Research Summary

Franklyn Turbak
Department of Computer Science
Wellesley College

August 1, 2001

My main research area is the design, analysis, and implementation of expressive programming languages. I also work on pedagogical aspects and applications of programming and of programming languages. This document summarizes my research and publications in these areas.

Much of the research described here was undertaken as part of the Church Project[1], a group of programming language researchers investigating applications of formal systems in programming language design, analysis, and implementation. I was a co-founder of the Church Project in September, 1995, and have been working closely with its members ever since. My research with the Church Project is supported with funding from an NSF Experimental Software Systems grant, an individual grant that is part of a larger collaborative grant with Boston University, Boston College, and Stevens Institute of Technology.

Some of my research was conducted with colleagues and students at my home institution, Wellesley College. I believe it is important for undergraduate computer science students to experience the field outside their coursework, so I encourage my students to become involved in research and independent study projects. Although the technical prerequisites for programming language research set a fairly high bar for undergraduates, I have managed to involve undergraduates in several projects related to programming languages (see Sections 4.3 and 6).

This summary is organized into sections according to research themes. In each section, I present the context of my research, describe my publications and contributions, and, where relevant, describe future research plans. The first three sections discuss my research areas related to the Church Project: flow types (Section 1), a compiler based on flow types (Section 2), and a module calculus (Section 3). Section 4 describes research projects that are related to my Ph.D. research on tree manipulation in functional programming. Sections 5 (robotics), 6 (visual programming) and 7 (information sharing) describe secondary research areas. My pedagogical publications are summarized in Section 8.

Bibliographic citations in bold font (such as [**TW01**]) indicate papers I have authored or co-authored that are included in my publication binder. Some of these papers are drafts of work still in progress, but whose completion is expected soon. When they are available, completed papers will be posted at `http://cs.wellesley.edu/~fturbak/pubs`.

[1]The Church Project is named in honor of logician Alonzo Church, inventor of the λ-calculus, a theoretical foundation for programming languages. The home page of the Church Project is `http://types.bu.edu`.

1 Theory of Flow Types

Most of my research in the Church Project has focused on (1) intersection and union types, (2) how these types can be used to encode information about the flow of values from points of creation to points of use within a program, and (3) how the resulting *flow types* can be used to improve compilation. In Section 1.1, I summarize some background to this research (type systems and flow analysis) that is necessary for motivating and describing my work. In Sections 1.2–1.4, I describe those research projects that I have undertaken in this area that have a theoretical flavor. On the practical side, my contributions to the Church Project compiler are detailed in Section 2.

1.1 Background

A *static type system* is one formalism for reasoning about properties of phrases in a program that can be determined without executing the program. Intuitively, a type describes (at an abstract level) the value and effects of evaluating a phrase. Types are just one kind of program analysis used within a compiler. There has been a concerted effort to use types as a "hook" off which to hang the results of various other kinds of program analyses not traditionally associated with types, such as effect analysis, strictness analysis, and usage analyses.

One analysis that is particularly important in reasoning about function-oriented and object-oriented languages is *flow analysis*. This analysis conservatively approximates how values (particularly functions) created at run-time flow from their sources (points of definition) to their sinks (points of use). While type systems and flow analyses seem different on the surface, both describe the "plumbing" of a program (i.e., how values flow within a program), and recent work has uncovered deep connections between them. Palsberg & O'Keefe [PO95] demonstrated an equivalence between determining flow safety in a simple flow analysis known as 0-CFA and typability in a system with recursive types and subtyping [AC93]. Heintze showed equivalences between four restrictions of 0-CFA and four type systems parameterized by subtyping and recursive types [Hei95]. A 0-CFA analysis is imprecise because it merges flow information for all calls to a function—a feature characteristic of so-called *monovariant* flow analyses. Greater precision can be obtained via *polyvariant* analyses, in which functions can be analyzed in multiple abstract contexts. Examples of polyvariant analyses include call-string based approaches, such as k-CFA [Shi91, JW95, NN97], polymorphic splitting [WJ98], type-directed flow analysis [JWW97], and argument-based polyvariance [Sch95, Age95].

The work of Palsberg & O'Keefe and Heintze raises two intriguing questions:

- *Is it possible to encode polyvariant flow analyses in a type system?*
- *Can encoding flow information in a type system improve type-directed compilation?*

Much of my research with the Church Project has focused on the theoretical and practical aspects of these questions.

1.2 λ^{CIL}: A Calculus with Flow Types

To address the above questions, I worked with Joe Wells, Allyn Dimock, and Bob Muller to integrate polyvariant flow information and polymorphic type information, yielding what we call *flow types*[2]. We developed a typed λ-calculus based on flow types, called λ^{CIL}, whose key features are:[3]

1. *Flow labels*: Source and sink terms are annotated with identifying flow labels. Each type constructor is annotated with a set of source labels approximating the source terms producing the values of that type and a set of sink labels approximating the sink terms consuming the values of that type.

2. *Intersection and union types*: The type system includes intersection and union types. An intersection type is a finitary analogue of an infinitary universal type that list the concrete types at which a polymorphic function is instantiated. Dually, a union type is a finitary analogue of an infinitary existential type that list the concrete types via which an abstract type is represented.

Intuitively, these types can encode polyvariant flow analyses as follows: intersection types model the multiple analyses for a given abstract closure, while union types model the merging of abstract values where flow paths join. From a theoretical point of view, flow labels are unnecessary, since any flow information representable with label-annotated types can instead be represented with intersection and union types. However, from an engineering point of view, flow labels are important for naming purposes and because type constructors annotated with sets of flow labels are a compact encoding of common patterns of union and intersection types.

A λ^{CIL} source term defining a value that flows to multiple sinks is represented as a *virtual tuple*, a term of intersection type that lists one copy of the definition for each conjunct of the intersection type. In λ^{CIL}, a usage site to which multiple values flow is represented as a *virtual case expression* that dispatches to copies of the sink based on a virtual injection tag. See Figure 1 for examples of flow types and virtual forms in λ^{CIL}. A more detailed explanation of λ^{CIL} containing more examples can be found in our journal paper **[WDMT0X]**, which expands on material presented in an earlier conference paper [WDMT97] and workshop paper [TDMW97]. The journal paper also shows important formal properties of λ^{CIL}: reduction in λ^{CIL} is confluent (modulo a notion of type erasure) and satisfies a subject reduction property. The latter property is used to show type soundness. λ^{CIL} also satisfies a standardization property that is proven in separate work [MW00] in which I was not a participant.

Our work on using intersection and union types to encode polyvariant flow information was in large part inspired by Banerjee's work ([Ban97]) on encoding a simple polyvariant flow analysis using rank-2 intersection types. However, our system is much more general because (1) it does not put a rank restriction on the intersection types and (2) it uses union types, which makes it possible to encode flow analyses that join distinct flow values. Although much work on intersection and union types preceded ours (e.g., [CDCV80, CDCV81, BCDC83, RDRV84, Pie91, vB93, BDCd95, Jim95, Rey96]), our work continues to play a major role in popularizing them within the programming languages community.

[2]Heintze coined the term "control flow type" in [Hei95] to describe arrow types annotated with source labels. Our flow type notion additionally includes sink labels, intersection and union types, introduction and elimination forms for these types, and explicit coercions between types.

[3]"CIL" stands for Church Intermediate Language.

