Program Representation Size in an Intermediate Language with Intersection and Union Types

Allyn Dimock[1,*,†], Ian Westmacott[2,*,†], Robert Muller[3,‡,†,§], Franklyn Turbak[4,‡,†], J. B. Wells[5,*,‡,†,¶], and Jeffrey Considine[2,†,¶]

[1] Harvard University
[2] Boston University
[3] Boston College
[4] Wellesley College
[5] Heriot-Watt University

Abstract. The CIL compiler for core Standard ML compiles whole programs using a novel typed intermediate language (TIL) with intersection and union types and flow labels on both terms and types. The CIL term representation duplicates portions of the program where intersection types are introduced and union types are eliminated. This duplication makes it easier to represent type information and to introduce customized data representations. However, duplication incurs compile-time space costs that are potentially much greater than are incurred in TILs employing type-level abstraction or quantification. In this paper, we present empirical data on the compile-time space costs of using CIL as an intermediate language. The data shows that these costs can be made tractable by using sufficiently fine-grained flow analyses together with standard hash-consing techniques. The data also suggests that non-duplicating formulations of intersection (and union) types would not achieve significantly better space complexity.

1 Introduction

1.1 The Compile-Time Space Costs of Typed Intermediate Languages

Recent research has demonstrated the benefits of compiling with an explicitly typed intermediate language (TIL) [Mor95, PJ96, TMC+96, PJM97, JS98, BKR98, TO98, FKR+99, CJW00, MWCG99, WDMT0X]. One benefit is that

* Partially supported by NSF grant CCR–9417382.
† Partially supported by Sun grant EDUD-7826-990410-US.
‡ Partially supported by NSF CISE/CCR ESS grant 9806747.
§ Partially supported by a Faculty Fellowship of the Carroll School of Management, Boston College
¶ Partially supported by EPSRC grants GR/L 36963 and GR/L 15685.

explicit types can be used in compiler passes to guide program transformations and select efficient data representations. Another advantage of using a TIL is that the compiler can invoke its type checker after every transformation, greatly reducing the possibility of introducing errors. If strongly typed intermediate languages are used all the way through the compiler to the assembly level (something we do not yet do), the resulting object code is certifiably type safe [Nec97, MWCG99]. Furthermore, types that survive through the back end can be used to support run-time operations such as garbage collection [Tol94] and run-time type dispatch [Mor95].

The benefits of using a TIL are not achieved without costs. These costs include the space needed to represent the types at compile-time, the time to manipulate the types at compile-time, and the added complications of transforming types along with terms. This report focuses on the compile-time space cost.

Using a naive type representation can incur huge space costs, even if types are only used in the compiler front end for initial type checking. In the worst case, the tree representation of types in Standard ML (SML) programs can have size doubly exponential in the program size, and the DAG representation can be exponential in the program size [Mit96]. Although we are mainly concerned with ordinary programs where the worst case space complexity is not encountered, these ordinary programs often have types with impractically large tree representations but acceptable DAG representations. So in practice, DAG representations of types and other techniques are necessary to engineer types of tractable size. For example, the SML/NJ compiler's FLINT intermediate language uses hash-consing, memoization, explicit substitutions, and de Bruijn indices to achieve space-efficient implementation of types [SLM98]. The TIL compiler achieves type sharing by binding all types to type variables, and then performing dead code elimination, hoisting and common subexpression elimination on the types [Tar96, pp. 217–219]. The compiler must then preserve type bindings across transformations, or else repeat the type-sharing transformations. Tarditi reports that the representation size increase imposed by using types in TIL averages 5.15 times without this sharing scheme, but only 1.93 times with sharing.

We have constructed a whole-program compiler for core SML based on a typed intermediate language we call CIL[1]. Unlike FLINT and TIL, CIL has three features that make compile-time space issues potentially more challenging to address than in other typed intermediate languages:

1. **Listing-based types:** The CIL type system can encode polyvariant flow analyses using *polyvariant flow types* where labels on type constructors provide flow information and intersection and union types provide polyvariant analysis. Intersection and union types can be viewed as finitary (listing-based) versions of infinitary (schema-based) universal and existential types.

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

For example, CIL uses the intersection type

$$\tau_{\mathrm{id}} \equiv \wedge\{p_1 : \mathrm{int} \rightarrow \mathrm{int},\ p_2 : \mathrm{real} \rightarrow \mathrm{real}\}$$

to represent an occurrence of the universal type $\forall\alpha.\alpha \rightarrow \alpha$ that is instantiated only at types int and real. The intersection type τ_{id} is similar in structure to the CIL product (record) type

$$\tau_{\mathrm{funs}} \equiv \times\{p_1 : \mathrm{int} \rightarrow \mathrm{int},\ p_2 : \mathrm{real} \rightarrow \mathrm{real}\}.$$

The difference is that that a value of type τ_{funs} is a pair of two possibly distinct functions having the respective component types while a value of type τ_{id} is a single function having *both* component types. CIL union types (introduced via $\vee$) are the dual of intersection types; they are listing-based versions of existential types that are similar in structure to CIL sum (variant) types (introduced via $+$).

Encoding polyvariant analyses, which analyze a function multiple times relative to different contexts of use, can introduce components of intersection and union types that differ only by flow information. For instance, when encoding polyvariance, an innocuous type like $\mathrm{int} \rightarrow \mathrm{int}$ might expand into

$$\vee\{q_1 : \mathrm{int} \xrightarrow[\{3,4\}]{\{1\}} \mathrm{int},\ q_2 : \wedge\{r_1 : \mathrm{int} \xrightarrow[\{3\}]{\{2\}} \mathrm{int},\ r_2 : \mathrm{int} \xrightarrow[\{4\}]{\{2\}} \mathrm{int}\}\}.$$

In the function type notation $\sigma \xrightarrow[\psi]{\phi} \tau$, the annotation $\begin{smallmatrix}\phi\\ \psi\end{smallmatrix}$ is a *flow bundle* in which ϕ (resp. ψ) conservatively approximates the sites in a program that can be sources, or introduction points (resp. sinks, or elimination points) for the function values having this type. In this paper, we only show flow bundles annotating function types, but CIL supports such annotations on almost all types.

Intersection and union types have several advantages over universal and existential types as a means of expressing polymorphism [WDMT0X]: (1) by making usage contexts apparent, they support flow-based customizations in a type-safe way; (2) finitary polymorphism can type some terms not typable using infinitary polymorphism, thus potentially allowing some program transformations to be typable which would not be allowable in a TIL based on infinitary polymorphism; and (3) the listing-based nature of finitary polymorphic types can avoid some complications of bound variables in representing and manipulating quantified types (see Sec. 2.2). There is a space cost for these benefits: the listing-based nature of finitary polymorphic types, in combination with flow annotations encoding finer grained types, can lead to CIL types that are much larger than those expressed via infinitary polymorphic types.

Assuming whole-program compilation, the finitary polymorphism afforded by flow types is sufficient to compile SML programs. In this respect, the CIL SML compiler is similar to monomorphizing whole-program compilers [TO98, BKR98, CJW00].

