Thesis Proposal:
Data Representations and Polymorphic Languages

Greg Morrisett

December 1993

Abstract

Languages that provide polymorphism tend to place restrictions on object representations, separate compilation, and/or performance. For instance, most Ada implementations require that polymorphism be resolved before compilation. Consequently, separate compilation is sacrificed, since eliminating polymorphism is a cross-module transformation. Some languages, like C++ and Modula-3, restrict the representations of polymorphic objects to "pointer" types. These languages provide good separate compilation but restrict programmers from writing certain types of polymorphic code. In addition, some runtime overhead might be paid due to the extra pointer indirection. Languages such as Scheme, Lisp, SML, and Haskell, place no explicit restrictions on the representations of objects. However, almost all implementations use the "pointer" restriction to implement the polymorphic features. In fact, most implementations make *every* object a pointer. This allows the languages to support very flexible forms of polymorphism and separate compilation. However, the performance of such languages suffers due to the space and time overheads introduced by the indirection.

The goal of this work is to show that polymorphic languages can be implemented without restrictions on object representations or separate compilation. Furthermore, by lifting these restrictions, programmers and compilers can choose representations that lead to superior performance.

1 Introduction

A *polymorphic* object is an object that is parameterized by a type. For example, a module that imports an abstract type `T` is a polymorphic module. Polymorphism is traditionally broken into two classes [20], *parametric* and *ad-hoc*. Conceptually, a *parametric* object behaves the same regardless of the type that is substituted for the parameter, while an *ad-hoc* object's behavior can depend upon the type substituted for the parameter. Many statically typed languages such as Ada, C++, CLU, Modula-3, NESL, and Standard ML, provide parametric types, operations, modules, and/or values, while many dynamically typed languages such as Smalltalk, Lisp, and Scheme provide ad-hoc operations.

Implementing a language with polymorphic features can be difficult because compilers need to know the size, or more generally, the shape of objects in order to generate code to manipulate them. Usually, this sort of information is apparent from the type, but polymorphic objects have components of unknown or varying types.

Two basic approaches to implementing parametric polymorphic features have been used for *statically* typed languages:

1. Restrict the language (or the implementation) so that polymorphism is resolved at compile time. For example, most implementations of Ada require that type arguments to polymorphic modules be resolved at compile time.
2. Restrict the form of polymorphic objects to some "universal representation". For instance, most implementations of Haskell and SML require that objects be *boxed*. That is, objects are placed in memory and a pointer to the object is passed to functions, placed in data structures, etc.

Restricting a language so that polymorphism can be resolved at compile time places no restrictions on the representations of objects. However, the restrictions usually destroy the ability to separately compile polymorphic objects (e.g. modules) and result in code duplication. Restricting the form of polymorphic objects allows separate compilation and does not force code duplication. However, performance can

```
functor Stack (type t) =
  struct
     type stack = t list
     fun push (x:t) (s:stack) = x::s
     fun top (x::_ : stack) = x
     fun pop (_::s : stack) = s
  end
```

Figure 1: An SML Parameterized Stack Module

suffer if the "right" representation is not allowed. In particular, creating and accessing boxed objects requires an extra memory operation, due to the indirection.

Dynamically typed languages usually *tag* values with types so that type-checking may be performed at run-time. Since the shape of an object can usually be determined from the tag, implementing polymorphic features in such a language is not difficult. However, recent work on *soft typing* [5, 11, 1] has shown that many of the run-time type tests can be eliminated at compile-time by using type inference. In order to eliminate most tests, the inference systems are based on parametrically polymorphic type systems. Consequently, techniques used to implement parametric polymorphism for statically typed languages can apply to dynamically typed languages.

The goal of this work is to show that languages with parametrically polymorphic type systems can be implemented without restrictions on object representations and without restrictions on separate compilation. Furthermore, by lifting these restrictions, programmers and compilers can choose representations that lead to superior performance. This paper proceeds as follows: Section 2 discusses polymorphism, why implementing it is a problem, and the primary approaches that have been used to solve the problem in the past. Section 3, discusses applications that require control over data representations and how such control interacts with polymorphism. Section 4, discusses an approach to implementing polymorphism based on a very general notion of *tag*. The approach places few restrictions on the representations of objects, allowing us to choose representations and even expose them to the programmer. Section 5 presents the work that I propose to do in order to validate the thesis. The work consists of language design, theory, implementation, and performance evaluation. Section 6 discusses related work and Section 7 closes with a summary of expected contributions.

2 Polymorphism

Figure 1 shows an SML module that takes a type `t` as an argument and produces an implementation of a stack of objects of type `t`. The module is parametric in `t` because it implements a stack regardless of the type substituted for `t`. Parametric module polymorphism appears in many languages including Ada, CLU, Modula-3, and Standard ML (SML). Ideally, a parametric module can be compiled in isolation, that is, without knowledge as to what types that will be substituted for the parameters. Typically, polymorphic modules are instantiated at *link-time*.

Parametrically polymorphic types, values, and operations appear in languages such as Haskell, Hope, Miranda, and SML. For example, the SML function declaration:

```
fun pair (x) = (x,x)
```

has the type $\forall\alpha.\alpha \rightarrow (\alpha * \alpha)$. This type is parameterized by the type variable α. Polymorphic types, values, and operations in these languages may be applied to a type at *run-time*.

Ad-hoc operations appear in languages such as C++, Smalltalk, Lisp, and Scheme. For example, one might write an increment routine in Scheme as:

```
(define increment
   (lambda (x)
      (cond ((integer? x) (integer_plus x 1))
            ((real? x) (real_plus x 1.0))
            ...
            )))
```

This operation takes an object of any type but behaves differently things (i.e. adds an integer, adds a real, fails) according to the type. Obviously, polymorphic operations like this are applied to a type a *run-time*.