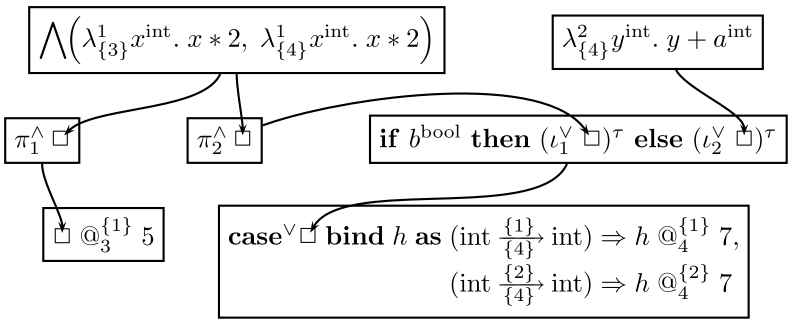
$$\begin{aligned}&\textbf{let } f = (\lambda x.x * 2)\\ &\textbf{in let } g = (\lambda y.y + a)\\ &\quad \textbf{in } \times(f \mathbin{@} 5, (\textbf{if } b \textbf{ then } f \textbf{ else } g) \mathbin{@} 7)\end{aligned}$$

(a) This is an untyped λ^{CIL} term M. λ introduces a function abstraction, @ indicates a function application, and $\times(M_1, M_2)$ introduces a tuple of M_1 and M_2.

$$\begin{aligned}&\textbf{let } f^{\sigma_1} = \lambda^1_{\{3,4\}} x^{\mathrm{int}}.\ x * 2\\ &\textbf{in }\ \textbf{let } g^{\sigma_2} = \lambda^2_{\{4\}} y^{\mathrm{int}}.\ y + a^{\mathrm{int}}\\ &\qquad \textbf{in } \times\ \Big(\textbf{coerce}\,(\sigma_1, \sigma_3)\, f \mathbin{@}^{\{1\}}_3 5,\\ &\qquad\qquad\quad (\textbf{if } b^{\mathrm{bool}}\ \textbf{ then coerce}\,(\sigma_1, \sigma_4)\, f \textbf{ else } \textbf{coerce}\,(\sigma_2, \sigma_4)\, g)\ \mathbin{@}^{\{1,2\}}_4 7\Big)\end{aligned}$$

$$\text{where } \sigma_1 = \mathrm{int} \xrightarrow[\{3,4\}]{\{1\}} \mathrm{int}, \sigma_2 = \mathrm{int} \xrightarrow[\{4\}]{\{2\}} \mathrm{int}, \sigma_3 = \mathrm{int} \xrightarrow[\{3\}]{\{1\}} \mathrm{int},\ \text{and } \sigma_4 = \mathrm{int} \xrightarrow[\{4\}]{\{1,2\}} \mathrm{int}$$

(b) This is a typed version of M from (a) illustrating flow labels on values of function type. The notation λ^ℓ_ψ denotes a function labeled ℓ that may flow to the sinks whose labels are in the set ψ, while $@^\phi_k$ denotes a sink labeled k to which functions whose labels are in the set ϕ may flow. The arrow type $\tau_1 \xrightarrow[\psi]{\phi} \tau_2$ is the type of functions that flow from abstractions with labels in ϕ to application sites with labels in ψ and that map type τ_1 values to type τ_2 values. The notation $\textbf{coerce}\,(\tau_1, \tau_2)\, N$ performs a type coercion from τ_1 to τ_2 by adding source labels or removing sink labels.



$\bigwedge\Big(\lambda^1_{\{3\}} x^{\mathrm{int}}.\ x * 2,\ \lambda^1_{\{4\}} x^{\mathrm{int}}.\ x * 2\Big)$
$\lambda^2_{\{4\}} y^{\mathrm{int}}.\ y + a^{\mathrm{int}}$
$\pi^\wedge_1 \square$
$\pi^\wedge_2 \square$
$\textbf{if } b^{\mathrm{bool}} \textbf{ then } (\iota^\vee_1 \square)^\tau \textbf{ else } (\iota^\vee_2 \square)^\tau$
$\square \mathbin{@}^{\{1\}}_3 5$
$\textbf{case}^\vee \square\ \textbf{bind } h \textbf{ as } (\mathrm{int} \xrightarrow[\{4\}]{\{1\}} \mathrm{int}) \Rightarrow h \mathbin{@}^{\{1\}}_4 7,$
$(\mathrm{int} \xrightarrow[\{4\}]{\{2\}} \mathrm{int}) \Rightarrow h \mathbin{@}^{\{2\}}_4 7$

$$\text{where } \tau = \bigvee\Big[\mathrm{int} \xrightarrow[\{4\}]{\{1\}} \mathrm{int}, \mathrm{int} \xrightarrow[\{4\}]{\{2\}} \mathrm{int}\Big]$$

(c) This is a typed version of M annotated with intersection and union types as well as labels. It is drawn in a graphical format to highlight its flow-based nature. The term $\bigwedge\Big(\lambda^1_{\{3\}} x^{\mathrm{int}}.\ x * 2,\ \lambda^1_{\{4\}} x^{\mathrm{int}}.\ x * 2\Big)$ is a virtual tuple with intersection type $\bigwedge\Big[\mathrm{int} \xrightarrow[\{3\}]{\{1\}} \mathrm{int}, \mathrm{int} \xrightarrow[\{4\}]{\{1\}} \mathrm{int}\Big]$. Components of a virtual tuple are extracted via the virtual tuple projections $\pi^\wedge_i$. Values of union type $\bigvee\Big[\mathrm{int} \xrightarrow[\{4\}]{\{1\}} \mathrm{int}, \mathrm{int} \xrightarrow[\{4\}]{\{2\}} \mathrm{int}\Big]$ are created via virtual injections of the form $\iota^\vee_i$ and decomposed via virtual case expressions of the form $\textbf{case}^\vee N \textbf{ bind } x \textbf{ in } \tau_1 \Rightarrow N_1, \ldots, \tau_m \Rightarrow N_m$ that dispatch to a clause N_j (in which x has type τ_j) based on the type of the discriminant N.

Figure 1: Flow type example.

In this work, my key contribution was helping to work out the details of how intersection and union types can be used in conjunction with flow labels to encode polyvariant flow analyses. In particular, Joe Wells and I were the first to recognize the importance of union types for encoding polyvariant analyses, and we led the effort to include them in our type system and compiler framework, which previously had focused only on intersection types. I also helped to articulate the principles underlying the design of flow types and to motivate them in terms of practical compiler optimizations (see Section 2).

1.3 Faithful Translations between Flows and Types

In our work on λ^{CIL}, we showed examples of how polyvariant flow analyses could be encoded in λ^{CIL} and conjectured that a wide range of polyvariant analyses could be encoded. However, we did not show a formal correspondence between polyvariant flow analyses and our type system. Inspired in part by our work on λ^{CIL}, Palsberg and Pavlopoulou (henceforth P&P) were the first to formalize this correspondence by demonstrating an equivalence between a class of flow analyses supporting argument based-polyvariance and a type system with intersection and union types [PP98, PP01]. In this section , I describe how I helped to improve the P&P flow/type correspondence.

If type and flow systems encode similar information, it is natural to expect that translations between the two should be *faithful*, in the sense that "round-trip" translations from flow analyses to type derivations and back (or from type derivations to flow analyses and back) should not lose precision. I discovered that P&P's translations were not faithful: translations from flows to types and back often lose precision, and translations from types to flows and back can introduce recursive types that were not present in the original type derivation.

I worked closely with Torben Amtoft to develop the first faithful translations between a broad class of polyvariant flow analyses and a type system with polymorphism in the form of intersection and union types; these translations are detailed in **[AT0X]**. Our translations are faithful in the sense that a round-trip translation acts as the identity for canonical types/flows, and otherwise canonicalizes. We achieve this result by adapting the translations of P&P to use a modified version of the flow analysis framework of Nielson and Nielson (henceforth N&N) [NN97]. Label annotations play a key role in the faithfulness of our translations: we (1) annotate flow values to indicate the sinks to which they flow, and (2) annotate union and intersection types with component labels that serve as witnesses for existential quantifiers that appear in the definition of subtyping. These annotations can be justified purely in terms of the type or flow system, independent of the flow/type correspondence.

Additionally, our framework solves several open problems posed by P&P:

1. *Unifying P&P and N&N*: Whereas P&P's flow specification can readily handle only argument-based polyvariance, N&N's flow specification can also express call-string based polyvariance. Since our flow framework is built on top of N&N's, our translations give the first type systems corresponding to k-CFA analysis where $k \geq 1$.

2. *Subject evaluation for flows*: We inherit from N&N's flow logic the property that flow information valid before an evaluation step is still valid afterwards. In contrast, P&P's flow system does not have this property.

3. *Letting "flows have their way"*: P&P discuss mismatches between flow and type systems that imply the need to choose one perspective over the other when designing a translation between

the two systems. In their translations, P&P choose to always let types "have their way"; for example they require analyses to be finitary and to analyze all closure bodies, even though they may be dead code. In contrast, our design also lets flows "have their way", in that our type system does not require all subexpressions to be analyzed.

The type system I developed in conjunction with Torben Amtoft is similar to the one for λ^{CIL}, but differs in one critical respect: it uses *deep subtyping* on compound types such as arrow types, whereas λ^{CIL} requires *shallow subtyping* on these types. In deep subtyping, the components of types related by the subtype relation are themselves related by the subtype relation. In shallow subtyping, the components of types related by the subtype relation must be related by type equality. For this reason, our work cannot be used in its current state for the practical purpose of encoding flow analyses in λ^{CIL}. Formally characterizing the difference between deep and shallow subtyping systems, and potentially developing translations between them, is an area for future work.

