" in the generated object code and replacing the case dispatch by a jump, but their type systems do not support this as a well typed operation and hence this must be done in the code generator after types are erased.

[3] We report on compile-time space costs of the CIL compiler in [16].

the introduction and elimination forms are consistent, then the representation invariant will automatically be satisfied.

A classic example of the URA is the representation of functions. Compilers for languages with higher-order functions must at some point in the compilation process convert every open function (one with free variables) into a closed function (one with no free variables). One way to accomplish this in a uniform manner is to represent every function uniformly as a *closure*, which pairs (1) a *code pointer* to a closed function with (2) an *environment* containing the values of the free variables of the function. Each code pointer addresses a function that expects an argument that pairs (1) the argument of the original function with (2) the environment of the closure. The process of transforming all functions into closures is known as *closure conversion*.

In this discussion, we will compare closures with several other function representations in the context of a simple concrete example. Figure 1(a) illustrates a fragment from a higher-order functional program.[4] The fragment shows three function abstractions (λ) and two application sites ($@$). These pieces are represented graphically to highlight how the function values created at abstraction sites flow to application sites. As in CIL, some sites are annotated with *flow labels* that approximate the flow of values in the program.[5] In the λ^l_ψ notation, l is a *source label* identifying a function definition point and ψ is a set of *sink labels* conservatively approximating those call sites at which the function labeled l may be used. Dually, the notation $@^\phi_k$ specifies a sink label k identifying a function use point and a set of source labels ϕ conservatively approximating the functions that may reach that point. In the figure, the abstraction bodies P, Q, R are superscripted with the set of free variables they reference (not including the abstraction parameter). Thus, $\lambda^3_{\{5\}}$ is an open function with free variables a and b, while $\lambda^1_{\{4\}}$ and $\lambda^2_{\{4,5\}}$ are closed functions.

The result of closure converting the fragment in Figure 1(a) is shown in Figure 1(b). Both closures and environments are represented as tuples (parenthesized sequences of elements separated by commas). Projections from these tuples are implicit in the destructuring notation for λ and `let` bindings. Although we show only untyped terms in the example, the terms are typable, albeit with some difficulty. In a naive approach, closure converting two functions of the same type can yield closures whose types differ due to differences in free variables exposed by the transformation. This problem is usually addressed with existential types [25, 27, 12]. In contrast, CIL uses intersection and union types to solve this problem; as argued in [44], this has advantages over existential quantifiers for customizing data representations.

The URA simplifies compiler writing and facilitates the support of features like type polymorphism, separate compilation, and dynamic linking. However, because it effectively requires worst-case representations to be used for all values, the URA stands in the way of customized representations and classical optimizations. For instance, in Figure 1(b), the closed functions $\lambda^1_{\{4\}}$ and $\lambda^2_{\{4,5\}}$ must be represented as closures with empty environments, because all application sites have been compiled assuming this representation. Ideally, we would prefer to use what Wand and Steckler call a *selective* function representation, in which closed functions are represented as code pointers and function invocations are implemented via a jump to this code pointer [41]. This avoids allocating and projecting from a closure pair.

The key difficulty with using such customized representations is that care must be taken that they do not invalidate the representation invariant. One approach to customization is to relax the URA by adopting *single representation assumption (SRA)*, in which a single representation is independently chosen for each definition point and each use point. Although the SRA would appear to allow flexibility in choosing customized representations, it has two problems:

1. The single representation at a definition point must be consistent with all use points where it could be used. Dually, the single representation at a use point must be consistent with definition points defining a value that could be used there. These constraints imply that there must be a more precise notion of where a value flows than "all sites at which a value of the same type is used". Thus, some form of flow analysis [33, 21, 28, 22] is required to perform representation customization.

2. An inefficient representation at definition point D_1 of a program can propagate through a program in a way that constrains the choice of representation at a distant definition point D_2, where D_1 and D_2 do not even directly share a use point. (A dual problem holds for use points that do not share a definition point.) Define a *colleague* of a definition point D recursively as either the point itself, or any definition point that shares a use point with a colleague of D. Then by the constraints of the SRA and the representation invariant, all colleagues must share the same representation. Figure 1(b) provides a simple example of this. Even though closed abstractions $\lambda^1_{\{4\}}$ and $\lambda^2_{\{4,5\}}$ share an application site ($@^{\{1,2\}}_4$), they cannot be represented as code pointers because $\lambda^2_{\{4,5\}}$ shares an application site with open function $\lambda^3_{\{5\}}$. By the SRA, this constrains $\lambda^2_{\{4,5\}}$ to be represented as a closure, which in turn forces $\lambda^1_{\{4\}}$ to be represented as a closure.

   We call the ability of one "bad" representation to "spoil" the representation of all its colleagues the *representation pollution* problem.[6] Representation pollution is a serious obstacle to customizing representations in a compiler. For instance, the selective customization suggested by Wand and Steckler [41] can only be applied when all colleagues of a closed function are closed.

In the CIL compiler, our solution to pollution is to break the SRA constraints by *splitting* some definition and/or use points to use multiple representations. Figure 1(c) shows the

[4] The syntactic notation used in the example is a stylized notation chosen to simplify the presentation of the example. This notation differs significantly from "real" CIL, as described in [16]. In particular, typed CIL terms would require significantly more annotations in order to be type correct. Even untyped CIL terms would be more verbose than the notation in the figure, since CIL does not support destructuring in variable binding positions.

[5] In the CIL compiler, *every* introduction and elimination form, as well as every type constructor, is annotated with flow labels. To reduce visual clutter, we only highlight function value labels in the example.

[6] Other names for this problem are the *poisoning problem* [42] and the *"W" problem* [Jens Palsberg, personal communication]. The latter is named from the shape of a simple flow diagrams, like Figure 1, that illustrate the problem.