2. **Duplicating term representations:** CIL represents the introduction of intersection types by a *virtual record* — a term that explicitly lists multiple copies of the same component term that differ only in their flow type annotations. For example, here is a CIL term that has the type τ_{id} defined above:

$$\wedge(p_1 = \lambda x^{\mathrm{int}}.x,\ p_2 = \lambda x^{\mathrm{real}}.x).$$

   Virtual record components are extracted via *virtual projections*. Similarly, values of union type (virtual variants) are introduced via *virtual injections* and are eliminated by a *virtual case expressions* — terms whose branches explicitly list multiple type-annotated versions of the same untyped branch. Virtual terms that persist until code generation are eliminated at that time. Code is generated for only one component of a virtual record and for one branch of a virtual case expression, and virtual projections and injections disappear entirely. Thus, these virtual term constructs have a compile-time space cost but no run-time space (or time) cost.
   Because it makes copies of terms that differ only in type annotations, we call CIL a *duplicating* representation. An advantage of the duplicating approach is that type information for guiding customization decisions is locally accessible in each copy of a duplicated term. An obvious disadvantage of this representation is the duplicated term structure, which is potentially much larger than the more compact introduction and elimination forms used for universal and existential types. Duplication arises in the CIL compiler whenever intersection or union types are used. The Type/Flow Inference and Flow Separation compiler stages discussed in Sec. 2.3 both introduce additional uses of intersection and union types.
3. **Closure types exposing free variable types:** CIL does not have universal or existential types because they hide important information about contexts of use and encourage uniform data representations rather than customized ones [WDMT0X]. However, existential types are particularly useful for abstracting over differences in free variables that are exposed in typed closure representations for functions of the same source type [MMH96, MWCG99, CWM98]. In the CIL compiler, these differences are reconciled by injecting the types of closures into a union type and performing a virtual case dispatch at the application site [DMTW97]. In a type-erasure semantics, these injections do not give rise to any run-time code. However, they can potentially cause a blowup in compile-time space when many functions with different free variables flow together.
   Our approach to closure conversion is similar to that used by TIL-based compilers that remove higher-order functions via defunctionalization [TO98, CJW00]. As in the CIL compiler, these compilers use flow analysis to customize the closure representation for particular application sites, although they support representing flow information in their type systems only after monomorphization and defunctionalization. These defunctionalizing compilers maintain type correctness during closure conversion by injecting closures

with different free variables that flow to the same application site into a sum type, and performing a case dispatch on the constructed value at the application site. The difference here is that in CIL this can be done with a mix of virtual and real sum types while in the defunctionalizing compilers all of the sum types must be real and hence require run-time analysis. 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 dropped. In contrast, in CIL the combination of a virtual sum (i.e., union) type with real closure types makes this approach well typed.

1.2 Contributions

Taken together, listing-based types, duplicating term representations, and closure types that expose free variable types raise the specter of compile-time space explosion at both the term and the type level. However, preliminary experiments with a small benchmark suite indicate that standard hash-consing techniques are able to keep the size of CIL types and terms tractable.

The main contributions of this paper are the following two observations:

1. **Duplicating term representations are practical:** Our experiments show that, for the flow analyses that we have investigated, the space required for CIL terms in our benchmarks is always within a factor of 2.1 of (and usually significantly closer to) our estimate of a minimal size for a non-duplicating TIL. This result is surprising, since we and many others expected the duplicating term representation to have a significantly higher space cost. Before we obtained these results, we expected that it would be essential to develop a *non-duplicating* term representation in which a single term schema somehow contains multiple flow type annotations. For example, using the notation of [Pie91], τ_{id} could be expressed as something like: **for** $\alpha \in \{\mathrm{int}, \mathrm{real}\}.\lambda x^{\alpha}.x$. Although this notation is more compact, it makes type information less accessible and can be tricky to adapt to more complex situations [WDMT0X]. We have made preliminary investigations into other representations, e.g., one based on the skeletons and substitutions of [KW99]. Based on the empirical results presented here, we believe that developing a non-duplicating representation of CIL may be not critical (though it may still be worthwhile). However, only one of the flow analyses we have experimented with to date expresses a non-trivial form of polyvariance, so it remains to be seen whether these results hold up in the presence of more polyvariant flow analyses.
2. **Finer-grained flow analyses yield smaller types and terms:** Our experiments indicate that, for some classes of flow analyses, increasing the precision of flow analysis can significantly reduce the size of program representations in CIL. Benchmarks require the most compile-time space for the least precise type-respecting flow analysis (one that assumes that any

function with a given monomorphic type can flow to any call site applying a function with this type). This imprecision leads to union types for closures that are much larger than necessary. More precise flow analyses can substantially reduce the size of these closure types.
Flow analysis has similarly been used to reduce the size of closure types in monomorphizing and defunctionalizing TIL compilers [TO98, CJW00]. However, previous work has neither quantified the benefits of using flow analysis in this context nor studied the effects of different flow analyses on compile-time space. We believe that we are the first to present a detailed empirical study of the effects of a variety of flow analyses on program representation size.

1.3 Representation Pollution

In addition to our results about the tractability of compile-time space in the CIL compiler, we have preliminary evidence that the compiler may be able to achieve one of its main design goals: avoiding *representation pollution* when choosing customized data representations. Representation pollution occurs when a source form is constrained to have an inefficient representation because it shares a sink with other source forms using the inefficient represention. A complementary phenomenon occurs with pollution of sink representations.

As an example of representation pollution, as well as some other issues that arise in a compiler based on CIL, we will consider the compilation of the untyped CIL source term in Fig. 1.[2] The term contains two abstractions, two applications (denoted by the @ symbol), and a tuple introduction form (introduced via $\times$[3]). The abstraction $(\lambda x.x * 2)$ flows to both application sites while the abstraction $(\lambda y.y + a)$ flows only to the rightmost application site.

$$
\begin{array}{l}
\textbf{let } f = (\lambda x.x * 2) \\
\textbf{in let } g = (\lambda y.y + a) \\
\quad \textbf{in } \times(f \ @ \ 5, (\textbf{if } b \textbf{ then } f \textbf{ else } g) \ @ \ 7)
\end{array}
$$

Fig. 1. An untyped CIL term.

The diagram in Fig. 2 gives an abstract depiction of a CIL compiler intermediate representation of the code in Fig. 1 that might emerge from the Type

[2] We introduce and explain elements of the CIL language on an "as needed" basis in the context of our examples; readers interested in the details of the language and its type system should to consult the appendix of the companion technical report [**?**].

[3] In CIL, as in ML, a *tuple* is a record with implicit positional labels. In general, the notation $P(M_1, \ldots, M_n)$ is a shorthand for $P(\mathrm{f}_1 = M_1, \ \ldots, \ \mathrm{f}_n = M_n)$, where P ranges over $\times$ and $\wedge$, and f_1, f_2, $\ldots$, is some fixed infinite sequence of distinct field names. Similarly, $Q[\tau_1, \ldots, \tau_n]$ is shorthand for $Q\{\mathrm{f}_1 : \tau_1, \ \ldots, \ \mathrm{f}_n : \tau_n\}$, where Q ranges over $\times$, $+$, $\wedge$, and $\vee$.

Inference/Flow Analysis (TI/FA) stage of the compiler. The TI/FA stage (described in more detail in Sec. 2.3) computes an approximation of the flow of values between sources and sinks in the input term and represents the analysis in the output typing. In this case, the CIL representation of the source term $(\lambda x^{\mathrm{int}}.x * 2)$ has been *split* into the virtual tuple

$$\bigwedge\Big(\lambda^1_{\{3\}} x^{\mathrm{int}}.x * 2,\ \lambda^1_{\{4\}} x^{\mathrm{int}}.x * 2\Big),$$

which contains one copy of the function for each of the application sites to which it flows. The notation λ^ℓ_ψ denotes an abstraction labelled ℓ that may flow to the sinks whose labels are in the set ψ, while $@^\phi_k$ denotes a sink labelled k to which abstractions whose labels are in the set ϕ may flow. Free variables and λ-bound variables are superscripted with their type. Terms of the form $(\pi^\wedge_i\,\square)$ are *virtual tuple projections* that select the ith component of a virtual tuple.