1.4 Complexity of Type Inference for Finite-Rank Intersection Types

An important question for any type system is whether it is possible to infer the type of a term without any type annotations. It has long been known that type inference for terms of the standard λ-calculus in a type system with intersection and function types is undecidable. However, Kfoury and Wells showed that type inference is computable for such terms in *finite-rank* intersection type systems [KW99]. The finite-rank restriction on intersection types bounds how deeply the $\wedge$ can appear in type expressions, counting nesting in the left arguments of the $\rightarrow$ type constructor.

Given a type inference algorithm, one typically wants to know its worst-case computational time complexity as a function of the size of the term on which type inference is performed. The complexity of type inference for intersection type systems with a finite upper bound k on the rank of intersection types (System $\boldsymbol{\mathcal{I}}_k$) was an open problem. In collaboration with Assaf Kfoury and Joe Wells, I showed an upper bound on System $\boldsymbol{\mathcal{I}}_k$ type inference by adapting a strategy used in analyzing ML typability. An exponential upper bound for ML typability can be shown by a "monomorphization" process that (1) translates a possibly polymorphic ML program into a (potentially exponentially larger) program in a monomorphic language and (2) using polynomial-time type inference in the monomorphic language to determine if the resulting program is typable. Inspired by this idea, I designed a type inference algorithm for System $\boldsymbol{\mathcal{I}}_k$ that (1) tranforms a given term into a (potentially much larger) term in a more constrained intersection type system (call it S), (2) performs type inference in S, and (3) "lifts" the result of type inference from S back to System $\boldsymbol{\mathcal{I}}_k$. This algorithm succeeds if and only if the original term is typable in System $\boldsymbol{\mathcal{I}}_k$; otherwise it indicates that the term is not typable. There are a number of technical hurdles to overcome in this approach, but everything works out in the end.

The approach sketched above is not an efficient way to perform type inference in System $\boldsymbol{\mathcal{I}}_k$, but is a convenient way to show an upper bound on type inference. Using this approach, I showed that the type inference problem for a System $\boldsymbol{\mathcal{I}}_k$ program of size n has the astronomical Kalmar-elementary upper bound $O(\mathbf{K}(k-1,n))$, where $\mathbf{K}(h,x) = 2^{2^{\cdot^{\cdot^{\cdot^{2^x}}}}}$ (a stack of h 2s). This upper bound is complemented by a Kalmar-elementary lower bound on System $\boldsymbol{\mathcal{I}}_k$ type inference due to Harry Mairson, yielding a complete analysis of the time complexity of System $\boldsymbol{\mathcal{I}}_k$ type inference. This work is reported in **[KMTW99]**, which also relates the complexity of typability in System $\boldsymbol{\mathcal{I}}_k$ to the question of expressiveness (i.e., do two typable terms have the same normal form?).

Although the worst-case time complexity of finite rank intersection type inference is clearly intractable, it is unknown what, if any, practical implications this result has. Theoretical time complexity bounds are not always important in practice. For instance, it is well-known that type inference in the ML programming language takes exponential time in the size of a program in the worst case, but the worst cases are pathological ones that are rarely encountered in practice. Indeed, most ML programmers experience ML type inference as being efficient. I plan to investigate whether this is also true for finite-rank intersection types.

The fact that finite-rank intersection types have a *principal typing* property [Jim96, KW99], and therefore provide a modular analysis of a program fragment, makes them a promising candidate for the basis of a module type system that supports link-time compilation—another interesting avenue for exploration.

2 Compiling with Flow Types

In addition to helping to develop the theory of flow types, I played a major role in the design and implementation of an experimental compiler based on this theory. The goal of this project is to investigate the benefits and drawbacks of using flow types in a compiler, and to explore engineering issues involved in implementing flow types.

Our compiler is organized around a *typed intermediate language*. The past decade has witnessed an explosion of research focusing on how to propagate static type information through compiler phases via such languages, and how to utilize this information to generate more efficient code for function-oriented and object-oriented programming languages (e.g., [Mor95, TMC+96, PJ96, PJM97, Sha97, BKR98, TO98, FKR+99, CJW00]). Types that are carried all the way through to the generated code can guide run-time operations such as garbage collection. Moreover, the ability to certify the type correctness of a program after every compilation stage allows the safety benefits of types to be enjoyed within the compiler (where it serves as a sanity check on the compiler implementation) and even at the level of *typed assembly language* [MWCG99] (where types can serve as a "proof" of important safety properties).

2.1 Flow-Based Customization

The key novel feature of our compiler is its use of flow types to choose customized data representations for values based on the context of their use. Traditionally, many compilers have used a *uniform representation assumption* in which all values made with the same constructor have the same representation. For example, every list node might be represented as a pair of machine words: one to hold a list element, and one to hold a pointer to the tail of the list. If type information is available, it can be used to choose more efficient representations. For instance, nodes for a list of double-precision floating point numbers might use three machines words: two to hold the (unboxed) number, and one for the tail pointer.

Flow types allow more fine-grained data customizations based on how a value is used. Consider function representations: an open function (one with free variables) is typically represented as a closure structure that combines a function pointer with the free variable values, but a closed function (one without free variables) can ideally be represented solely as a code pointer. However, this efficient representation can be used only if the closed function shares call sites only with other customized closed functions—information that can be determined from flow types. In particular,

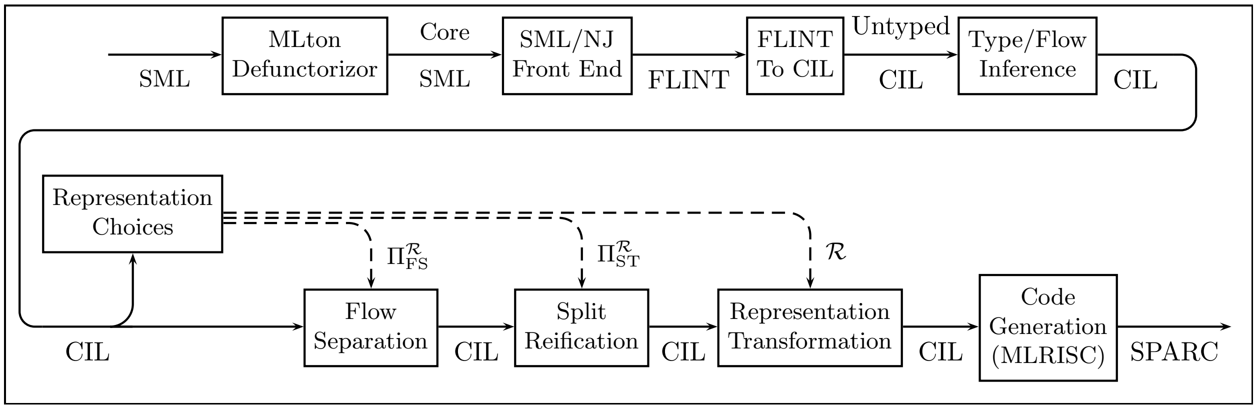


Figure 2: Architecture of the CIL ML compiler. A type checker (not shown) is run on the result of every stage that produces CIL code to certify the type and flow soundness of the stage.

the customization cannot be used for a closed function that shares a call site with an open function. This is an example of what we call *representation pollution*: a situation where a value is constrained to have a less efficient representation than it otherwise would because it is used in the same context as another value that must have a less efficient representation.

Our compiler framework can generate customized data representations even in the presence of representation pollution. Pollution is removed by (1) generating multiple and mutually incompatible customized representations of value source and sink forms and (2) introducing sufficient "plumbing" to ensure that only compatible representations flow together. This is accomplished by converting virtual products and sums (i.e., terms of intersection and union type), which are compile-time annotations implying no run-time execution costs, to real products and sums, which do have a run-time execution cost. See [**DMTW97**] and [**DWM$^+$01a**] for examples and details.

Our approach is similar to the function customization strategies used by type-based compilers that remove polymorphic higher-order functions via monomorphization and defunctionalization [TO98, CJW00]. 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. As in the CIL compiler, these compilers use flow analysis to customize function representations for particular application sites, but the analysis is not integrated into their type systems. After monomorphization and defunctionalization, a flow analysis can be implicit in the types of a defunctionalized program. Advantages of explicit flow types include (1) the availability of explict flow information for optimizations, (2) the ability to distinguish virtual structures manipulated by the compiler from real structures in the generate code, and (3) a formal framework for preserving explicit flow information from one compiler stage to the next.