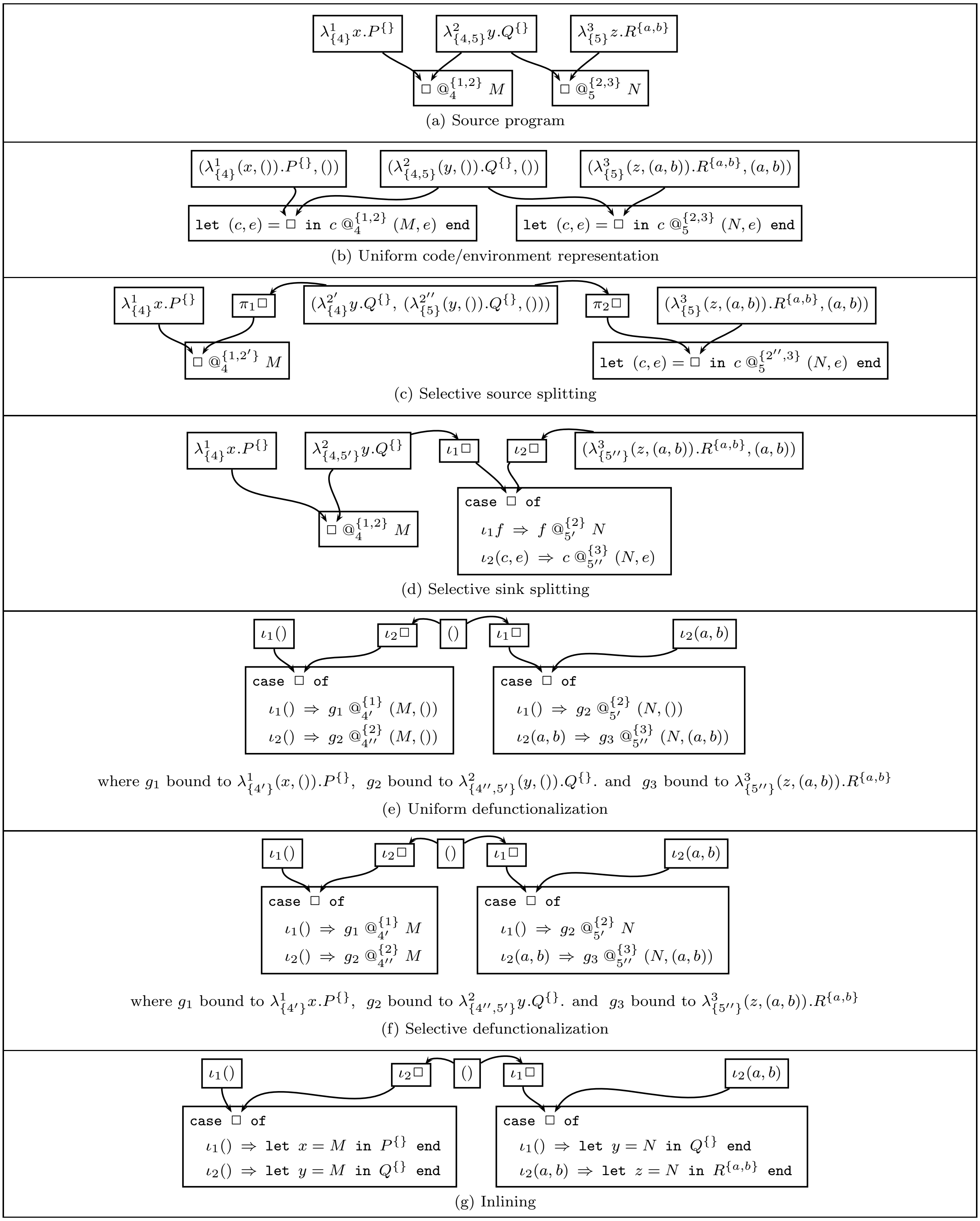


Figure 1: Transformations on a W-shaped flow diagram.

effect of splitting the source point $\lambda^2_{\{4,5\}}$ into two representations: (1) a code pointer that flows to call site 4, which is shared with another closed function; and (2) a closure that flows to call site 5, which is shared with an open function. This permits call site 4 to use an optimized calling sequence even though call site 5 uses the inefficient one. The costs of this customization are pairing the two representations and extracting them (via explicit projections π_i) from the pair.

An alternative approach for breaking pollution constraints is to split a sink point, as illustrated in Figure 1(d). Sink splitting involves injecting (via ι_i) inconsistent representations into a tagged variant value, and then dispatching off the tag (via `case`) to an appropriate handler at a sink site. Again, breaking the pollution constraints allows the two closed functions to share a call site with an optimized calling convention. The costs of this approach are the injections and case analyses necessary at the other call site.

Source and sink splitting enable customizations by eliminating some pollution, but also introduce manipulations of tuples or variants that were not in the original program. In the balance, are such customizations a good idea? This question is a complex one that is addressed in Sections 4 and 5, which investigate the conditions under which the customizations can improve the performance of a program.

It is important to note that closure conversion is not the only uniform strategy for transforming higher-order functions to first-order functions. Another uniform strategy is *defunctionalization*, in which every function value is represented as an element of an algebraic datatype whose constructor uniquely identifies the abstraction of the function and whose components are the values of the free variables of the abstraction [32, 5]. This is similar to a closure, except that the environment is paired not with a code pointer but with an abstract tag denoting the function. Call sites are transformed to dispatch off this tag to a direct call of an appropriate global closed function. In the simplest approach, there is one algebraic datatype for all functions of a given monomorphic type, and one constructor for each abstraction appearing in the program with this type.[7] However, flow analysis can significantly reduce the size of the datatypes introduced by defunctionalization [39, 8].

Uniform defunctionalization is illustrated for our example in Figure 1(e). The code portions of the functions have been lifted to global functions g_1, g_2, and g_3, leaving behind only environment injections. Call sites dispatch off the injection tags to invoke an appropriate global function. Uniform defunctionalization can be viewed as maximally splitting each sink according to all the functions that might flow there. This maximal splitting effectively eliminates *all* representation pollution, and allows representations to be chosen completely independently for each abstraction. Figure 1(f) shows how selective representations can be used in conjunction with defunctionalization to improve the representations of g_1 and g_2 and the three sites at which these are invoked.

Inlining can be viewed as a variant of defunctionalization in which the known call to a global function within a branch of a case dispatch has been reduced at compile time (see Figure 1(g)). This perspective covers the flow-directed inlining of open functions [45], as well as the inlining of multiple functions at a single call site.

[7] Polymorphic functions are handled by a *monomorphization* process that specializes a polymorphic definition to each monomorphic type at which it is used.

3. CIL COMPILER OVERVIEW

3.1 Compiler Architecture

We have constructed a whole-program compiler for core SML based on CIL, our typed intermediate language. The key features that distinguish CIL from other such languages are its use of flow labels in conjunction with intersection and union types to encode polyvariant flow analyses in the type system of the language [44]. The resulting *flow types* support the customizations presented in this report and serve as an important sanity check in the compiler implementation. In CIL, customization opportunities are represented by *virtual records* (introduction forms for terms of intersection type) and *virtual case expressions* (elimination forms for terms of union type). The compiler enables customization by *reifying* some of the virtual forms into real forms, as seen in the above examples of source and sink splitting.

The core compiler implementation is based on the architecture specified in [14]. We have extended the simple intermediate language described there with numerous features (e.g., references, arrays, exceptions, standard library functions, etc.) necessary to support the compilation of SML. In implementing the compiler, we took advantage of existing tools and other freely available SML compilers. The CIL compiler uses the MLton source-to-source defunctorizer [8] as a prepass to convert SML into Core SML. It then uses the front end of the SML/NJ 110.03 compiler (somewhat modified) to produce FLINT code. The FLINT code is translated to untyped CIL code, keeping datatype information on the side to avoid reinference of recursive types. The untyped CIL code is then processed by one of four flow analyses we have implemented as part of the type inference/flow analysis stage that transforms it into flow-typed CIL code.