The typing rules of CIL (not detailed here) guarantee that the flow annotations appearing in CIL types are sound. That is, an abstraction may only be applied at sites listed in its sink set, and only the abstractions appearing in the source set of an application site may be applied at that site. In Fig. 2, the type of the first component of the virtual tuple $(\mathrm{int} \xrightarrow[\{3\}]{\{1\}} \mathrm{int})$ is the type required for the function position of the application site $@^{\{1\}}_3$ to which the function flows. The type on the second component of the virtual tuple $(\tau_1 = \mathrm{int} \xrightarrow[\{4\}]{\{1\}} \mathrm{int})$ does not match the type $(\tau_3 = \mathrm{int} \xrightarrow[\{4\}]{\{1,2\}} \mathrm{int})$ required at its application site $@^{\{1,2\}}_3$, so this component value must be coerced to the correct type somewhere along the flow path to the application site. A subtype coercion from a term M of type σ to a supertype τ of σ is witnessed by an explicit term of the form $\mathbf{coerce}\,(\sigma, \tau)\,M$.

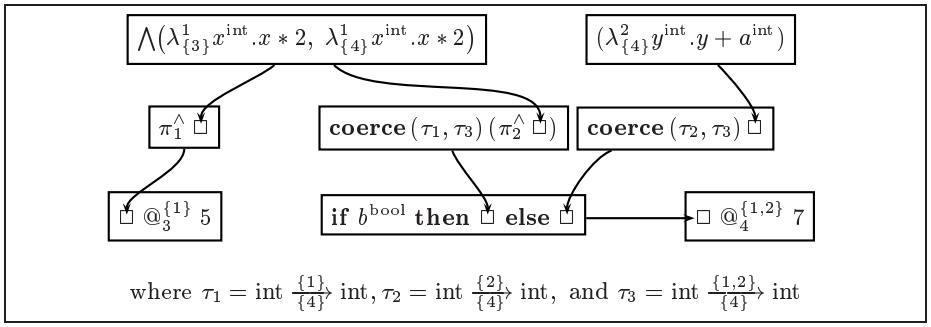

Fig. 2. A possible result of Type Inference/Flow Analysis.

The typing rules also require that the type erasures of all the components of a virtual record and all the branches of a virtual case expression must be the same. The *type erasure* of a term is the untyped terms that result from eliminating all types, labels, and virtual forms (virtual records, virtual projections, virtual

injections, virtual case expressions, and coercions) from the term. This type erasure constraint guarantees that virtual record components and virtual case expression branches are just different typings of the same untyped term and can therefore share the same run-time representation if the virtual forms survive to the code generation phase. If the compiler elects to customize the representations of the components of a virtual record, the virtual record will be *reified* into a real record (by changing $\wedge$ to $\times$ in terms and types) that is explicitly represented in the run-time code. Similarly, by changing $\vee$ to $+$, the compiler can reify a virtual case expression to be a real case expression that performs a dispatch on a real variant at run-time. The compiler is designed so that reifying virtual forms in this manner is type-safe.

As representation decisions are made during subsequent stages of compilation, further duplication may occur. Fig. 3 depicts a possible output of the Flow Separation stage.This stage (described in more detail in Sec. 2.3) introduces new virtual forms to guarantee that the output of the later Representation Transformation stage will be well-typed. In Fig. 3, the Flow Separation stage has split the application site $@_4^{\{1,2\}}$ into two applications sites $@_4^{\{1\}}$ and $@_4^{\{2\}}$. These applications occur within a virtual case expression, which has the form

$$\mathbf{case}^{\vee}\ M_{\mathrm{disc}}\ \mathbf{bind}\ x\ \mathbf{in}\ \tau_1 \Rightarrow M_1 \ldots \tau_n \Rightarrow M_n.$$

A virtual case expression dispatches to the branch $\tau_k \Rightarrow M_k$ based on the positional tag k of the of the discriminant M_{disc}, which must have type τ_k; within the chosen branch, the variable x of type τ_k is bound to the value of M_{disc}. In Fig. 3, the functions formerly flowing to the single application site $@_4^{\{1,2\}}$ are now injected into virtual variants (values of union type τ) via $(\iota_i^{\vee}\ \Box)^{\tau}$, where i in $\{1,2\}$ is the positional tag of the variant. These virtual variants both flow to the discriminant position of the virtual case expression, which chooses one of the two type-annotated versions of the application $h\ @\ 7$. Splitting $h\ @\ 7$ in this manner gives the compiler the option to use different representations for the closed abstraction $\lambda^1_{\{4\}}$ and the open abstraction $\lambda^2_{\{4\}}$.

As with source splitting, this kind of sink duplication increases the size of the compile-time representation of the program, but the object code size and run-time space costs increase only if some of the virtual variants and virtual case expressions are reified in a subsequent compilation stage. Observe that the sink duplication introduced by Flow Separation in this example has eliminated the need for both of the coercions present in Fig. 2 and will usually reduce the sizes of flow sets. In general, there are many trade-offs between the amount of virtual duplication and subtype coercion. The trade-offs are very sensitive to the granularity of the flow analysis and to the representation customization strategy.

We have developed several strategies for reducing (and in some cases completely eliminating) representation pollution in the case of function representations (see Sec. 2.3). More work is necessary to evaluate the run-time aspects of the customization capabilities of the CIL SML compiler. In a future report we will present a detailed study of the run-time consequences of compiling with polyvariant flow types.

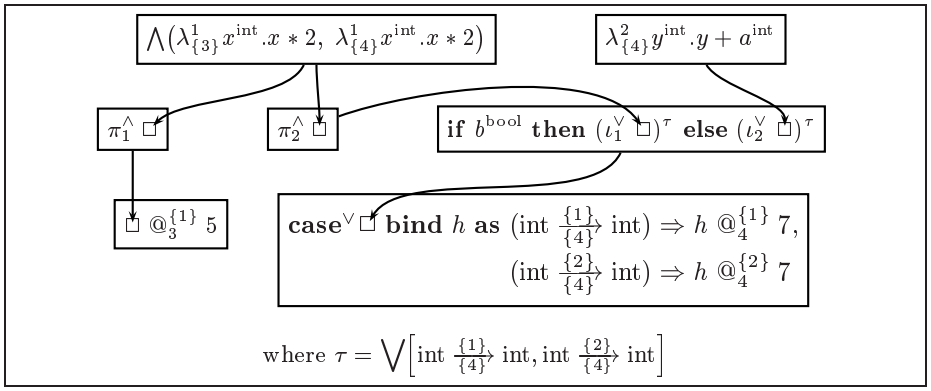

Fig. 3. A possible result of Flow Separation.

1.4 Outline

The remainder of this paper is organized as follows. Sec. 2 provides an overview of the CIL compiler for SML. Sec. 3 presents space-related measurements for several standard benchmark programs at various phases of compilation. Sec. 4 summarizes our conclusions and describes future work.

2 An Overview of the CIL Compiler

2.1 The Intermediate Language

To implement the features of core SML, CIL, extends the purely functional λ^{CIL}-calculus [WDMT0X] with primitive datatypes, references, arrays, and exceptions. The syntax and typing rules of CIL are summarized in appendix A. Although CIL is based on the λ^{CIL}-calculus, CIL itself is not a calculus. We have implemented a semantics for CIL, but we have not written its formal counterpart. While we have proven formal properties like standardization, subject reduction, and type soundness for the λ^{CIL}-calculus, we have not yet established any of these properties for CIL.

2.2 Type and Term Representations

To keep the sizes of types tractable, the CIL compiler uses hash-consing to represent types as compact directed acyclic graphs instead of as trees. This is similar to the type representation in the SML/NJ compiler's implementation of its FLINT intermediate language [SLM98]. One important issue faced in FLINT is not an issue for CIL. FLINT types have higher-order features such as abstractions and applications, i.e., a λ-calculus inside the types. Because FLINT types are identified modulo β-conversion, and because eager β-normalization of types can lose sharing and do excess work, the hash-consing scheme for FLINT types

uses explicit substitutions [KR95] and memoization of substitution propagation steps. Unlike FLINT, the CIL types do not have such higher-order features, so the CIL hash-consing of types is simpler.