2.2 Compiler Architecture

We have implemented a whole-program compiler for core Standard ML using CIL, a typed intermediate language that is based on our λ^{CIL} calculus. For implementing features of core Standard ML, CIL extends the purely functional λ^{CIL} with primitive datatypes, references, arrays, and exceptions. These extensions are described in [**DWM$^+$01b**].

The architecture of the CIL ML compiler, first proposed in [**DMTW97**], is shown in Figure 2. I am the original architect of this compiler in the sense that it reflects a modular organization that I designed in the early stages of the project. I improved upon an earlier monolithic design by separating the details of how flow information is encoded in the flow types from the details of how data representations are transformed. In particular, I invented the initial notion of *flow separation*, which turned out to be essential for the development of the flow-based customization aspects of the compiler. Although my initial design has been refined in collaboration with my colleagues, many of the stages I proposed have remained essentially intact in our current compiler implementation.

Below is a brief summary of the stages depicted in the figure:

- The MLton source-to-source defunctorizer [CJW00] is used as a prepass to convert multi-module SML programs into a single core SML program.
- A modified front end of the SML/NJ 110.03 compiler produces FLINT code [Sha97], which is translated to untyped CIL.
- The Type Inference/Flow Analysis stage takes an untyped CIL term (plus some 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. Programs passing the ML type checker in the front end are sufficiently constrained that the potentially high theoretical costs of type inference involving intersections (see Section 1.4) do not come into play. This stage is parameterized over a choice of flow analyses. We currently support five different flow analyses, which vary with respect to the precision of the approximation. These are discussed in [**DWM$^+$01b**].
- The Representation Choices module uses a flow type information and a function representation strategy to select the representation for functions. Thus far, we have experimented with seven different function representation strategies [**DWM$^+$01a**]. This module produces information (the dotted lines in the figure) that controls the behavior of several program transformation stages.
- The Flow Separation stage accepts as input (1) a typed program and (2) a flow path partitioning function ($\Pi^{\mathcal{R}}_{\mathrm{FS}}$) supplied by Representation Choices that specifies which flow paths are compatible. For flow paths that are not compatible, the Flow Separation phase introduces whatever coercions and virtual forms (i.e., virtual variant injections, virtual case expressions, virtual tuples, or virtual tuple projections) are required to ensure that the result of the later Representation Transformation stage will be well-typed.
- The Split Reification stage accepts as input a typed term and a flow path partitioning function ($\Pi^{\mathcal{R}}_{\mathrm{ST}}$) supplied by Representation Choices. This phase reifies whatever virtual forms are required to remove representation pollution. We refer to the reification process as *splitting* because it causes the code generator to generate multiple copies of a term in situations where only one copy would have been generated without reification.
- The Representation Transformation stage accepts as input a typed term and a representation map ($\mathcal{R}$) provided by Representation Choices. It walks the term and installs the function representations specified by the map. The Representation Transformation stage performs

the actual work of changing the code for specialized representations. In contrast, the Flow Separation stage only introduces virtual forms, and the Split Reification stage only reifies virtual forms.

- The Code Generation stage transforms typed CIL programs into assembly code for the SPARC processor. It does not currently insert any type annotations into 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].

2.3 Compiler Implementation

During the past five years, the CIL compiler has evolved from a paper design to a working compiler (implemented in Standard ML) that is able to compile medium sized (up to 3000 source lines) whole Standard ML programs to executable SPARC machine code. The key implementers of this compiler have been Allyn Dimock and Ian Westmacott, but I have made significant contributions to the design and implementation effort.

Below is a list of my contributions to the compiler implementation effort:

- I designed and implemented the Flow Separation stage.
- I worked with Bob Muller to extend the Representation Choices and Representation Transformation stages to handle defunctionalization as a function representation strategy. Defunctionalization was not in our original design ([**DMTW97**]), but is important for comparing our work with compilers based on defunctionalization (such as [TO98, CJW00]).
- I worked with Allyn Dimock and Bob Muller on the design and implementation of flow-based *known-function optimization.* This optimization does not count a variable in a function application position as a free variable if its run-time function value is known at compile time. This is a particularly important optimization in our function customization framework, because it gives rise to more closed functions, and thus more customization opportunities.
- I developed techniques for handling recursive values and recursive types in CIL. Handling recursive types is especially challenging; it turns out that the output of some of our compiler stages can contain recursive types even when their inputs do not. I designed and implemented a modular technique that allows the compiler to introduce new recursive types without changing the existing specification for the Representation Transformation stage. This technique led to my work with Joe Wells on manipulating cyclic data structures in declarative languages [**TW01**] (see Section 4.2).
- In the type checker, I made significant extensions to early work by Santiago Pericas, including an efficient implementation of subtype checking on recursive flow types.
- I implemented a collection of simple partial-evaluation style optimizations.
- Together with Allyn Dimock, I designed and implemented the first version of the CIL abstract syntax and associated utilities, including a property system.
- I designed the exception handling mechanism for CIL.

- I sketched a new design for the Split Reification stage (this has not been implemented).

Our compiler implementation efforts thus far have focused on the function customization framework. There are numerous standard compiler optimizations that we have not yet implemented, so our compiler is still a long way from being a production quality optimizing compiler. Nevertheless, the current CIL compiler represents a significant technical achievement for a number of reasons:

- Our framework has required the development of many new algorithms that have a significantly different flavor from those used in other compilers based on typed intermediate languages: e.g., encoding flow analyses in our type system, performing flow separation, propagating splits in the Split Reification phase, handling "out-of-nowhere" recursive types in the Representation Transformation phase, flow-based inlining of multiple functions at a single call site, hash-consing of recursive types, etc. Developing these new algorithms has been intellectually invigorating, but time consuming.

- As anyone who has actually worked on the implementation of a TIL-based compiler has discovered, significant effort must be spent to preserve the type information across compiler transformations. Type information is very useful in later stages of a compiler, but getting it there can be challenging. The difficulty is a function of the kind and amount of analysis information that is embedded in the type system. In the case of flow types, it is necessary to guarantee that the flow analysis embedded in the type annotations is still valid after a transformation – something that is non-trivial in general.

- The duplicating term representation used in the CIL compiler requires the manipulation of so-called *parallel terms* (see [**WDMT0X**]), which are difficult to reason about and manipulate. For example, consider the following flow-annotated term:

$$\begin{array}{l} \mathbf{let}\ g^{\tau_1} = \bigwedge\Big((\lambda^1_{\{7\}} f^{\tau_2}.\ f\ @^{\{3\}}_5\ 10), (\lambda^2_{\{8\}} f^{\tau_3}.\ f\ @^{\{4\}}_6\ 10)\Big) \\ \ \mathbf{in}\ \times\Big((g\ @^{\{1\}}_7\ (\lambda^3_{\{5\}} x^{\mathrm{int}}.\ x+1)), (g\ @^{\{2\}}_8\ (\lambda^4_{\{6\}} y^{\mathrm{int}}.\ y * y))\Big) \end{array}$$

$$\begin{array}{lrcl} \text{where} & \tau_1 & = & \bigwedge\Big[\tau_2 \xrightarrow[\{7\}]{\{1\}} \mathrm{int}, \tau_3 \xrightarrow[\{8\}]{\{2\}} \mathrm{int}\Big] \\ & \tau_2 & = & \mathrm{int} \xrightarrow[\{5\}]{\{3\}} \mathrm{int} \\ & \tau_3 & = & \mathrm{int} \xrightarrow[\{6\}]{\{4\}} \mathrm{int} \end{array}$$

  At first glance, it might seem that the f bound by $\lambda^1_{\{7\}}$ is a known function whose value must be $\lambda^3_{\{5\}}$, and similarly that the f bound by $\lambda^2_{\{8\}}$ is a known function whose value must be $\lambda^4_{\{6\}}$. But since $\lambda^1_{\{7\}}$ and $\lambda^2_{\{8\}}$ are combined via $\bigwedge$ in a virtual tuple, they are parallel terms, and there are *two* functions ($\lambda^3_{\{5\}}$ and $\lambda^4_{\{6\}}$) that can reach the type-erased application site f @ 10. So f is *not* a known function at sites $@^{\{3\}}_5$ and $@^{\{4\}}_6$.

- Because our flow type framework is so different from other frameworks, we cannot just include off-the-shelf optimization technology in our framework. Adapting existing optimizations to our framework requires significant engineering.