The representation transformations detailed in [14] are parameterized over seven *function representation strategies* that heuristically choose the representation for each abstraction and call site. The CIL compiler uses flow types to manage the "plumbing" of the chosen representations, splitting sources and sinks as needed to ensure that all representations are used consistently relative to type and flow information. Despite complex types powerful enough to encode polyvariant flow analyses and term representations that duplicate the components of virtual records and the branches of virtual case expressions, the size of the intermediate representations of CIL programs is tractable in practice [16].

The CIL compiler back-end transforms typed CIL programs into assembly code for the SPARC processor. It does not currently add any type annotations, or assertions, to the assembly code, although this is planned for future work. The produced assembly code is linked with a runtime library providing the environment in which CIL programs are executed. The back-end is based on MLRISC, a framework for building portable optimizing code generators [18].

The runtime library is written in C and provides memory management, exception handling, basis functions and a foreign function interface for CIL programs at runtime. The runtime library currently manages memory using the Boehm-Demers-Weiser conservative garbage collector for C [7]. CIL programs use stack-allocated activation records, which have a layout similar to C stack frames. The code generator does not yet implement tail recursion.

CIL data representations are straightforward. Records, arrays, references, and strings are heap-allocated and include size headers[8]. Exception identifiers and all other constants are immediate. Injections may either be immediate or heap allocated, depending on the number and types of summands in their type.

3.2 Function Representation Strategies

The CIL compiler currently implements the following seven function representation strategies:

- *uniform (uni)*: represents all functions as closures. We investigate two uniform closure representations: (1) a *paired closure* that is a record of a code pointer and environment, which is itself a record of free variable values; and (2) a *flat closure* (see [2]) that is a single record containing both code pointer and free variable values.

- *polluted selective (pse)*: represents a closed abstraction as a code pointer if and only if all the colleagues of the abstraction are closed. This implements the approach proposed by Wand and Steckler in [41].

- *selective source splitting (src)*: generates a code-only representation for a closed function flowing to call sites that are not shared with open functions. If a closed function shares some application sites with other closed functions but shares other application sites with open functions, then the framework will "split the source" by generating a record containing multiple copies of the function.

- *selective sink splitting (snk)*: generates a selective representation when the function has no free variables. This representation is called "sink splitting" because if the function shares call sites with open functions, the transformation framework will inject the function representations into a sum type and the application site will be split into multiple sites governed by a case dispatch.

- *defunctionalization (dfn)*: represents all functions as injected environments and all call sites as dispatches to an invocation of an appropriate global function on the argument and the environment. Functions not sharing call sites with other functions are not injected.

- *selective defunctionalization (sdf)*: like *dfn*, but represents closed functions as code pointers to functions that expect just an argument (but no environment).

- *inlining (inl)*: inlines (possibly open) functions at the call site. The run-time representation of an inlined function is a record of the function's free variable values. Call sites may be split to allow non-inlined and (possibly multiple) inlined representations at the same call site. Thus far, we have investigated only one point in a huge space of possible inlining heuristics: any non-recursive function flowing to two or fewer call sites will be inlined; when inlining is not possible, the strategy falls back to *pse*.

The results of using these strategies to transform the example in Figure 1(a) are shown in Figures 1(b)-(g). Note that the *pse* strategy does not have its own figure because the result is the same as that for *uni* due to pollution.

[8]Such headers are currently unnecessary since we use conservative GC. But we expect to develop customized memory management in the future.

3.3 Optimizations

The CIL compiler is currently a research tool and is far from an industrial-strength program. In particular, beyond the function customizations described in this report, some standard "partial-evaluation style" optimizations, and some back-end optimizations performed by MLRISC, there are few optimizations performed in CIL.

One important optimization we do implement is *known function variable elimination (KVFE).* If at representation choice time it is guaranteed that a function f will not have a closure, then we do not consider a reference to f to be free in g for any other function g in the program. Since invocations of known functions can be compiled as a jump to an address known at compile-time, the name of a known function appearing in the rator position of an application need not be treated as a free variable. This optimization significantly increases the number of closed functions in our compiler, which in turn creates more customization opportunities. In early versions of our compiler, the free variables of a function included every externally defined function name used within the body of the function. This hobbled customization because very few functions were closed.

We use a slightly different version of KFVE than other compilers. First, we use a flow-based calculation for KFVE rather than the standard syntax-based algorithm, so we need not worry about tracking names through copy propagation or about functions escaping their scope of definition. In this respect our algorithm is similar to one hinted at in [3]. Second, we do not currently use KFVE to eliminate closures where the function pointer is known and the environment is empty. We have postponed this work until we have developed a more general flow-based *known value elimination* algorithm that handle constants, unused values, and records of known values in a single framework.

The KFVE algorithm enables the compiler to find more empty environments and to decrease the environment size in many cases. For *uni*, KFVE has no effect on the cost, since all variables are bound to closures and none is bound to a known function. For *dfn*, KFVE has been disabled to maintain its status as a "straw man" strategy comparable to *uni*. In other strategies, KFVE results in a roughly 13% decrease in cost for paired closures under our cost model (see the next section), depending on both the strategy and the benchmark.

4. COST MODEL

An important goal of this paper is to evaluate and compare the seven function representation strategies presented in Section 3.2 using benchmark programs. Although one way to do this is to measure running times, our compiler does not yet implement numerous optimizations (see Section 3.3) so such measurements would not represent the possible costs and benefits of the various representations. Furthermore, we want some way to dissect the costs of function creation, function application, and pollution removal in order to better understand where the time charged to function representations is being spent.

Based on these considerations, our approach in this paper is to instrument the CIL compiler to track intermediate code points related to function representation, count the dynamic number of times these points are reached when executing the compiled code, and use a *cost model* to attribute a cost to these points. Our approach models only those costs directly related to packaging, unpackaging, and otherwise manipulating the code pointer, argument, and free variables of a function when creating or invoking it. It does not model costs of manipulating values of non-function type, nor does it account for other operations associated with functions, such as preserving live registers on the stack across a function call. Moreover, our model does not reflect many optimizations commonly associated with functions, such as storing some free variables on the stack or stack-allocating some closure and environment records, nor does it reflect many other compiler optimizations (e.g., loop optimizations, tuple flattening, cheap representations of certain variant values) that could affect the costs we charge to function representations.

In light of these disclaimers, we stress that there may be little correlation between the relative costs of function representation strategies under our cost model and those obtained in optimizing compilers. Nevertheless, we believe that our model gives some sense for the benefits and drawbacks of various function representation strategies and suggests that certain strategies are worthy of further exploration.

This section develops the cost model for comparing function representation choices made by the CIL compiler. The model abstracts low-level implementation details, and focuses on cost as the number of abstract assembly language instructions of three classes: register-to-register (r), register-to-memory (m), and memory allocation (a). This metric captures the relative cost of customized representations, and can be applied to a particular implementation to account for architectural details such as cache and branch prediction (see section 4.4 below).