Sets of flow labels are often used by many types and/or terms. A single copy of each set is shared by all uses. Using the duplicating representation for terms, two CIL term occurrences are rarely structurally equivalent, so we do not use hash-consing for terms. However, the types and flow sets annotating terms are hash-consed, as described above. Strings, used for record field names and constructor names, are also shared by all uses and lists of strings are hash-consed.

2.3 Compiler Architecture

The architecture of the CIL compiler [DMTW97] is summarized in Fig. 4. This section briefly describes the compilation stages depicted in the figure.

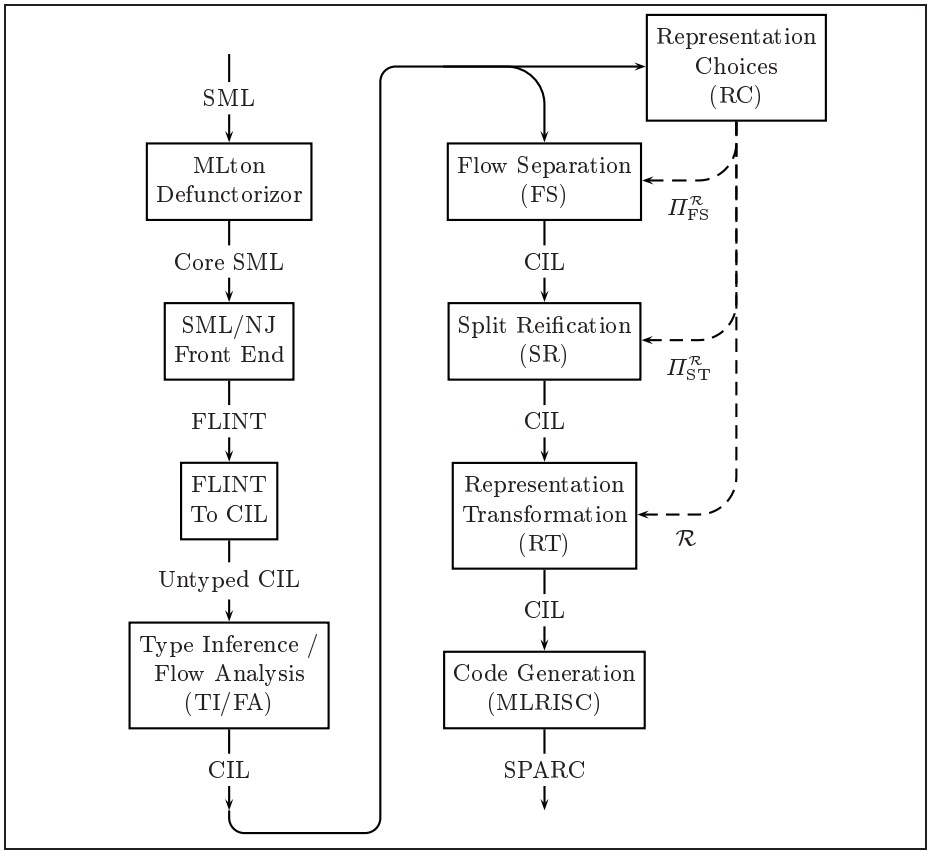

Fig. 4. Compiler Architecture.

Defunctorizing, Parsing, Elaboration. 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 [CJW00] 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.

Type Inference/Flow Analysis(TI/FA). This stage accepts an untyped CIL term (plus some of the FLINT type information) as input and returns a typed CIL term as output. The typed term encodes a flow analysis that is a conservative approximation of the run-time flow. The TI/FA module supports flow analyses that vary with respect to the precision of the approximation.

We currently support five different flow analyses. In this paper, we present data from two of these: what we call *typed source split* and *min type respecting*. The *typed source split* analysis is an variant of Banerjee's [Ban97] modified for shallow subtyping [WDMT0X]; the use of shallow subtyping makes it slightly less precise than the combination of monomorphization and 0CFA analysis. It introduces virtual tuples and virtual projections but neither virtual variants nor virtual case forms.

The *min type respecting* analysis is the least precise flow analysis that is still type-correct (cf. [JWW97]). It conflates the flow information on all values of the same flow erased type. For example, an abstraction of type **int** $\rightarrow$ **int** will be assumed to flow to every application site whose rator has this type. This analysis models a monomorphizing compiler in which types carry no useful flow information.

We have also implemented a finer analysis that splits some **let** and **letrec** definitions based on variable occurrences. Both *typed source split* and this *limited let split* analysis may be implemented either with shallow subtyping constraints, or with equality constraints.

The granularity of the flow analysis is important to the program size: A coarser grained flow analysis will show more functions flowing to a given call site than will a finer analysis. A coarser grained analysis may also show more functions bound to an environment variable in a closure representation[4].

Representation Choices (RC). This module selects representations for a function that are adequate for each of the application sites to which it flows. Four different function representation choice strategies have been implemented. The *uniform* strategy represents all functions with closure records having the

[4] The interaction between flow analysis granularity and our flow-based version of the known function optimization leads us to present data with the known function optimization turned off, so as not to inflate the size differences between different the profiles for different flow analyses.

type

$$\times\{\text{code} : \{\text{arg} : \tau_{\text{arg}},\ \text{env} : \tau_{\text{env}}\} \rightarrow \tau_{\text{body}}, \text{env} : \tau_{\text{env}}\},$$

where the *code* field contains a closed function and the *env* field contains a record of the values of the free variables of the function. A closure data structure is applied to an argument by projecting both fields from the closure record and applying the function from the code field to an argument record consisting of (the closure conversion of) the actual argument packaged together with the projected environment.

The other three representation strategies generate specialized representations based on various conditions detected in the term structure. Wand and Steckler [WS94] coined the term "selective" representation to refer to representations of functions that do not include an environment component. A selective representation is adequate for a closed function if the function flows only to call sites with compatible application protocols. In [WS94], selective representations were disabled in the presence of representation pollution — i.e., when a closed function shared a call site with some number of open functions. In contrast, the CIL compiler can still use selective representations in such situations removing the pollution via a *splitting strategy*.

The *selective sink splitting* strategy implemented in the CIL compiler 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. The transformation of the program depicted in Fig. 2 to the one depicted in Fig. 3 is a sample application of the selective sink splitting strategy. It is also possible that selective sink splitting will cause virtual records created by TI/FA to be reified into normal records if, e.g. a selective representation is chosen for an call site in one element of the virtual record, and a closure representation is chosen for the corresponding call site in a different element of the virtual record.

The *selective source splitting* strategy generates a selective representation for a closed function flowing to call sites that are not shared with open functions. Under this strategy, 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 several copies of the function. The appropriate representations are projected from the record somewhere along the flow path to the respective call sites.

The final representation strategy inlines (possibly open) functions at the call site. The inlined representation of a function consists of the record containing the values of the function's free variables. It is possible to specify many different inlining heuristics. Currently, the inliner will select an inlined representation for any non-recursive function flowing to two or fewer call sites.

The selective sink splitting generates more duplication than the selective source splitting strategy, and is thus of more interest in this paper.

Flow Separation (FS). This stage accepts as input a typed program and a flow-path partitioning function ($\Pi^{\mathcal{R}}_{\mathrm{FS}}$) supplied by RC. It specifies which flow paths can coexist in the same flow bundles. For flow paths that cannot coexist in the same bundle, the FS phase will introduce whatever coercions and virtual forms (i.e., virtual variant injections, virtual case expressions, virtual tuples, or virtual tuple projections) are required to ensure that the result of the later Representation Transformation stage will be well-typed.