2.4 Empirical Results

Our compiler implementation has matured to the point where we have been able to perform some preliminary experiments that involve using the CIL compiler to compile a suite of Standard ML benchmarks. We first studied the compile-time space costs of using CIL [**DWM$^+$01b**]. CIL's listing-based intersection and union types and its duplicating term representations threaten to cause compile-time space explosion at both the type and the term level, but we have not observed such blowups in practice. Our experiments show that space costs can be made tractable by using sufficiently fine-grained flow analyses together with standard hash-consing techniques. A surprising result of our experiments is that they suggest that non-duplicating formulations of intersection and union types would not achieve significantly better space complexity than our duplicating term representation. 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 flow analyses expressing more polyvariance.

To determine the effect of customizations and pollution removal on the dynamic costs of function representations, we have measured the run-time performance of code generated using various function customization strategies [**DWM$^+$01a**]. Because of the lack of optimizations in our compiler, the wall-clock time for executing the generated code is not a particularly meaningful measurement. Instead, we have modified the compiler to emit 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. Our experiments show that the flow-based customization of closed functions can give significant improvements over uniform closure representations when no pollution needs to be removed. However, the jury is still out on the efficacy of using flow types to remove representation pollution. In our experiments, the pollution removal strategies we consider often cost more in overhead than they gain via enabled customizations. However, some strategies that use defunctionalization and flow-based inlining often achieve significant customization benefits via aggressive pollution removal.

My main contribution to both evaluation projects was situating our work in a context that explains the importance of our results. In particular, in [DWM$^+$01a], I led the effort to implement within the CIL compiler several function representation strategies (defunctionalization and Steckler and Wand's selective representation) and the known-function optimization. These allowed us to compare our other strategies to ones already known in the literature. While most of the measurements and interpretation in these papers was the work of Allyn Dimock, I helped (1) to interpret the empirical results and (2) to present the results effectively in a graphical form.

2.5 Future Work

Having invested in the construction of a working flow-typed compiler, we are planning a number of extensions and experiments that should require relatively little additional work:

- We plan to investigate other function optimizations and function representation strategies, such as higher-order uncurrying [HH98] and closure representations that exclude a free variable from an environment if its value is available at all call sites [SW97].
- There is much debate as to whether flow-based inlining of functions is a good idea [WJ98] or whether syntax-based inlining techniques suffice [App92, Tar96]. We believe that the CIL

compiler is an excellent workbench for investigating this question, and plan to undertake such a project in the near future. 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 syntax-based inlining techniques.

- We plan to generalize our known function optimization to a flow-based known-value optimization that works on any type of value. As with function inlining, we plan to compare flow-based and syntax-based algorithms to see whether the flow-based algorithms are sufficiently beneficial to justify the extra work of maintaining flow information.

There are also a number of longer-range projects that will require significantly more work. One is to develop a flow-typed assembly language and investigate opportunities for flow-based optimizations at this level. Another is to investigate whether flow types can be used as the basis for a system that supports separate compilation rather than just whole-program compilation (see Section 3). The principal typing property of finite-rank intersection types discussed in Section 1.4 is important in this regard.

3 Module Calculi

As a first step toward exploring separate compilation and link-time issues in the context of the Church Project, I have been working closely with Elena Machkasova on the development of a call-by-value module calculus that serves as a framework for studying simple meaning-preserving module transformations. Our work is described in **[MT0X]**, a draft of a Boston University technical report that expands and improves upon our earlier conference paper on this topic [MT00].

Our module calculus is stratified into three levels: a term calculus, a core module calculus, and a linking calculus. At each level, we define both a calculus reduction relation and a small-step operational semantics and relate them by a *computational soundness* property: if two terms are equivalent in the calculus, then they have the same observable outcome in the operational semantics. We develop a formal framework for reasoning about a multi-level calculus, which includes a notion of one calculus being embedded in another. This notion allows us to show that calculus-based transformations at one level of the calculus are meaning preserving with respect to another level.

Our modules have both public and private components. We formalize the notion of privacy by identifying modules up to alpha-renaming of hidden (i.e., private) labels. Because of this identification, module linking can be defined without the need to resolve naming conflicts between the hidden labels of two modules. We resolve a thorny interaction between alpha-renaming and module contexts with fixed private labels. In addition to alpha-renaming at the core module level, we also formalize alpha-renaming in namespaces at the term and linking levels of our calculus. This is important, because many properties of our module calculus hold only for alpha-equivalence classes of terms and not for concrete terms.

An important contribution of this work is a novel technique that we have developed for proving computational soundness. The traditional proof technique for computational soundness, due to Plotkin [Plo75], requires that the calculus be *confluent* and that the calculus and operational semantics be related by a *standardization* property. However, because our module calculus is not confluent, the traditional technique for proving computational soundness does not apply. Nevertheless, we developed a proof technique for computational soundness based on a weaker set of properties, which we call *lift* and *project*, and have used this technique to show that our module calculus is computationally sound.

A particularly important domain for module transformations is link-time compilation, where many optimization opportunities are not apparent until two modules are linked together. We present a simple model of link-time compilation and introduce the *weak distributivity property* for a transformation T operating on modules D_1 and D_2 linked by $\oplus$: $T(D_1 \oplus D_2) = T(T(D_1) \oplus T(D_2))$. We argue that this property finds promising candidates for link-time optimizations, and present simple conditions that imply weak distributivity.

We plan to explore several research directions in this work. With respect to computational soundness, we are searching for other non-confluent calculi to which we can apply our proof technique for computational soundness. With respect to our module calculus, we plan to explore a typed version of our (currently untyped) calculus. In particular, we want to experiment with intersection and union types in the context of our module system. Our long-term goal is to investigate which program analyses can be effectively modularized and used to guide link-time optimizations.

4 Tree-Based Programming

A hallmark of functional programming is expressing programs as the composition of functions, many of which produce and consume tree-shaped data structures (where a linked list is an important degenerate case of a tree). I have undertaken several projects related to tree-based manipulations of functional programs. These are described in the following subsections.

4.1 Synchronized Lazy Aggregates

Expressing programs as the modular composition of tree-manipulating functions has many advantages [Hug90]. One *disadvantage* is that programs written in this style can require more space than programs written in a monolithic (i.e., non-modular) style. I designed and implemented a functional language in which modular fragments could be composed to yield programs with the same space complexity as a corresponding monolithic program. In this section, I briefly describe the problem in more detail and discuss my solution.

Consider the following Haskell functions:

```
avg g p a = (sum xs) / (len xs)
  where xs = gen g p a
        sum = foldl (+) 0
        len = foldl inc 0
        inc x y = x + 1

gen g p a = if (p a) then [] else a:(gen g p (g a))
```

The `avg` function first generates the list `xs` of numbers `[a, g a, g (g a), ... ]` until a value is reached for which the predicate `p` is true (this value is not included in the list). Then the `avg` function returns the average of the numbers in this list. In a normal Haskell implementation, this program is *guaranteed* to take space linear in the length of the sequence because laziness allows at most one of the two consumers of `xs` (i.e., `sum` and `len`) to coroutine with its generator. In contrast, the following monolithic recursive version of the averaging program can take constant space:

```
avg2 g p a = loop a 0 0
  where loop x sm ln
          | p x = sm / ln
          | otherwise = loop (g x) (sm + x) (ln + 1)
```

Is it possible to design a language in which modular decompositions like those in `avg` are guaranteed to have the same space complexity as their monolithic recursive counterparts? This is a question I answered in the affirmative in my Ph.D. thesis [Tur94] and summarized in a related paper [**Tur96**]. The key idea is to use laziness and concurrency in conjunction with first-class synchronization tokens I call *synchrons* to modularize strict function call barriers.

The essence of the approach is illustrated for the `avg` program in Figure 3. Figure 3(a) depicts a representation of a computation generated by a call to the monolithic recursive version of `avg`; the dotted lines indicate (strict) function call barriers. In Figure 3(b), the same computation has been decomposed into tall, thin program fragments I call *slivers*. Slivers communicate via so-called *synchronized lazy aggregates*, tree-shaped data that not only contain "regular" values but also contain synchrons that model the function call barriers within a particular sliver. Synchrons from different slivers can be unified (Figure 3(c)) to simulate the function call barrier of the monolithic computation.