Our model handles both paired and flat closures. For paired closures we assume that the code references a single argument register holding a pointer to a tuple of the function argument and the environment. For flat closures we assume that the code references two argument registers holding (1) the function argument and (2) the closure record itself.[9]

4.1 Function Definition

Record Creation Environments and closures are represented as heap-allocated records. The cost of creating a record includes an allocation cost a plus a store cost m for each field of the record. We assume that a record includes a one-word header (to facilitate garbage collection); it costs $r+m$ to store the header. A zero-field record can be represented specially as a null pointer.[10] The total cost $RC(n)$ of creating an n-field record is:

$$\begin{aligned} RC(0) &= r \\ RC(n) &= r + (n+1) * m + a, \text{ if } n > 0 \end{aligned}$$

Closure Creation Paired closures require both a pair record and an environment record. Flat closures require only a single record. The cost of creating a closure with n free variable values is:

$$\begin{aligned} CC_{paired}(n) &= RC(2) + RC(n) \\ CC_{flat}(n) &= RC(n+1) \end{aligned}$$

4.2 Function Invocation

Known Function Call Calling a function known at compile time jumps directly to the function address with a cost of $KF = 3r$[11].

Unknown Function Call Calling a function unknown at compile time loads the function address into a register and jumps indirectly with a cost of $UF = 4r$.

Record Projection Extracting closure and environment components requires accessing a slot of a heap-allocated record with cost $RP = m$.

Closure Application The cost of applying a closure depends on the paired vs. flat representation, the number of free variable values, and whether or not the function is known. Paired closures with unknown functions require projecting both the code pointer and environment and invoking an unknown function; for known functions, the code pointer projection is avoided. In either case, an argument/environment record is constructed and deconstructed, and the n environment components are projected. The cost of applying a paired closure with n free variables is:

$$\begin{aligned} CA_{paired/known}(n) &= RC(2) + KF + (3+n) * RP \\ CA_{paired/unknown}(n) &= RC(2) + UF + (4+n) * RP \end{aligned}$$

For flat closures with unknown functions, the code pointer is projected before the call and free variables are projected after the call; the code pointer projection is avoided for a known function. By assumption, there are no allocation or projection costs for passing the argument and closure:

$$\begin{aligned} CA_{flat/known}(n) &= KF + n * RP \\ CA_{flat/unknown}(n) &= UF + (n+1) * RP \end{aligned}$$

4.3 Pollution Removal

Virtual Tuples Virtual tuples that are reified into real tuples incur the cost of record creation (RC) and record projection (RP). We see an example of pollution removal cost due to virtual tuples in Figure 1(c) (selective source splitting). Here the $\lambda^2_{\{4,5\}}$ abstraction has been split into two different representations, incurring the cost of a record creation at definition and record projection at invocation.[12]

Virtual Variants Virtual variants which are reified into real variants incur the cost of injection (IN) and case dispatch (CD). An example of pollution removal cost due to virtual variants is Figure 1(d) (selective sink splitting).

Injections are allocated on the heap, and contain the injection tag and the injected value. Assuming headers on records as above, then injections have the same cost as 1-field records (where the tag is the header and the field is the value): $IN = RC(1)$.[13]

[9] Whether the function argument and record of free variable values are passed in separate registers or as a tuple in one register is orthogonal to the paired vs. flat distinction. To avoid considering four combinations, we associate the more efficient two-register strategy with the more efficient flat representation to yield the two "extreme" combinations.

[10] A one-field record can sometimes be represented as the field itself. CIL's *parallel contexts* [44] prevent performing this optimization in all cases. Our cost model (but not our compiler) implementation assumes this optimization for one-element environments (where it is always safe).

[11] The actual cost of known and unknown function calls is implementation-dependent. For example, on the UltraSPARC a branch misprediction can cost as many as 18 cycles. But since KF and UF are taken to be constant in the model, they can be defined according to the implementation.

[12] It is possible to lift abstractions (but not necessarily closures) out of tuples to top-level and remove the corresponding projections. The CIL compiler does not yet do this, nor does the cost model implementation, except in the defunctionalization strategies.

[13] In some cases the injection may be represented only as an

The cost of a case dispatch with n clauses ($CD(n)$) includes m to load the discriminant tag, $r + m$ to load and bind the discriminant value, r to branch after a clause, and the cost of dispatching using the better of a conditonal tree or jump table (depending on the number of clauses; see [40]):

$$CD(n) = 2r + 2m + \texttt{cases}(n), \text{ where}$$
$$\texttt{cases}(0|1) = 0r;\quad \texttt{cases}(2) = 3r;\quad \texttt{cases}(3) = 4r;$$
$$\texttt{cases}(4) = 5r;\quad \texttt{cases}(5) = 7r;\quad \texttt{cases}(\_) = 4r + 1m.$$

4.4 Example

Consider the representations of the example program in Figure 1. Assuming paired closures, we can use the cost model to calculate the static costs of different representations along particular paths in the program. For example, along the path from $\lambda^1_{\{4\}}$ to $@^{\{1,2\}}_4$ in Figure 1(d), we have no function definition cost, a function invocation cost of UF, and no pollution removal cost. This results in a cost of $4r$. Along the same path in Figure 1(f), we have no definition cost, an invocation cost of KF, and a pollution removal cost of $IN + CD(2)$[14], for a total cost of $9r + 4m + a$.

Using this cost model and reasonable weights w_r, w_m, and w_a for the r, m and a operations for a particular implementation, we can calculate the actual cost of different representations. For example, on a typical SPARC implementation [35] with a fast memory allocator we might choose: $w_r = 1$, $w_m = 3$ and $w_a = 16$ to represent machine cycles. In the example, we find that the selective sink splitting representation costs 4 cycles using these weights, and the selective defunctionalization representation costs 37 cycles.

The costs for each of the four paths in the program, and for each of the representations in Figure 1 are given in the following table, using the weights given above.

Representation	(1,4)	(2,4)	(2,5)	(3,5)
uni	88	88	88	100
src	7	33	117	103
snk	4	4	37	130
dfn	69	69	69	101
sdf	37	37	37	69
inl	34	34	34	66

Cache effects may be accounted for by adding an expected miss cost to the base cost for a memory operation. The expected miss cost is the product of the miss rate and miss penalty for the implementation [29]. For example, a typical UltraSPARC implementation has a first level cache miss penalty of 6 cycles and a second level cache miss penalty of 13 cycles [37]. So in the example above, selective sink splitting has no data miss cost (having no memory operations), and selective defunctionalization has a total cost of $9r+4m+a+((4/14)*\texttt{l1missrate}*6)+((4/14)*\texttt{l1missrate}*\texttt{l2missrate}*13)$.

The results reported in the next section are based on the simpler instantiation of the cost model, without modeling cache misses or branch mispredictions.

-*-LaTeX-*-

immediate tag, and not allocated on the heap. The CIL compiler supports this, but we do not account for this optimization in the cost model.

[14]An optimization in defunctionalization with uniform representation at an invocation site is to use the code pointers themselves as the tag, avoiding the injection altogether. The CIL compiler does not currently use this optimization.