Split Reification (SR). This stage accepts as input a typed term and a flow-path-partitioning function ($\Pi^{\mathcal{R}}_{\mathrm{ST}}$) supplied by RC. This phase reifies whatever virtual forms are required to remove representation pollution. We refer to the reification process as *splitting* because it causes the code generator to generate multiple copies of a term in situations where only one copy would have been generated without reification. In general, the current simple algorithm may split more than is necessary[DMTW97]. Specifying and implementing a more efficient splitting algorithm remains for future work.

Representation Transformation (RT). This stage accepts as input a typed term and a representation map ($\mathcal{R}$) provided by RC. It walks the term and installs the function representations specified by the map. The FS stage only introduces virtual forms, and the SR stage only reifies virtual forms. The RT stage performs the actual work of changing the code for specialized representations. For instance, in the case of selective closure conversion, it is RT which changes some functions to closures, and some call sites to calls to closures.

An interesting aspect of the transformation is that the result of the transformation may have a recursive type even though the source of the transformation has no recursion in either terms or types. Recursion through flow labels in the source term may be enough to cause the transformed term to have a recursive type.

Code Generation. 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 [Geo97]. CIL programs are translated into the MLRISC intermediate language, and the framework is specialized with CIL conventions for each target architecture.[5] MLRISC handles language-independent issues such as register allocation and code emission.

[5] Although an advantage of the MLRISC framework is its portability, it still requires substantial work to port a code generator based on MLRISC. For this reason we have concentrated only on the SPARC architecture to date.

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 [Boe93]. CIL programs use stack-allocated activation records, which have a layout similar to C stack frames. Basis functions are called through the foreign function interface, which provides data and activation record conversions between CIL and foreign languages. The code generator does not yet optimize tail recursion.

CIL data representations are straightforward. Records, arrays, references, and strings are heap-allocated and include size headers[6]. Exception identifiers and all other constants are immediate. Injections may either be immediate or heap allocated, depending on the number and type of summands in their type.

Recursive bindings are restricted to values, as defined in Fig. **??** (see appendix A). The extended notion of value presented there ensures that terms bound to variables in recursive definitions cannot diverge, affect the store, or raise exceptions. Although input programs must adhere to SML restrictions on recursive definitions (because we use the SML/NJ elaborator), compiler transformations may (and do) create recursive definitions which bind extended values to variables. The extended value restriction allows the code generator to use a two phase algorithm for recursive bindings: the first phase allocates memory for the values, while the second phase fills them in.

3 Representation Measurements

The main purpose of this paper is to determine whether CIL has acceptable compile-time space costs and to evaluate how flow analysis and representation strategy combinations affect these costs. This section presents data indicating that CIL is tractable as a compiler intermediate language when used with a reasonably fine-grained flow analysis.

3.1 Space Profiles

We have tested the CIL SML compiler for most combinations of flow analyses and function representation strategies on 22 kernels and small benchmarks taken from the O'Caml, TIL and SML/NJ benchmark suites. Figures 4 and 5 present space profiles for a geometric weighted average of all our benchmarks, and profiles for five individual benchmarks for two flow analyses and two function representation strategies. We show data for the *uniform* function representation strategy to indicate the amount of data needed to correctly closure convert functions without customizing representations. We show the *selective sink splitting* strategy as an example of a strategy that customizes function representations. The *typed source splitting* flow analysis is currently our most accurate analysis

[6] Such headers are currently unnecessary since we use conservative GC. But it is expected that in the future we will develop customized memory management.

that does not split on variable occurrences. The *min type respecting* flow analysis is included to show size bloat that can occur when flow analysis provides no information beyond the type.

Each space profile shows intermediate representation size information at various CIL compiler stages. The legend in Figure 4 explains how to interpret the data. Of particular importance is the position of the horizontal tick mark found in each bar of a profile. The portion of the bar below the tick mark is our conservative estimate of the space that might be required for a hypothetical non-duplicating representation of the term (including the space for type and flow information in such a term). The position of the horizontal tick mark is computed as the term size ignoring all but the leftmost branches of virtual records and virtual case expressions. Virtual record nodes and virtual case nodes are included in the count because they serve as markers for intersection type introduction and union type elimination points. We assume that such markers would be required in any non-duplicating representation. Virtual projection and virtual injection nodes and included to approximate (resp.) the markers required for intersection type elimination and union type introduction forms. Finally, the count also includes coercion nodes.[7]

The size information was gathered by adding a function to the SML/NJ runtime system which runs the *mark* stage of the SML/NJ garbage collector using a particular object as the root. The function reports the size of all marked objects that are reachable from the root object. We present all size information in bytes rather than in type or term constructor nodes. We find that the average size of our type nodes and of our term nodes for a given benchmark is generally in the range of 10 to 12 times the size of a machine word.

3.2 Interpretation of the Space Profiles

Interpreting the size of the untyped term. When compiling small programs, the untyped CIL code, **U**, is smaller than the typed FLINT code, **F**. For benchmark programs of any reasonable size, the untyped CIL code is slightly larger than the typed FLINT code. This is due in part to the fact that the CIL representation carries more information about records and datatypes than does the FLINT representation. Of the profiles shown in this paper, only **quad** shows less space for untyped CIL than for FLINT; in all other cases that we show, the untyped CIL code is larger than the FLINT code. While other small benchmarks are smaller in untyped CIL than in FLINT, the weighted average shows that untyped CIL is usually the bulkier representation.

The **F** and **U** columns are not quite comparable for several reasons. The **F** column overestimates the size of the FLINT code in the sense that it includes the size of FLINT type information. FLINT and CIL also differ in terms of which basis functions are compiled with the program and which are pre-compiled in the run-time system.

[7] An even more conservative approximation of the space required for a nonduplicating representation would be the size of the type-erased term. We believe that this is unrealistically small.