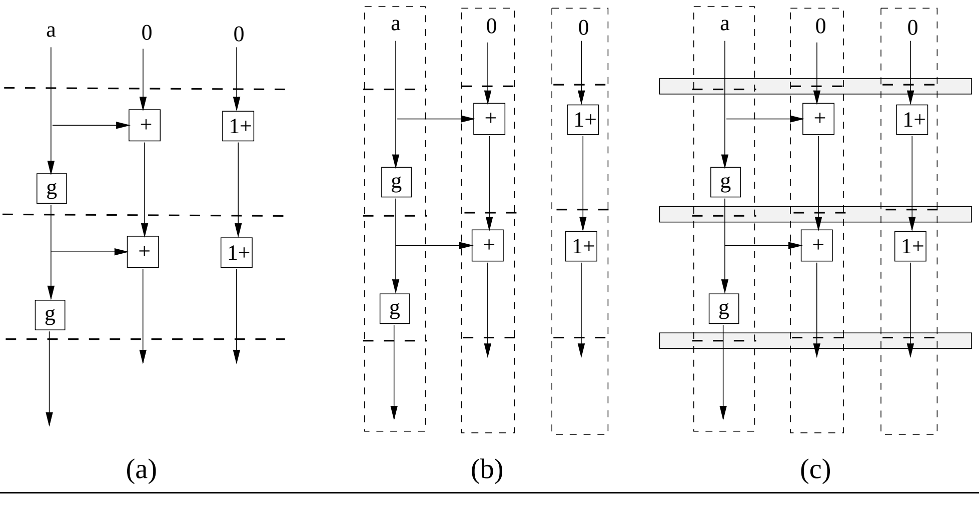


Figure 3: Shared synchrons (c) allow local strict procedure call barriers (b) to simulate the global strict procedure call barriers of monolithic computations (a).

4.2 Regular Tree Manipulation

Cyclic structures — collections of linked nodes containing paths from some nodes back to themselves — are ubiquitous in computer science. From the declarative perspective, cyclic structures are naturally viewed as denoting *regular trees*, those trees that may be infinite but have only a finite number of distinct subtrees. Regular trees can be efficiently represented as finite cyclic structures, but these efficient representations are challenging to create and manipulate in declarative programming languages.

In [**TW01**], Joe Wells and I present some abstractions that we developed to simplify the creation and manipulation of regular trees in both lazy and eager languages:

- We show how to implement the `unfold` (*anamorphism*) operator so as to create cyclic structures when the result is an infinite regular tree as opposed to merely infinite lazy structures.
- We specify the meaning of a `fold` (*catamorphism*) function for accumulating results over (potentially infinite) trees. We also introduce a novel variant of `fold`, which we call `cycfold`, that can produce useful results for infinite regular trees when used with a strict combining function; the usual `fold` function will always yield an undefined result in this case. In practice, `cycfold` can be calculated on bounded representations of regular trees via an iterative fixed point process. For suitable arguments to `cycfold`, the same results are obtained for any finite graph representating a given regular tree; `cycfold` does not expose any details about the particular finite representative on which `cycfold` is invoked.
- We introduce *cycamores*, an abstract data type for regular trees that is largely insensitive to the features of the programming language in which it is embedded. Modulo details of threading state and certain infelicities involving eagerness and laziness, cycamores (including the `unfold` and `cycfold` operations) can be implemented equally well in an eager language or lazy language. We demonstrate this by presenting implementations of cycamores in both SML and Haskell.

As a concrete example motivating our abstractions, consider the following Haskell definition:

```
alts = 0 : 1 : alts
```

This definition creates a conceptually infinite list `alts` of alternating 0s and 1s. In a lazy language like Haskell, this "infinite" list is implemented efficiently as a cyclic arrangement of two list nodes. However, this efficient representation is not preserved by transformations. For instance, the expression `map (\ x -> x * 2) alts` denotes an alternating list of 0s and 2s that could also be implemented using two list nodes, but Haskell creates a representation which, depending on the context in which it is used, could require an arbitrarily large number of list nodes. In contrast, in our system `unfold` can be used to implement a mapping function over a regular cycamore (i.e., a regular tree represented as a cycamore) that returns another regular cycamore. Though it is easy for humans to see that `alts` denotes a regular infinite list whose only elements are $\{0, 1\}$, it is not possible in standard Haskell to write a function that returns the finite set of elements used in an infinite regular list. But such a function can be written in our framework using `cycfold`.

4.3 Deforestation

Much of the inefficiency of modular functional programs arises from the creation and manipulation of intermediate trees. This has prompted research into automatically transforming modular programs into more efficient programs by *fusing* composed functions into a single function (e.g., [Wad90, GLJ93, SF93, LS95, Gil96, HIT96, TM95, OHIT97, Chi99, Chi00, Ném00]). Because program fusion eliminates intermediate trees, this research is known as *deforestation* (a pun due to Wadler [Wad90]).

While there have been many advances in the theory of deforestation, relatively little experimental work has been conducted on the efficacy of deforestation in practice. In the summer of 2000, Patty Johann (then at Bates College, now at Dickinson College) and I started a project on the empirical evaluation and comparison of three approaches to deforestation:

- Launchbury and Sheard's warm fusion [LS95] as implemented by Németh [Ném00];
- Hu, Iwasaki, and Takeichi's hylo fusion [HIT96], as implemented by Onoue [OHIT97];
- Chitil's type-inference based approach to short-cut deforestation [Chi99].

Our goal is to compare implementations of all three approaches within the same implementation of Haskell on the same set of benchmarks so that we can better understand the benefits and drawbacks of each approach. We are particularly interested in exploring how deforestation interacts with other program transformations.

During the summer of 2000, Patty Johann and I used this empirical study of deforestation as the basis for a cross-institutional summer research experience for six undergraduate students (two from Bates, four from Wellesley) that was partially supported by my NSF grant CCR-9804053. Because of the tie to deforestation, this summer experience was christened *Lumberjack Summer Camp*. The ten-week project began with a short course bringing the students up to speed on functional programming and program transformation, after which students chose an individual project that contributed to the larger empirical evaluation project. Some students studied, modified, or extended the existing deforestation engines, while others worked on benchmarks and support software for the project. The structure of this summer research experience is described in more detail in **[JT01]**.

The empirical evaluation project is still ongoing, and several students continue to contribute to the project. For her honors thesis, Wellesley student Kirsten Chevalier '01, one of the summer 2000 lumberjacks, extended Chitil's simple implementation of his system to work on real Haskell programs. Kirsten is planning to continue with this and related programming language projects when she becomes a computer science graduate student at Berkeley this fall. Julie Weber '03 is continuing the work of Kate Golder '02 on implementing a *demodulizer* for Haskell — a program that can convert a multi-module Haskell program into a single module. This is important for our project because most deforestation engines only work (or work best) on whole programs, yet many standard benchmark programs are built out of multiple modules. Nausheen Eusuf '02 is planning an independent study project in which she continues her work from summer 2000 on expressing automata theory algorithms in a functional style; we plan to add her programs to our deforestation benchmark suite. We are continuing the empirical deforestation study with the goal of producing several papers with our undergraduate student contributors as co-authors.

Inspired by my experience with flow-based transformations from the Church Project, I am also planning to conduct research on flow-based approaches to deforestation. Many deforestation techniques are based on the *short-cut rule*, a kind of cancellation rule that transforms expressions of the form `(foldr` *k z* `(build` *g*`))` to `(`*g k z*`)`. The applicability of this rule depends on whether enough inlining has been performed to bring the `foldr` and `build` together syntactically — something that depends heavily on the inlining algorithm used within a compiler, and when it is applied relative to deforestation. Chitil developed a clever type-based approach that (1) infers the placement of `builds`, so programmers do not have to write them explicitly and (2) detects and performs inlining that will be profitable for applying the short-cut rule. I plan to investigate the application of flow-based inlining techniques to Chitil's approach to see if this can further enhance the applicability of the short-cut rule.

5 Robotics

Since January 1996, I have worked with Robbie Berg of Wellesley's Physics department to develop *Robotic Design Studio*, an introductory robotics course that teaches the "big ideas" of engineering in the context of a liberal arts education. In [**TB0X**], we describe our course and the principles behind it. Although engineering has traditionally been viewed as being incompatible with the liberal arts, we argue that (1) an exposure to engineering should be a key component of a liberal arts experience and (2) robotics is a particularly good domain for introducing engineering to liberal arts students. We have also found that it is a rich area for independent projects accessible to students.

In many ways, Robotic Design Studio has exceeded our wildest expectations. Our course has no prerequisites and over the last six years has been taken by over 115 students with a wide range of backgrounds. Representing 37 different departmental majors and often coming without prior programming or mechanical building experience, our students have created robots that surprise and delight us with creativity and ingenuity. The best way to get a sense for the variety of these projects is to visit the on-line museum of student projects at `http://cs.wellesley.edu/rds/museum.html`.