5. MEASUREMENTS

To determine the effect of customizations and pollution removal on the dynamic costs of function representations, we use the cost model to measure the performance of our function representation strategies for a set of kernel codes and small benchmarks of 50 to 3000 lines of source code. The SML benchmarks that we use in this paper are:

Name	Source lines of code	From	dynamic function creations	dynamic function applications
msort	55	TIL	1.8m	13m
church	73	Church	2.7k	188m
soli	115	O'Caml	1.7m	3.1m
quicksort	120	O'Caml	1.6m	4.7m
life	147	SML/NJ	643k	6.5m
matmult	156	TIL	40k	40.m
fft	194	O'Caml	1.4m	2.8m
tsp	249	SML/NJ	4.9m	10m
barnes-hut	401	SML/NJ	502k	9.2m
kb	467	O'Caml	7.3m	13m
frank	487	TIL	36m	60m
ratio-regions	505	SML/NJ	36m	162m
tyan	856*	TIL	29k	594k
boyer2	856	Church	551k	1.2m
lexgen	1067	SML/NJ	1.5m	5.2m
simple	1228*	SML/NJ	2.2m	22m
pia	2081*	TIL	2.5m	5.1m
nucleic	2923	O'Caml	1.1m	2.5m

The source lines count does not include comments or blank lines. Source lines marked with an asterisk are the output of the MLton defunctorizer, which tends to insert more line breaks than a human programmer. The "dynamic function creations" column shows the number of closure creations when the program is run after being compiled with the *uni* strategy. The "dynamic function applications" column shows the number of function applications when the program is run after being compiled with the *uni* strategy. The letter "k" means 10^3 and "m" means 10^6.

Most of the benchmarks are from standard benchmark suites. Some occur in slight variations in more than one suite. We have contributed two new benchmarks showing extremes of programming style. The **boyer2** benchmark is a variant of the O'Caml **boyer** benchmark modified to be entirely first-order; **church** is a library of arithmetic on Church numerals, a simple example of a program built out of many very small higher-order functions.

Table 1 and the bar charts in Figure 2 summarize the results of running the benchmarks using the seven representation strategies presented in Section 3.2. Table 1 lists the average costs of each strategy in our cost model relative to that for *uni*, which is given a cost of 100. Because the **church** benchmark is such an extreme case, averages for the benchmarks without **church** are also presented. Figure 2 presents the costs of each strategy for each benchmark for the two closure representations, and displays the cost as a bar broken into three segments:

1. the segment filled with represents the (relative) cost of function creation;
2. the segment filled with represents the (relative) cost of calling a function, including the cost of unpacking the environment if an environment is used;

closures	uni	pse	src	snk	inl	dfn	sdf
paired	100	83	83	90	73	111	86
w/o church	100	76	76	76	62	87	66
flat	100	88	88	157	89	180	171
w/o church	100	86	86	87	88	100	89

Table 1: Average relative costs of function representation strategies as % of cost of *uni* strategy.

3. the segment filled with ■ represents the (relative) cost of pollution removal, accounting for the new record creations/projections and variant creations/case dispatches introduced by splitting.

Based on data not shown here, the *uni* strategy using flat closures on average incurs only 68% of the cost of the *uni* strategy using paired closures. The actual costs of pollution removal are independent of the closure representation. As seen in Figure 2, the lower total costs when using flat closures makes the costs of pollution removal proportionally larger than when using paired closures.

There is a wide variation in the relative cost of using any of the selective representations vs. the uniform representation: from 9% for **msort** to > 99% for **church**, **matmult**, and **quicksort**. In the cases of **matmult** and **quicksort**, most of the functions are open, so we always have to accept the cost of using environments. In the case of **church**, a substantial amount of representation pollution precludes using optimized representations in the most frequently called functions. In both **church** and **matmult**, the cost of building closures is not visible in the bar charts due to the high ratio of function calls to closure creations.

Because it never pays any plumbing costs but in some cases still enjoys the benefits of more efficient closed function representations, *pse* is never worse than, and is sometimes significantly better than, *uni*. On average, the cost of using *pse* is 83% (for paired closures) and 88% (for flat closures) of the cost of using *uni*. There does not seem to be any strong correlation between the size of the benchmark and the cost savings, although there is an obvious correlation between small size and large variation. The actual speedup in a program would be less, since we are only modeling costs of function closing and invocation. But it seems that the advantage of specialized function representations is clear, and that even a simple representation customization such as selective closure conversion is worthwhile.

Our experimental results show that for the simple function representations and cost models we consider, several pollution breaking strategies are rarely more effective than *pse* and can be far worse due to the cost of breaking pollution. What follows is an explanation of these results.

For certain programs in our benchmark set, our flow analysis finds that no call site has more than a single function flowing to it. In this case there is no pollution to remove. There is also no overhead involved in using a defunctionalization strategy. Benchmarks **matmult**, **fft**, **tsp**, and **boyer2** fall into this category.

There are some benchmarks in which some call sites have more than a single function flowing to them – so there is some overhead involved in a defunctionalization strategy – but all functions flowing to a given call site have the same representation. In this case there is no pollution to remove in *src* and *snk*. Benchmarks in this category are **soli**, **quicksort**, **frank**, **ratio-regions**, and **lexgen**.

Paired closures:	Excluding pollution removal costs		Including pollution removal costs	
benchmark	**src**	**snk**	**src**	**snk**
msort	0.00	1.37	0.00	-8.20
church	0.00	46.66	0.00	-26.67
life	0.68	1.15	0.65	0.28
barnes-hut	0.00	0.00	0.00	0.02
kb	0.10	0.11	0.08	0.11
tyan	0.37	0.84	0.33	-3.09
simple	4.99	5.33	4.55	3.40
pia	0.60	0.60	0.60	0.60
nucleic	0.00	3.11	0.00	0.57

Table 2: Improvements and costs of pollution removal as % of cost of the *pse* strategy.

The *src* strategy will clone a copy of a closed function if it shares one call site with an open function and another call site with only closed functions. If every closed function has an open function as an immediate colleague at every shared application site, then *src* is identical to *pse*. Benchmarks in this category are **msort**, **barnes-hut**, and **nucleic**.