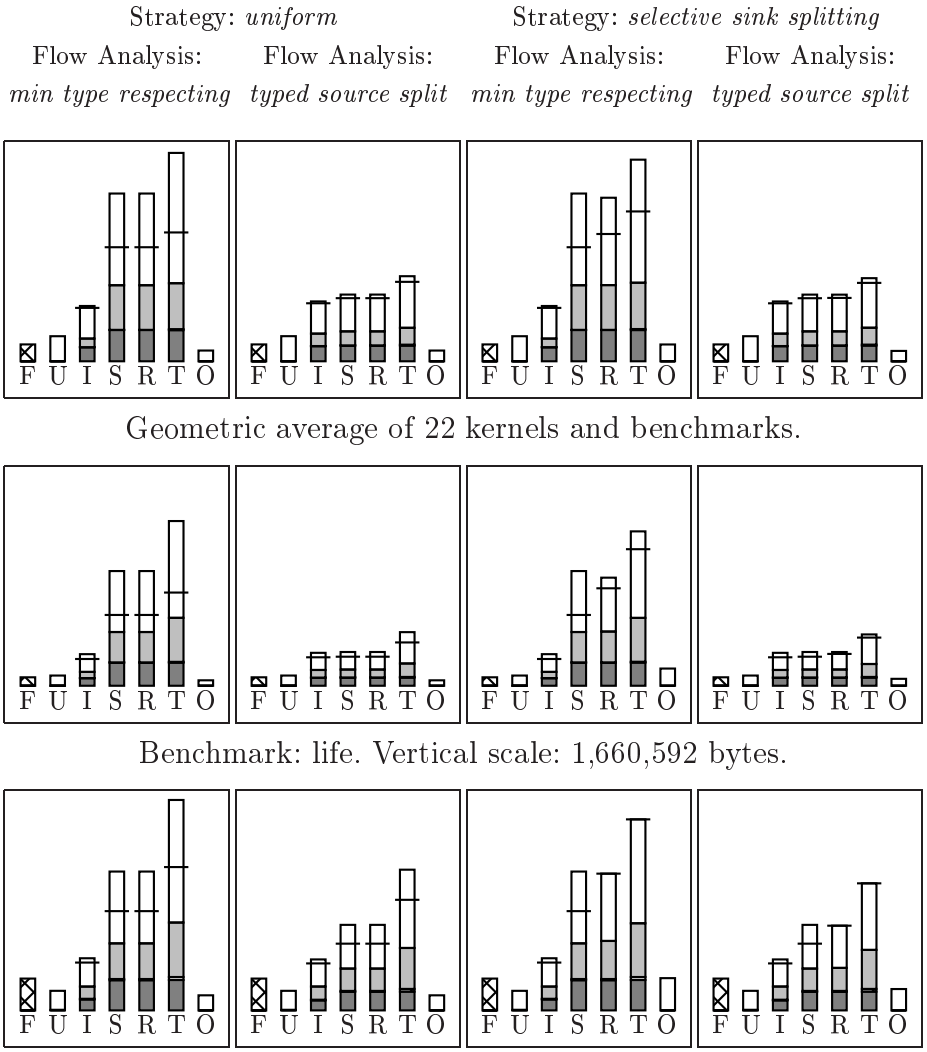

Fig. 5. Sizes of benchmark phases by strategy and flow analysis I

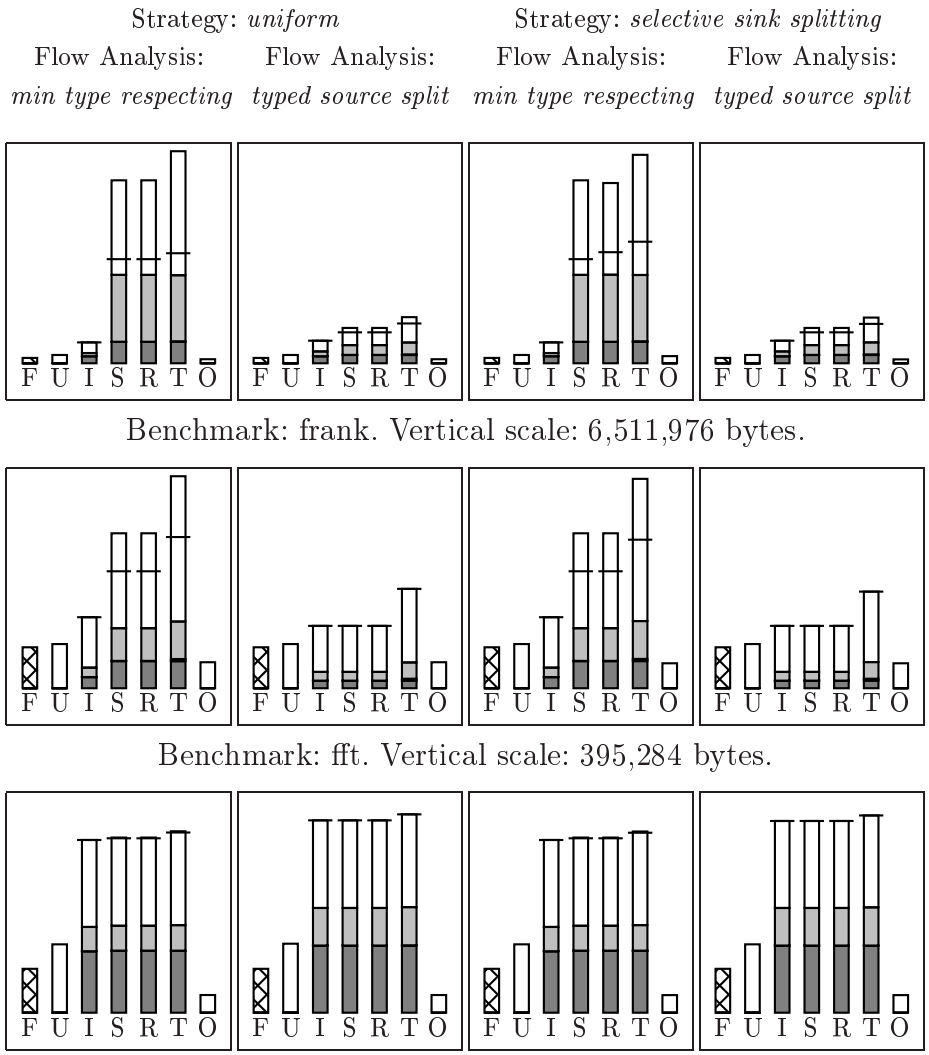

Fig. 6. Sizes of benchmark phases by strategy and flow analysis II

Columns **F** and **U** are independent of the flow analysis or the function representation strategy, but are repeated in each profile as reference points.

Interpreting the output of the Type Inference/Flow Analysis stage. Column **I** shows the size of the typed and flowed term output from the TI/FA stage. As illustrated by the representative space profiles, the TI/FA pass can expand the size of the term by introducing virtual nodes. In monomorphic benchmarks, (e.g., **boyer2**, **fft**, and **frank**), term size is only increased by the addition of **coerce** forms that indicate where subtyping is used. In benchmarks with polymorphic functions (e.g., **life**, and **quad**), the TI/FA stage makes one virtual copy (using $\wedge$) of each polymorphic function at each flow-erased type at which the function is used.

In the two flow analyses shown, the distance of the tick mark from the top of the **I** bar reflects the amount of type polymorphism in the benchmark. In general, the tick mark indicates the amount of *polyvariance* of the analysis, which, for some analyses, may be substantial even for monomorphic code.

Interpreting the output of the Flow Separation stage. Column **S** shows the size of the output from the FS stage. The FS stage introduces whatever new virtual constructs are required to ensure that the result of the (later) RT stage will be well-typed. For example, abstractions that share a call site may have the same type, up to flow information, after the TI/FA stage, but may differ from each other in the number, name and types of free variables. The FS stage must create types that differ in structure as well as in flow information for these different terms.

Under the uniform strategy, the growth in size from **I** to **S** is due only to differences in the environment component of closures – differences that will not be reflected in the object code. In other strategies, some of the growth may be due to function representations that require different object code.

The growth in size from **I** to **S** depends on the accuracy of the flow analysis. In the *min type respecting* flow analysis, the labels for all abstractions of a given (flow erased) type appear in the source label set for each application site for that type. This requires the flow separator to introduce larger intersection and union types, and to perform more virtual term duplication than would be required for a finer flow analysis. This is seen consistently throughout the data, with **frank** being the most dramatic example, and **boyer2** being the least dramatic. The **frank** benchmark is a combination of human written code for a Warren Abstract Machine using some curried and higher-order functions, and machine generated code to play a solitaire game on the WAM. The machine-generated code contains many different anonymous functions of the same few types but with different free variables. The *min type respecting* flow analysis causes these calls to be conflated. The **boyer2** benchmark is a tautology checker which has been written in closed,

uncurried, first-order style. In **boyer2**, all abstractions are closed up to names of known functions[8], so there are few free variables requiring separation.

Interpreting the output of the Split Reification stage. Column **R** shows the size of the output from the SR stage, which reifies some virtual constructs — splitting them to take advantage of differing representations. The number of term and type nodes remains the same because the transformation is merely changing virtual entities to real ones.[9]

Under the *uniform* strategy, the **S** and **R** columns show identical tick mark positions. This is expected because we implement only a single function calling convention for the uniform strategy. Under the *selective sink splitting* strategy, the position of the tick mark may change upwards due to reification of virtual constructions: this is what we expect from splittings introduced to circumvent representation pollution and to insert customized data representations. This is shown most dramatically in **quad** (a kernel repeatedly applying a doubling function), in which all virtual constructs are reified. In contrast, the **fft** (Fast Fourier Transform) benchmark shows no pollution of function representations when compiled with the *selective sink splitting* strategy. Most functions in **fft** are open, but the control flow structure of **fft** is quite simple: just nested loops, so open functions and closed functions never flow together.