A key feature of our course is that it culminates in an exhibition, where students display their final robotic projects. This stands in contrast to the vast majority of robotic experiences, which culminate in a competitive tournament. While competitions are exciting and motivational for many students (particularly the winners), we believe that an exhibition format is more welcoming to novices, attracts a broader range of students, and allows room for a greater range of creative expression, while still maintaining the motivational benefits of a public display of the projects.

Our course has had high visibility and has generated excitement not only among Wellesley students but also among the greater Wellesley College community and at other liberal arts colleges. Faculty at several other liberal arts colleges (most prominently, Middlebury College) have followed our lead by adapting the Robotic Design Studio course. We have begun to work with the organizers of Botball (`www.botball.org`), a robotics tournament targeted at middle school and high school students, to include exhibition-style activities at their tournaments.

The accessibility of robotics has made it a good area for undergraduate student projects. I have supervised five undergraduates (in three teams) who have constructed candle-extinguishing robots for Trinity College's annual Fire-Fighting Robot Contest (`http://www.trincoll.edu/events/robot`). I have also supervised two undergraduate research projects involving graphical programming languages for robotics (see Section 6).

6 Visual Programming

I have had a long-term interest in graphical programming environments and the visualization of computational processes. This interest began with my MIT Master's thesis on a visual programming environment for a procedural language supporting first-class procedures [Tur86]. Although visual programming has not been on the main trajectory of my programming language research, its accessibility has made it a good area for undergraduate research projects:

- In the summer of 1996, I supervised Laura Diao '98 in a summer research project, "Visual Robot Programming", in which she designed a graphical rule-based programming language for controlling robots.
- In the 1996-7 academic year, I supervised the honors thesis project of Anna Mitelman '97 on "Visual Graph Abstraction", in which she designed and implemented a Java program for viewing a graph of nodes and edges in terms of "supernodes" (subgraphs) connected via "superedges".
- In the summer of 1997, I supervised Yan Zhang '99 in a summer research project, "Visual Term Rewriting", in which she implemented a Java program for animating the sequence of terms (displayed as trees) generated by a term rewriting system. We use Yan's program in our introductory programming class to illustrate how the computational processes generated by recursive functions grow and shrink over time.
- In the spring semester of 2000, I supervised an independent study project by Emily Horton '00 in which she extended the LogoBlocks visual robotics programming language developed at the MIT Media Lab. She was subsequently hired by the Media Lab as a summer intern to continue this work.

7 Information-Sharing Systems

In 1985, I (along with Ken Grant, and under the supervision of Tom Malone of the MIT Sloan School) implemented the first version of the Information Lens, an early email filtering and information-sharing system. This work is described in [**MGT$^+$87**], which expands on work reported earlier in [MGT86]. This work has been frequently cited in the literature.

8 Pedagogical Publications

The remainder of the publications in my binder have a pedagogical flavor. Arranged in reverse chronological order, these are:

- Beginning in the fall semester of 1997, I overhauled Wellesley's introductory programming course, *Introduction to Programming and Problem Solving (CS111)*. In addition to switching from Pascal to Java, I changed the order and emphasis of a number of topics. One of the most important changes was instituting a *recursion early* approach, in which recursion is not only taught in-depth but is taught before iteration. While this approach is common in the teaching of functional programming languages, it is almost unheard of in the teaching of imperative and object-oriented languages. Other important changes were teaching linked lists before arrays and teaching the *Java Execution Model*, an explicit model of Java evaluation that I developed. This work is described in [**TRSH99**], for which I was the lead author and developer. Materials from the last time I taught the course can be found at `http://cs.wellesley.edu/~cs111/spring00`.

- Between 1988 and 1994, as a Ph.D student at MIT, I was the main developer of materials (lecture notes and problem sets) for MIT's graduate programming languages course (6.821). Starting with preliminary notes written by David Gifford (MIT) and working with several other graduate students (particularly Jonathan Rees, Trevor Jim, and Brian Reistad), I developed an extensive set of course notes for 6.821 [**TGR00**] that is still used for that course. I have not worked on the notes since the summer of 1995, but in the near future plan to resume my collaboration with David Gifford to revise these notes and publish them as a book.

- During the summer of 1988, I, together with Mike Eisenberg and under the supervision of Roy Pea (then at New York University), designed and implemented *Creatures of Habit*, a computer-based microworld for engaging middle-to-high school students in the process of scientific inquiry. This work is described in [**PET88**].

- In 1986, Mike Eisenberg, Mitch Resnick, and I conducted interviews with students in MIT's introductory programming course (6.001) to better understand their conceptual difficulties in understanding the notion of first-class procedures in the Scheme programming language. We worked closely to interpret the results of these interviews, which we reported in [**ERT87**].

References

[AC93] Roberto Amadio and Luca Cardelli. Subtyping recursive types. *ACM Trans. on Prog. Langs. & Systs.*, 15(4):575–631, 1993.

[Age95] Ole Agesen. The Cartesian product algorithm. In *Proceedings of ECOOP'95, Seventh European Conference on Object-Oriented Programming*, volume 952, pages 2–26. Springer-Verlag, 1995.

[App92] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[AT00] Torben Amtoft and Franklyn Turbak. Faithful translations between polyvariant flows and polymorphic types. In *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000*, volume 1782 of *Lecture Notes in Computer Science*, pages 26–40. Springer-Verlag, 2000.

[AT0X] Torben Amtoft and Franklyn Turbak. Faithful translations between polyvariant flows and polymorphic types. Technical report, 200X. Draft of an expanded full report of [AT00], in preparation for submission to *ACM Transactions on Programming Languages and Systems (TOPLAS).*

[Ban97] Anindya Banerjee. A modular, polyvariant, and type-based closure analysis. In *Proc. 1997 Int'l Conf. Functional Programming.* ACM Press, 1997.

[BCDC83] Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *J. Symbolic Logic*, 48(4):931–940, 1983.

[BDCd95] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and union types: Syntax and semantics. *Inform. & Comput.*, 119:202–230, 1995.

[BKR98] Nick Benton, Andrew Kennedy, and George Russell. Compiling Standard ML to Java bytecodes. In *Proc. 1998 Int'l Conf. Functional Programming.* ACM Press, 1998.

[CDCV80] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Principal type schemes and λ-calculus semantics. In J[onathan] P. Seldin and J. R[oger] Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 535–560. Academic Press, 1980.

[CDCV81] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Functional characters of solvable terms. *Z. Math. Logik Grundlag. Math.*, 27(1):45–58, 1981.

[Chi99] Olaf Chitil. Type inference builds a short cut to deforestation. In *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, pages 249–260. ACM, September 1999.

[Chi00] Olaf Chitil. *Type-Inference Based Deforestation of Functional Programs.* PhD thesis, Aachen University of Technology (RWTH Aachen), 2000.

[CJW00] Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion for typed languages. In *Programming Languages & Systems, 9th European Symp. Programming*, volume 1782 of *LNCS*, pages 56–71. Springer-Verlag, 2000.

[DMTW97] Allyn Dimock, Robert Muller, Franklyn Turbak, and J. B. Wells. Strongly typed flow-directed representation transformations. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 11–24. ACM Press, 1997.

[DWM$^+$01a] Allyn Dimock, Ian Westmacott, Robert Muller, Franklyn Turbak, and J. B. Wells. Functioning without closure: Type-safe customized function representations for Standard ML. In *Proc. 2001 Int'l Conf. Functional Programming.* ACM Press, 2001.

[DWM$^+$01b] Allyn Dimock, Ian Westmacott, Robert Muller, Franklyn Turbak, J. B. Wells, and Jeffrey Considine. Program representation size in an intermediate language with intersection and union types. Technical Report BUCS-TR-2001-02, Comp. Sci. Dept., Boston Univ., March 2001. This is a version of [DWM$^+$01c] extended with an appendix describing the CIL typed intermediate language.

[DWM$^+$01c] Allyn Dimock, Ian Westmacott, Robert Muller, Franklyn Turbak, J. B. Wells, and Jeffrey Considine. Program representation size in an intermediate language with intersection and union types. In *Types in Compilation 2000*, volume 2071 of *Lecture Notes in Computer Science*, pages 27–52. Springer-Verlag, 2001.

[ERT87] Michael Eisenberg, Mitchel Resnick, and Franklyn Turbak. Understanding procedures as objects. In Gary M. Olson, Sylvia Sheppard, and Elliot Soloway, editors, *Empirical Studies of Programmers: Second Workshop*, pages 14–32. Ablex Publishing Corporation, 1987.