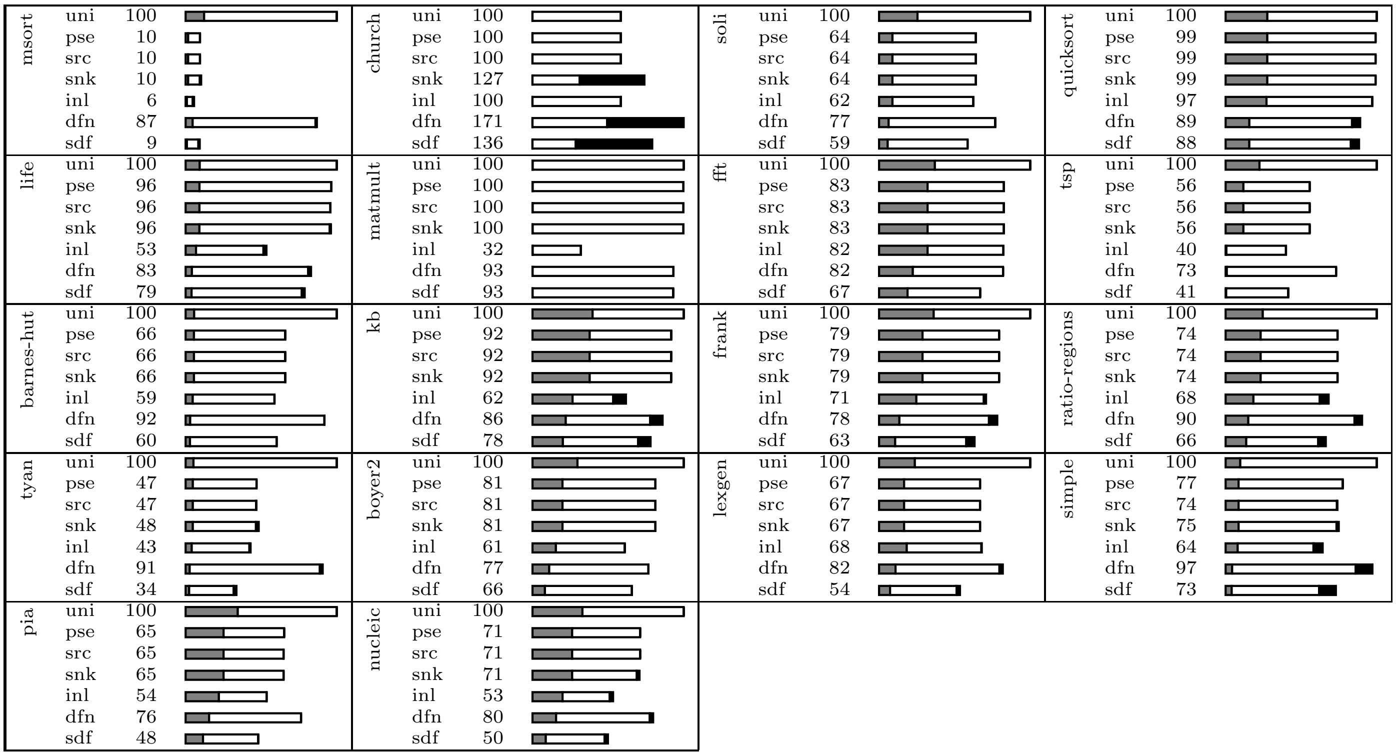
This leaves **church**, **life**, **kb**, **tyan**, **simple**, and **pia** to explore as to whether *src* gives improved performance over *pse*. To evaluate *snk*, we need to additionally investigate **msort**, **barnes-hut**, and **nucleic**.

Table 2 summarizes the improvements due to pollution removal for *src* and *snk*, using paired closures, as well as the total cost of pollution removal. The first two columns of numbers show the percentage improvement in the cost over *pse* if the cost of pollution removal itself is excluded. The last two columns show the percentage improvement when the pollution removal costs are included.

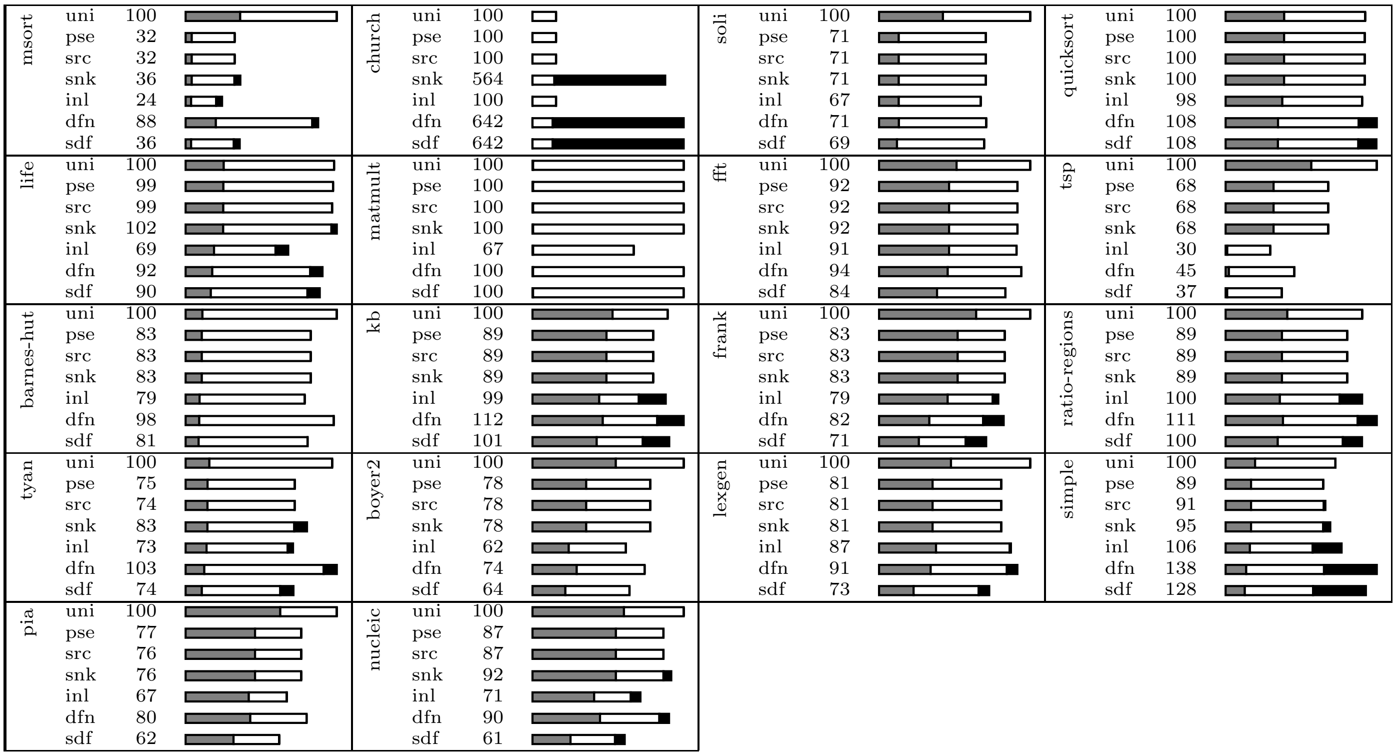
The first two data columns show that if pollution removal were "free", it could reduce (in some cases, significantly) the cost of function creation and application. However, when the cost of creating and manipulating the values required by pollution removal is included, these benefits vanish in almost all cases. Indeed, the cost of pollution removal typically makes the *snk* strategy *less* effective than *pse*. This result is clearest with **church**, where a lot of representation specialization is blocked by pollution but the cost of removing pollution turns out to be prohibitive. There is one exception: **simple** shows non-trivial gains for paired closures, even when the costs of pollution removal are factored in.