If we see even a little reification for a strategy, we know that some part of the transformed program will use a simpler representation. If this change is in an inner loop, then a single reification may dramatically affect program performance. To determine the effectiveness of a strategy, we need to show data about the performance of the transformed programs — something outside the scope of this paper.

Our current SR stage is quite simple: If it encounters two different representations in a single virtual construct, then it converts the virtual construct into the equivalent real construct. Our current splitting algorithm can oversplit because it reifies a virtual form whenever it contains components that require different representations. But given an n-way virtual form whose components require $m < n$ different representations, the virtual form could be replaced with a real form containing m virtual forms. Oversplitting will result in unnecessary duplicated code in the object file. Oversplitting impacts the performance of the generated code when the m-way real form could be more efficiently compiled than the n-way form. We have neither measured the amount of oversplitting arising from the current algorithm nor have we experimented with other splitting algorithms.

[8] In the current version of the CIL compiler, known function names are treated as free variables. This will improve in future versions.

[9] The size of the term component decreases slightly in some profiles due to assymetries between virtual and real injections in the current implementation (e.g., **life**, with strategy = *selective sink splitting* and flow analysis = *min type respecting*).

Interpreting the output of the Representation Transformation stage. The type information in a closure-converted term is larger than in the pre-converted term. This is visible in the profiles for all the benchmarks. Part of this growth is in the creation of types for the required closure and argument records. Part of this growth is the creation of types for environments. In our framework, programs with more open terms will experience more growth in types.

The introduction of closure and argument records and the storage of free variable values in environments causes an increase in term size. In our implementation of closure conversion, the major increase in term size is from projections from the environment: our implementation puts in a projection from the environment wherever a free variable occurs. The creation and destructuring of closure and argument records will show different percentage effects in different benchmarks depending on the relation of the number of abstractions and applications to other term constructors.

The **boyer2** benchmark has the highest ratio of closed to open terms, so its term size grows, essentially, only by introduction of closure and argument records. In this case the growth in size is relatively small. In contrast, **fft** has a high percentage growth.

The ratio of the size of the CIL representation to the size of a non-duplicating TIL can decrease in the RT stage, but can never increase since the size of the types can only grow, and since no more duplication is introduced into the term at this stage.

Duplicating vs. nonduplicating intermediate representations. Columns **I**, **S**, **R** and **T** have tick marks showing our estimated lower bound on the size of a typed and flowed term in a non-duplicating TIL. The position of the tick mark shows that in the benchmark programs presented (and so far in all benchmarks that we have tried), the space used in CIL's duplicating term representation is never more than twice our estimate for a non-duplicating representation. This is both surprising and encouraging. However, it remains to be seen whether these results hold up in the presence of more polyvariant flow analyses.

Coarse vs. fine flow analysis. We have shown that the choice of flow analysis can greatly influence the growth in term size needed to produce well-typed function representations. The most dramatic example occurring in the benchmark **frank**, where, for the *uniform* function representation strategy the *min type respecting* analysis resulted in a size after Flow Separation 5.2 times the size of that produced using the *typed source split* analysis. At the other extreme, the benchmark **boyer2** shows a slight decrease in overall size from *typed source split* analysis to *min type respecting* analysis. The *min type respecting* flow analysis yields a smaller number of flow types for the number of underlying flow erased types than the *typed source split* analysis. In the case of boyer2, the slightly larger term size using *min type respecting* analysis is offset by the significantly smaller size of the flow types.

We have accumulated some data so far for the version of *typed source split* using only equality constraints. This analysis can be thought of as performing Henglein's "simple" flow analysis [Hen92] over monomorphized code, and is the flow analysis used in the RML compiler [TO98]. As expected, profiles generated using this analysis generate somewhat larger code in many cases, than profiles generated with the usual *typed source split*, but are much closer to the profiles for *typed source split* than they are to the profiles for *min type respecting*.

We have implemented an analysis, *limited let split*, which causes some **let** and **letrec** bound definitions to be duplicated per occurrence of the bound variable, rather than just once per type. In this analysis, benchmarks **life** after the RT stage, and **simple** after TI/FA stage (but not subsequently), show a ratio of CIL code size to non-duplicating TIL code size of 2.1. The code size ratios are less than 2 for all other compiler phases and benchmarks in our benchmark suite. A study of aggressive nested cloning in a lazy functional language, shows code size increases of a factor of up to 3 for some benchmarks of up to 800 lines of code. That study also shows that, when identical clones are merged after transformation, the code size increase is only a factor of 1.2 citeFaxen:SFPW-2000.

We have not accumulated any data for the combination of *limited let split* and equality constraints.

The cost of accurate closure types. The profiles give us some idea as to the compile-time space cost of accurately representing closure types. With *uniform* function representation and *typed source split* analysis the growth in size from the output of Type Inference/Flow Analysis stage to the output of the Representation Transformation stage shows the space needed for closure types and for virtual cases where multiple closures flow together. This growth ranges from the size of RT output 1.03 times the size of TI/FA output for **boyer2** to 2.76 times for **quad**. The ratio of the types sizes is 1.02 for **boyer2** and 3.11 for **quad**. **quad** is atypical, being a very small program constructed to have relativly large types.

4 Conclusions and Future Work

We have shown that the amount of space used in compiling SML with CIL terms and types is practical on our benchmarks for the more precise flow analyses that we have investigated. Most importantly, the term sizes in our straightforward duplicating representation are never more than twice our underestimate of term sizes using a non-duplicating representation. Transformations that use type and flow information on virtual terms to generate customized data representations would be more difficult to engineer in a non-duplicating representation. A factor of less than two in space is acceptable to avoid further complicating the transformations.

This is the kind of result that requires benchmarking to determine, as it depends on the style in which programs are written. It appears to be that case that

for the human written and machine generated programs which we have been able to test that (1) the bulk of the program code is not used in a highly polymorphic manner so that a whole program analysis finding actual polymorphism rather than potential polymorphism need not perform too much duplication – this limits the number of virtual records created in type inference; (2) A reasonable flow analysis will find that a large percentage of calls in most programs are direct calls – this limits the number of virtual cases created in the Flow Separation phase for correctness of typed closure conversion, and for pollution removal in the selective sink splitting strategy.

The typical non-trivial growth in size from the result of TI/FA to the result of RT is obviously undesirable, and might be smaller in an intermediate representation that could hide environment types with an existential quantifier. This raises the question of whether the more precise type information maintained in CIL after closure conversion without the $\exists$ type quantifier is useful in terms of transforming a program for better run-time performance. If not, we should extend CIL with existential types.

Although the standard technique for hash-consing types sketched earlier is the one used to generate the statistics for this paper, we have almost finished changing to a new type hash-consing scheme, which we expect to give much better performance. The motivation for the new scheme is due to the combination of (1) the pervasive use of recursive types in CIL and (2) the fact that the CIL type system identifies recursive types with the infinite trees that result from unwinding them infinitely. The new scheme represents types as directed graphs and implements recursion using cycles. The use of cycles to represent recursion automatically causes α-equivalent types to be shared — the variable names are no longer present leaving only the structure of the recursive type to be stored in this representation. It will also avoid the need to have type manipulation special-case the type recursion form (which can currently appear anywhere). The new scheme uses a method of incremental DFA minimization to maintain the invariant that each possible type is represented by at most one node in the graph. This will allow constant-time type equality checking, which our current hash-consing scheme does not support due to the possibility of differing representations of the same recursive type.

Our new method of incremental DFA minimization to represent all types in the same graph is similar to a method suggested by Mauborgne [Mau00], but was developed completely independently. Our method needs $O(n \log n)$ space to store the types, while Mauborgne's needs $O(n^2 \log n)$ space, where n is the number of distinct types and some upper-bound on the arity of type constructors is assumed. Also, even in cases where Mauborgne's method approaches linear space complexity, ours will typically use half as much space.