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

[Gil96] Andrew John Gill. *Cheap Deforestation for Non-strict Functional Languages.* PhD thesis, Department of Computing Science, University of Glasgow, January 1996.

[GLJ93] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Functional Programming and Computer Architecture*, 1993.

[Hei95] Nevin Heintze. Control-flow analysis and type systems. In SAS '95 [SAS95], pages 189–206.

[HH98] John Hannan and Patrick Hicks. Higher-order uncurrying. In POPL '98 [POPL98], pages 1–11.

[HIT96] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Deriving structural hylomorphisms from recursive definitions. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 73–82. ACM, May 1996.

[Hug90] Hughes. Why functional programming matters. In David Turner, editor, *Research Topics in Functional Programming*, pages 17–42. Addison Wesley, 1990.

[Jim95] Trevor Jim. Rank 2 type systems and recursive definitions. Technical Report MIT/LCS/TM-531, Massachusetts Institute of Technology, November 1995.

[Jim96] Trevor Jim. What are principal typings and what are they good for? In *Conf. Rec. POPL '96: 23rd ACM Symp. Princ. of Prog. Langs.* ACM, 1996.

[JT01] Patricia Johann and Franklyn Turbak. Lumberjack Summer Camp: A cross-institutional undergraduate research experience in computer science. *Computer Science Education*, 11(4), December 2001. To appear.

[JW95] Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *Conf. Rec. 22nd Ann. ACM Symp. Princ. of Prog. Langs.*, pages 393–407. ACM, 1995.

[JWW97] Suresh Jagannathan, Stephen Weeks, and Andrew Wright. Type-directed flow analysis for typed intermediate languages. In *Proc. 4th Int'l Static Analysis Symp.*, volume 1302 of *LNCS*. Springer-Verlag, 1997.

[KMTW99] Assaf J. Kfoury, Harry G. Mairson, Franklyn A. Turbak, and J. B. Wells. Relating typability and expressibility in finite-rank intersection type systems. In *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, pages 90–101. ACM Press, 1999.

[KW99] Assaf 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.*, pages 161–174, 1999.

[LS95] John Launchbury and Tim Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *FPCA '95, Conference on Functional Programming Languages and Computer Architecture*, pages 314–323. ACM, June 1995.

[MGT86] Thomas Malone, Kenneth Grant, and Franklyn Turbak. The information lens: An intelligent system for information sharing in organizations. In *Human Factors in Computing Systems: CHI'86 Conference Proceedings*, pages 1–8, 1986. Superseded by [MGT$^+$87].

[MGT$^+$87] Thomas Malone, Kenneth Grant, Franklyn Turbak, Stephen Brobst, and Michael Cohen. Intelligent information-sharing systems. *Communications of the ACM*, 30(5):390–402, May 1987. Expanded version of [MGT86].

[Mor95] Greg Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University, 1995.

[MT00] Elena Machkasova and Franklyn Turbak. A calculus for link-time compilation. In *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000*, volume 1782 of *Lecture Notes in Computer Science*, pages 260–274. Springer-Verlag, 2000.

[MT0X] Elena Machkasova and Franklyn Turbak. A computationally sound call-by-value module calculus. Draft of a Boston University technical report that expands on [MT00], 200X.

[MW00] Robert Muller and J. B. Wells. Two applications of standardization and evaluation in Combinatory Reduction Systems. Submitted for publication, 2000.

[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.

[Ném00] László Németh. *Catamorphism Based Program Transformations for Non-Strict Functional Languages*. PhD thesis, Department of Computing Science, University of Glasgow, 2000. Draft of dissertation.

[NN97] Flemming Nielson and Hanne Riis Nielson. Infinitary control flow analysis: A collecting semantics for closure analysis. In *Conf. Rec. POPL '97: 24th ACM Symp. Princ. of Prog. Langs.*, pages 332–345, 1997.

[OHIT97] Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. A calculational fusion system hylo. In *IFIP TC 2 Working Conference on Algorithmic Languages and Calculi*, pages 76–106. Chapman & Hall, February 1997.

[PET88] Roy Pea, Michael Eisenberg, and Franklyn Turbak. Creatures of Habit: A computational system to enhance and illuminate the development of scientific thinking. In *Proceedings of the Tenth Annual Conference of the Cognitive Science Society*, pages 340–346, 1988.

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

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

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

[Plo75] G[ordon] D. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoret. Comput. Sci.*, 1:125–159, 1975.

[PO95] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. *ACM Trans. on Prog. Langs. & Systs.*, 17(4):576–599, 1995.

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

[PP98] Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. In POPL '98 [POPL98], pages 197–208. Superseded by [PP01].

[PP01] Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. *J. Funct. Programming*, 11(3):263–317, May 2001.

[RDRV84] Simona Ronchi Della Rocca and Betti Venneri. Principal type schemes for an extended type theory. *Theoret. Comput. Sci.*, 28(1–2):151–169, January 1984.

[Rey96] John C. Reynolds. Design of the programming language Forsythe. In P. O'Hearn and R. D. Tennent, editors, *Algol-like Languages*. Birkhauser, 1996.

[SAS95] *Proc. 2nd Int'l Static Analysis Symp.*, volume 983 of *LNCS*, 1995.

[Sch95] David Schmidt. Natural-semantics-based abstract interpretation. In SAS '95 [SAS95], pages 1–18.

[SF93] Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *FPCA '93, Conference on Functional Programming Languages and Computer Architecture*, pages 233–242. ACM, June 1993.

[Sha97] Zhong Shao. An overview of the FLINT/ML compiler. In TIC '97 [TIC97].

[Shi91] Olin Shivers. *Control Flow Analysis of Higher Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

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

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

[TB01] Franklyn Turbak and Robert Berg. Robotic Design Studio: Exploring the big ideas of engineering in a liberal arts environment (extended abstract). In *Proceedings of the AAAI Spring Symposium on Robotics and Education*, 2001. This is an extended abstract of [TB0X].

[TB0X] Franklyn Turbak and Robert Berg. Robotic Design Studio: Exploring the big ideas of engineering in a liberal arts environment. Submitted to the *Journal of Science Education and Technology.* This is an expanded version of [TB01], 200X.

[TDMW97] Franklyn Turbak, Allyn Dimock, Robert Muller, and J. B. Wells. Compiling with polymorphic and polyvariant flow types. In *Proceedings of the 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC '97)*, 1997. Boston College Computer Science Dept. Technical Report BCCS-97-03. Superseded by [WDMT0X].

[TGR00] Franklyn Turbak, David Gifford, and Brian Reistad. *Applied Semantics of Programming Languages*. Course notes for 6.821, MIT's graduate programming languages course., 2000.

[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/.

[TM95] Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *FPCA '95, Conference on Functional Programming Languages and Computer Architecture*. ACM, June 1995.

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

[TRSH99] Franklyn Turbak, Constance Royden, Jennifer Stephan, and Jean Herbst. Teaching recursion before loops in CS1. *Journal of Computing in Small Colleges*, 14(4):86–101, May 1999.

[Tur86] Franklyn Turbak. *Grasp: A Visible and Manipulable Model for Procedure Programs.* S.M. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology., May 1986.

[Tur94] Franklyn Turbak. *Slivers: Computational Modularity via Synchronized Lazy Aggregates.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, February 1994. Accessible from `http://www-swiss.ai.mit.edu/~lyn/slivers.html`.

[Tur96] Franklyn Turbak. First-class synchronization barriers. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 157–168. ACM Press, May 24–26 1996.

[TW01] Franklyn Turbak and J. B. Wells. Cycle therapy: A prescription for fold and unfold on regular trees. In *Proc. 3rd Int'l Conf. on Principles and Practice of Declarative Programming*, Firenze, Italy, 5–7 September 2001.

[vB93] Steffen J. van Bakel. *Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems.* PhD thesis, Catholic University of Nijmegen, 1993.

[Wad90] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[WDMT97] J. B. Wells, Allyn Dimock, Robert Muller, and Franklyn Turbak. A typed intermediate language for flow-directed compilation. In *TAPSOFT' 97, Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE*, pages 757–771, 1997. Superseded by [WDMT0X].

[WDMT0X] J. B. Wells, Allyn Dimock, Robert Muller, and Franklyn Turbak. A calculus with polymorphic and polyvariant flow types. *Journal of Functional Programming*, 200X. Accepted; to appear. Expanded version of [WDMT97] and [TDMW97].

[WJ98] Andrew Wright and Suresh Jagannathan. Polymorphic splitting: An effective polyvariant flow analysis. *ACM Trans. on Prog. Langs. & Systs.*, 20:166–207, 1998.