While the conservative partial pollution removal by *src* never loses for paired closures on our benchmark set, it seldom improves performance by much, and the very aggressive *snk* may actually significantly degrade performance. For flat closures, the data in Figure 2(b) shows that the larger relative cost of pollution removal can cause the *src* strategy to have higher cost than the *pse* strategy in some cases, even in **simple**. Clearly the space of heuristics for pollution removal requires further exploration. There are some record creations and projections that the CIL compiler creates for pollution removal that can be eliminated. However, much of the cost of pollution removal for *snk* is due to injections and case dispatches, which are not obviously removable.

Our inlining strategy generally results in lower costs than *pse* for both paired and flat closures. The fact that *inl* exhibits pollution removal costs in some benchmarks indicates that sinks are being split to allow inlined functions to share call sites with other functions representations. When using flat closures, the cost of pollution removal sometimes out-



(a) Benchmark costs using paired closures.



(b) Benchmark costs using flat closures.

Figure 2: Relative costs of function representation strategies. In each benchmark, the number is the total cost as % of *uni* cost, and bar lengths are normalized to the length of the longest of the seven bars.

weighs the benefits of inlining (**kb**,**ratio-regions**,**simple**). The **lexgen** benchmark shows a case where *inl* appears more costly than *pse* even before pollution removal; this is a result of variations of the KFVE algorithm for various strategies. For the *pse* strategy, which introduces no splitting, and for *sdf*, which introduces maximal splitting, the KFVE algorithm is more aggressive than for the *src*, *snk*, and *inl* strategies, which may or may not introduce splitting at a given function definition or invocation.

The defunctionalization strategies *dfn* and *sdf* split all call sites, so it is easily possible to have a specialized representation per function definition. The *dfn* strategy realizes two benefits from splitting: (1) record manipulations involving code pointers are eliminated; and (2) all function calls become known calls. It does not distinguish between closed and open functions, and passes empty environments at call sites rather than discarding them. In many cases, *dfn* can beat *uni* because its benefits outweigh the costs of the injections and case dispatches due to splitting. Occasionally, *dfn* even beats *pse*, as in **tsp**, **frank**, and **boyer2**.

Unlike *dfn*, the *sdf* strategy handles closed functions specially. Our data show that this leads to significant gains over *dfn* and, in many cases, over *pse*. The costs of *sdf* are between 70% and 93% of the *pse* strategy in 16 of the 18 benchmarks using paired closures and between 54% and 91% in 8 of the 18 benchmarks using flat closures. If we do not consider the outlier **church** benchmark, then the *sdf* strategy out-performs *pse* on the average for paired closures, and is close to *pse*'s performance for flat closures.

We expect the combination of inlining and selective defunctionalization to prove a very cost-effective strategy for most of the benchmarks, except for cases where the high cost of pollution removal would make **pse** the most effective strategy (such as (1) the **church** benchmark for both closure representations and (2) the **simple**, and **quicksort** benchmarks with flat closures). However, we have not actually collected numbers for this combination.

We emphasize that our results include only the costs of function representations in our cost model, and that our conclusions are dependent on the architecture being modeled, on the presence or absence of compiler optimizations, and on the closure representations used by the compiler.

6. CONCLUSIONS AND FUTURE WORK

Our results suggest that flow-based customization of functions based on the presence or absence of free variables (our *polluted selective* strategy) is worthwhile. It remains to be seen how effective this customization is in a production compiler, and whether it (and other flow-based customizations) can be adapted to frameworks for separate compilation.

Our experiments with removing function representation pollution are less conclusive. In three strategies (*selective source splitting*, *selective sink splitting*, and *uniform defunctionalization*), there is little or no function representation pollution to remove in many benchmarks, and for benchmarks with pollution, the costs of removing the pollution often outweigh the benefits. In the *selective defunctionalization* strategy, pollution removal leads to significant gains for many benchmarks, but there are still cases where it does not pay off. This suggests that it would be worthwhile to characterize situations where pollution removal is beneficial and use it only in these situations. Note that we have not yet explored optimization opportunities enabled by our customizations that might give rise to additional benefits not reflected in our current data; these might affect our conclusions regarding pollution removal.

Our inlining strategy is very effective, but it is unclear how much this depends on the flow-based nature of the inlining and the fact that pollution removal allows multiple functions to be inlined at the same call site. In a future study, we plan to compare various heuristics for inlining in our framework (varying fan-in, fan-out, and fall-back strategy) with each other and with classical syntax-based inlining techniques (e.g., [2, 38]) to gain insight into the factors that make this strategy effective.

While pollution removal does not seem very helpful in the context of selecting closed vs. open functions, it may very well be effective for other representation decisions. In terms of function representations, we are currently investigating: (1) uncurrying [19], which can increase the number of closed functions; (2) closure representations that exclude free variables from an environment if their values are available at all call sites [36]; and (3) register allocation and calling conventions informed by flow information. There are numerous other closure representation tricks (e.g., those discussed in [24, 2, 34]) to investigate in the context of our framework. We have yet to explore customized representations for other structures, but CIL is rich enough to support flow-directed customizations for all types of data. For instance, we believe it can be used to treat certain data structures as *fictitious*, as in [34], and can be extended to encode polyvariant usage-based customizations, such as Haskell's used-once thunk optimization [42]. Much work remains to be done to optimize customized representations and develop heuristics for choosing between allowable representations.

There are many areas for improvement in the CIL compiler as a whole. The compiler can benefit from numerous standard optimizations not yet implemented (e.g., tuple flattening, loop optimizations, passing arguments in registers) as well as some important non-standard optimizations (e.g., the complete removal of polymorphic equality, a type system to formalize the code-pointer-as-tag optimization, generalized known value elimination). Several existing algorithms can be more efficiently implemented: e.g., our algorithm to produce clones and multiway dispatches for pollution removal currently fails to re-combine identical clones. There are also many opportunities for improvement in the representation of the intermediate language: e.g., although we can model the effect of multi-argument functions, we have yet to implement them in the compiler.

The CIL compiler currently maintains type and flow information through code generation, but its output is untyped. We plan to eventually produce typed assembly language (TAL) [27] from the CIL compiler.

Acknowledgments

We thank other Church Project members for their advice and support. We especially acknowledge Jef Considine's implementation of cyclic hash-consing and the contributions of Santiago Pericas-Geersten and Glenn Holloway to early versions of the CIL compiler. We thank the referees for their excellent questions and suggestions; we have incorporated as many as space permits. Thanks to Nausheen Eusuf for helping us to achieve closure on our title.

7. REFERENCES

[1] O. Agesen. The Cartesian product algorithm. In *Proceedings of ECOOP'95, Seventh European Conference on Object-Oriented Programming*, vol. 952, pp. 2–26. Springer-Verlag, 1995.

[2] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[3] J. M. Ashley. *Flexible And Practical Flow Analysis for Higher-Order Programming Languages*. Ph.D. thesis, Indiana University, May 1996.

[4] A. Banerjee. A modular, polyvariant, and type-based closure analysis. In ICFP '97 [20].