Encoding more flow analyses in CIL remains an important area for future work. Recent work has shown that many standard flow analyses, such as k-CFA [Shi91, JW95, NN97] and the cartesian product argument-based analysis [Age95] can be encoded into a type system with intersection and union types and flow labels [PP0X, AT00]. However, unlike CIL, these type systems have deep sub-

typing. We are exploring a translation between deep and shallow subtyping that will allow us to employ these recent theoretical results in the CIL compiler. We are eager to see how highly polyvariant flow analyses affect our results regarding the duplicating term representation.

There are many areas for improvement in the CIL compiler as a whole. The compiler can benefit from many standard optimizations not yet implemented (e.g., tuple flattening and special handling of known calls to global functions) as well as some important non-standard optimizations (e.g., the complete removal of polymorphic equality). Several existing algorithms can be more efficiently implemented, such as the algorithm used in Split Reification. There are also many opportunities for improvement in the representation of the intermediate language.

We have designed and implemented a general framework for generating customized data representations, but work remains to be done in optimizing those representations and developing heuristics for choosing between allowable representations. In terms of function representations, we are currently investigating function representations that do not close over variables whose values are available on the stack (the so-called *lightweight* closure conversion of [SW97]), higher-order uncurrying [HH98], removing manipulation of records with known components (along the lines of the *fictitious data* elimination in [Sis99]), and register allocation and calling conventions informed by flow information. We have yet to explore customized representations for other kinds of data, but CIL is rich enough to support flow-directed representation transformations for all types of data.

Finally, we emphasize that this report has focused only on compile-time space issues. In the future, we will report on compile-time time complexity as well as run-time space- and time-complexity.

Acknowledgments

This paper, and the CIL compiler project in general, have benefited greatly from the advice and support of other members of the Church Project. We especially acknowledge the contributions of Santiago Pericas-Geersten and and Glenn Holloway to early versions of the CIL compiler. We also thank the anonymous TIC workshop referees for their helpful feedback.

References

[Age95] 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.

[AT00] T. Amtoft and F. Turbak. Faithful translations between polyvariant flows and polymorphic types. In ESOP '00 [ESOP00], pp. 26–40.

[Ban97] A. Banerjee. A modular, polyvariant, and type-based closure analysis. In ICFP '97 [ICFP97].

[BKR98] N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In ICFP '98 [ICFP98].

[Boe93] H.-J. Boehm. Space efficient conservative garbage collection. In *Proc. ACM SIGPLAN '93 Conf. Prog. Lang. Design & Impl.*, pp. 197–206, 1993.

[CJW00] H. Cejtin, S. Jagannathan, and S. Weeks. Flow-directed closure conversion for typed languages. In ESOP '00 [ESOP00], pp. 56–71.

[CWM98] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type erasure semantics. In ICFP '98 [ICFP98], pp. 301–312.

[DMTW97] A. Dimock, R. Muller, F. Turbak, and J. B. Wells. Strongly typed flow-directed representation transformations. In ICFP '97 [ICFP97], pp. 11–24.

[ESOP00] *Programming Languages & Systems, 9th European Symp. Programming*, vol. 1782 of *LNCS*. Springer-Verlag, 2000.

[FKR+99] R. Fitzgerald, T. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: An optimizing compiler for Java. Technical Report 99-33, Microsoft Research, 1999.

[Geo97] L. George. MLRISC: Customizable and reusable code generators. Technical report, Bell Labs, 1997.

[Hen92] F. Henglein. Simple closure analysis. Technical Report D-193, DIKU, Mar. 1992.

[HH98] J. Hannan and P. Hicks. Higher-order uncurrying. In POPL '98 [POPL98], pp. 1–11.

[ICFP97] *Proc. 1997 Int'l Conf. Functional Programming*. ACM Press, 1997.

[ICFP98] *Proc. 1998 Int'l Conf. Functional Programming*. ACM Press, 1998.

[JS98] S. L. P. Jones and A. L. M. Santos. A transformation-based optimiser for Haskell. *Sci. Comput. Programming*, 32(1–3):3–47, Sept. 1998.

[JW95] 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.

[JWW97] 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.

[KR95] F. Kamareddine and A. Ros. A λ-calculus la de Bruijn with explicit substitution. In *7th Int'l Symp. Prog. Lang.: Implem., Logics & Programs, PLILP '95*, vol. 982 of *LNCS*, pp. 45–62. Springer-Verlag, 1995.

[KW99] A. J. Kfoury and J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In *Conf. Rec. POPL '99: 26th ACM Symp. Princ. of Prog. Langs.*, pp. 161–174, 1999.

[Mau00] L. Mauborgne. Improving the representation of infinite trees to deal with sets of trees. In ESOP '00 [ESOP00], pp. 275–289.

[Mit96] J. C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[MMH96] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Conf. Rec. POPL '96: 23rd ACM Symp. Princ. of Prog. Langs.*, 1996.

[Mor95] G. Morrisett. *Compiling with Types*. Ph.D. thesis, Carnegie Mellon University, 1995.

[MWCG99] 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.

[Nec97] G. C. Necula. Proof-carrying code. In POPL '97 [POPL97], pp. 106–119.

[NN97] F. Nielson and H. R. Nielson. Infinitary control flow analysis: A collecting semantics for closure analysis. In POPL '97 [POPL97], pp. 332–345.

[Pie91] B. C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, Feb. 1991.

[PJ96] S. L. Peyton Jones. Compiling Haskell by program transformation: A report from the trenches. In *Proc. European Symp. on Programming*, 1996.

[PJM97] S. L. Peyton Jones and E. Meijer. Henk: A typed intermediate language. In TIC '97 [TIC97].

[POPL97] *Conf. Rec. POPL '97: 24th ACM Symp. Princ. of Prog. Langs.*, 1997.

[POPL98] *Conf. Rec. POPL '98: 25th ACM Symp. Princ. of Prog. Langs.*, 1998.

[PP0X] J. Palsberg and C. Pavlopoulou. From polyvariant flow information to intersection and union types. *J. Funct. Programming*, 200X. To appear.

[Shi91] O. Shivers. *Control Flow Analysis of Higher Order Languages.* Ph.D. thesis, Carnegie Mellon University, 1991.

[Sis99] J. M. Siskind. Flow-directed lightweight closure conversion. Technical Report 99-190R, NEC Research Institute, Inc., Dec. 1999.

[SLM98] Z. Shao, C. League, and S. Monnier. Implementing typed intermediate languages. In ICFP '98 [ICFP98], pp. 313–323.

[SW97] P. Steckler and M. Wand. Lightweight closure conversion. *ACM Trans. on Prog. Langs. & Systs.*, 19(1):48–86, Jan. 1997.

[Tar96] D. Tarditi. *Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML.* Ph.D. thesis, Carnegie Mellon University, Dec. 1996.

[TIC97] *Proc. First Int'l Workshop on Types in Compilation*, June 1997. The printed TIC '97 proceedings is Boston Coll. Comp. Sci. Dept. Tech. Rep. BCCS-97-03. The individual papers are available at http://www.cs.bc.edu/~muller/TIC97/ or http://oak.bc.edu/~muller/TIC97/.

[TMC+96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM SIGPLAN '96 Conf. Prog. Lang. Design & Impl.*, 1996.

[TO98] 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.

[Tol94] A. Tolmach. Tag-free garbage collection using explict type parameters. In *Proc. 1994 ACM Conf. LISP Funct. Program.*, pp. 1–11, 1994.

[WDMT97] 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 [WDMT0X].

[WDMT0X] 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 [WDMT97].

[WS94] 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.