[5] J. M. Bell, F. Bellegarde, and J. Hook. Type-driven defunctionalization. In ICFP '97 [20], pp. 25–37.

[6] G. E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, Apr. 1993.

[7] H.-J. Boehm. Space efficient conservative garbage collection. In *Proc. ACM SIGPLAN '93 Conf. Prog. Lang. Design & Impl.*, pp. 197–206, 1993.

[8] H. Cejtin, S. Jagannathan, and S. Weeks. Flow-directed closure conversion for typed languages. In *Programming Languages & Systems, 9th European Symp. Programming*, vol. 1782 of *LNCS*, pp. 56–71. Springer-Verlag, 2000.

[9] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In *Proc. ACM SIGPLAN '89 Conf. Prog. Lang. Design & Impl.*, pp. 146–160, 1989.

[10] C. Chambers and D. Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. In *Proc. ACM SIGPLAN '90 Conf. Prog. Lang. Design & Impl.*, 1989.

[11] K. D. Cooper, M. W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2):105–118, 1993.

[12] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type erasure semantics. In *Proc. 1998 Int'l Conf. Functional Programming*, pp. 301–312. ACM Press, 1998.

[13] J. Dean, C. Chambers, and D. Grove. Selective specialization for object-oriented languages. In *Proc. ACM SIGPLAN '95 Conf. Prog. Lang. Design & Impl.*, 1995.

[14] A. Dimock, R. Muller, F. Turbak, and J. B. Wells. Strongly typed flow-directed representation transformations. In ICFP '97 [20], pp. 11–24.

[15] A. Dimock, I. Westmacott, R. Muller, F. Turbak, J. B. Wells, and J. Considine. Space issues in compiling with intersection and union types. In *Preliminary Proceedings of the Third Workshop on Types in Compilation (TIC 2000)*, 2000. Superseded by [16].

[16] A. Dimock, I. Westmacott, R. Muller, F. Turbak, J. B. Wells, and J. Considine. Program representation size in an intermediate language with intersection and union types. In *Proceedings of the Third Workshop on Types in Compilation (TIC 2000)*, vol. 2071 of *LNCS*, pp. 27–52. Springer-Verlag, 2001. Supersedes [15].

[17] K.-F. Faxén. The costs and benefits of cloning in a lazy functional language. In *Trends in Functional Programming, Volume 2*. Intellect, 2001.

[18] L. George. MLRISC: Customizable and reusable code generators. Technical report, Bell Labs, 1997.

[19] J. Hannan and P. Hicks. Higher-order uncurrying. In *Conf. Rec. POPL '98: 25th ACM Symp. Princ. of Prog. Langs.*, pp. 1–11, 1998.

[20] *Proc. 1997 Int'l Conf. Functional Programming*. ACM Press, 1997.

[21] S. Jagannathan and S. Weeks. A unified treatment of flow analysis in higher-order languages. In *Conf. Rec. 22nd Ann. ACM Symp. Princ. of Prog. Langs.*, pp. 393–407, 1995.

[22] S. Jagannathan, S. Weeks, and A. Wright. Type-directed flow analysis for typed intermediate languages. In *Proc. 4th Int'l Static Analysis Symp.*, vol. 1302 of *LNCS*. Springer-Verlag, 1997.

[23] M. P. Jones. Dictionary-free overloading by partial evaluation. In *PEPM '94 — ACM SIGPLAN Workshop Partial Eval. & Semantics-Based Prog. Manipulation*. ACM, 1994.

[24] D. Kranz, R. Kelsey, J. A. Rees, P. Hudak, J. Philbin, and N. I. Adams. Orbit: An optimizing compiler for Scheme. In *Proc. SIGPLAN '86 Symp. Compiler Construction*, pp. 219–233, 1986.

[25] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Conf. Rec. POPL '96: 23rd ACM Symp. Princ. of Prog. Langs.*, 1996.

[26] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Trans. on Prog. Langs. & Systs.*, 21(3):528–569, may 1999.

[27] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Trans. on Prog. Langs. & Systs.*, 21(3):528–569, May 1999.

[28] F. Nielson and H. R. Nielson. Infinitary control flow analysis: A collecting semantics for closure analysis. In *Conf. Rec. POPL '97: 24th ACM Symp. Princ. of Prog. Langs.*, pp. 332–345, 1997.

[29] D. A. Patterson and J. L. Hennessy. *Computer Organization & Design*. Morgan Kaufmann, 1998.

[30] J. Plevyak. *Optimization of Object-Oriented and Concurrent Programs*. Ph.D. thesis, University of Illinois at Urbana-Champaign, 1996.

[31] J. Plevyak and A. Chien. Type directed cloning for object-oriented programs. In *Workshop for Languages and Compilers for Parallel Computers*, Aug. 1995.

[32] J. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM Annual Conference*, pp. 717–740, 1972.

[33] O. Shivers. *Control Flow Analysis of Higher Order Languages*. Ph.D. thesis, Carnegie Mellon University, 1991.

[34] J. M. Siskind. Flow-directed lightweight closure conversion. Technical Report 99-190R, NEC Research Institute, Inc., Dec. 1999.

[35] SPARC International Inc., Menlo Park, CA. *The SPARC Architecture Manual, Version 8*.

[36] P. Steckler and M. Wand. Lightweight closure conversion. *ACM Trans. on Prog. Langs. & Systs.*, 19(1):48–86, Jan. 1997.

[37] Sun Microsystems, Inc., Palo Alto, CA. *UltraSPARC User's Manual*.

[38] D. Tarditi. *Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML*. Ph.D. thesis, Carnegie Mellon University, Dec. 1996.

[39] A. P. Tolmach and D. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *J. Funct. Programming*, 8(4):367–412, 1998.

[40] G.-R. Uh and D. B. Whalley. Effectively exploiting indirect jumps. *Software Practice and Experience*, 29(12):1061–1101, 1999.

[41] M. Wand and P. Steckler. Selective and lightweight closure conversion. In *Conf. Rec. 21st Ann. ACM Symp. Princ. of Prog. Langs.*, pp. 435–445, 1994.

[42] K. Wansbrough and S. P. Jones. Once upon a polymorophic type. In *Conf. Rec. POPL '99: 26th ACM Symp. Princ. of Prog. Langs.*, pp. 15–28, 1999.

[43] J. B. Wells, A. Dimock, R. Muller, and F. Turbak. A typed intermediate language for flow-directed compilation. In *Proc. 7th Int'l Joint Conf. Theory & Practice of Software Development*, pp. 757–771, 1997. Superseded by [44].

[44] J. B. Wells, A. Dimock, R. Muller, and F. Turbak. A calculus with polymorphic and polyvariant flow types. *J. Funct. Programming*, 200X. To appear. Supersedes [43].

[45] A. Wright and S. Jagannathan. Polymorphic splitting: An effective polyvariant flow analysis. *ACM Trans. on Prog. Langs. & Systs.*, 20:166–207, 1998.