2.1 Why is Implementing Polymorphism a Problem?

Consider the SML function definition:

```
fun pair (x:int) : (int * int) = (x,x)
```

This function takes an integer and produces two copies of the integer in a tuple (i.e. record). In order to allocate memory for the tuple, the compiler needs to know the tuple's size. For instance, an int might occupy 32 bits and thus a tuple of ints might occupy $32 + 32 = 64$ bits.

Similarly, for a function that returns the second component of a tuple of ints:

```
fun second ((x,y) : (int * int)) : int = y
```

the compiler needs to know the position and size of the second argument. In this case, it might be 32 bits from the start of the tuple.

Now consider the polymorphic versions of these two functions:

```
fun pair (x:'a) : ('a * 'a) = (x,x)
fun second (x:'a,y:'b) : 'b = y
```

How will the compiler determine the size of the result of `pair`? How will the compiler determine the position of the second component of the tuple in `second`? These properties depend upon how the types are instantiated.

Section 2.2 discusses two approaches that have been used to solve these problems in the past. Section 4 describes an alternative approach that I feel is superior to these previous approaches.

2.2 Previous Approaches to Polymorphism

Restrict and Eliminate: Some languages, such as Ada [24] and NESL [4], make restrictions so that all polymorphism may be eliminated at compile time. In these languages, a polymorphic object M parameterized by type T and defined by N is treated by the compiler as a function of the form:

$$M(T) = \mathsf{compile}(N)$$

That is, compilation is delayed until the actual argument for T is known. Consequently, the compiler never compiles an object with a polymorphic type. The restrict and eliminate approach to polymorphism insures that there is no runtime penalty for the polymorphism. Polymorphic and monomorphic code and data structures can look exactly the same and offer very high performance. Unfortunately, this approach can lead to code duplication since a new copy of N is generated every time M *might be* applied. More troublesome, this approach does not allow separate compilation of polymorphic objects.

Restrict to Pointers: Some languages restrict the set of types that can instantiate a polymorphic object. For instance, Modula-3 only allows pointer types (i.e. boxed objects) to instantiate polymorphic objects. Even though there is no requirement in the languages, most implementations of Lisp, Scheme, Haskell, and SML actually make the restriction that *all* objects must be boxed, since an object might be passed to any polymorphic operation[1]. The restriction to pointers makes it possible to calculate the size of polymorphic objects since they are always the size of a pointer (e.g. one machine word.) This approach allows one to separately compile polymorphic objects and does not require code duplication.

Unfortunately, the indirection of pointer objects can cost substantial amounts of space and time. Consider SML/NJ which provides both polymorphic arrays and floating point arrays. A polymorphic array instantiated to a floating point array can occupy twice as much space as the built-in floating point array, simply because SML/NJ uses the pointer restriction. In addition, reading/writing an element out of the polymorphic array requires extra reads/writes to memory. Consequently, as processors become faster with respect to memory, polymorphic arrays will become slower when compared to monomorphic floating-point arrays.

Performance is not the only consideration when it comes to the pointer restriction. Some programs need to manipulate data without indirection for *correctness* reasons. Section 3 discusses some of the issues involved in exposing representations in a polymorphic language.

[1] Previous work by Leroy has shown how some, but not all of the boxing can be eliminated for monomorphic code [13].

3 Exposing Data Representations

High-level languages such as SML do not specify representations of objects. While there are many good reasons for this (portability, modularity, compiler freedom, etc.), there are a variety of applications that *require* precise control over the representations of objects. Most of these applications need to communicate with some other entity and are constrained to use a particular representation in the communication. For example:

- **Memory-mapped devices**, such as frame buffers, require that all data be placed in a particular set of contiguous memory locations.
- **Garbage collectors** manipulate data whose format is constrained by other aspects (i.e. the compiler) of a language and its implementation.
- **Foreign language and operating system interfaces** are constrained by calling conventions and foreign data formats.
- **Communication protocols**, (e.g. TCP, IP, UDP) require that packets have a particular format.
- Compilers use **intermediate languages** that make representation choices explicit.

All of these applications require precise, programmer control over some data structures and cannot tolerate implicit representation changes. Many of these applications are coded in relatively low-level languages such as C precisely for this reason.

As shown in the previous section, techniques for implementing polymorphism typically restrict representations or separate compilation. Thus, it appears that a "systems" programming language that gives programmers control over representations must sacrifice polymorphism, separate compilation, or both.

4 The Tagging Approach to Polymorphism

How can we implement polymorphic languages so that there is neither a restriction on separate compilation nor a restriction on object representation? A very general strategy is to carry a *tag* for an object at runtime and to make primitive polymorphic operations interpret the tag. For instance, the polymorphic `pair` function would look at the tag of its argument to determine the object's size. Thus, the parametric `pair` function is compiled into an ad-hoc function whose behavior depends upon types. This approach is used to implement *dynamically* typed languages.

However, in a dynamically typed language, tags are placed with the data values, so the representation of a value is not completely free. Tags and values must be separated so that representations are not restricted in order to support applications like those described in Section 3. Section 4.1 describes a simple approach to this problem that I have already implemented. The basic idea is to associate compound tags with compound objects, but leave the components tag-free. This simple technique allows us to choose representations and even expose them to the programmer. However, the technique requires a tag for each object, even if the object is not passed to a polymorphic function. Section 4.2, describes a much more promising approach that allows one to *reconstruct* the type of an object so that objects need never be tagged. This approach frees representations to the degree that an object's representation can be recovered from its type. In Section 4.3 I show how the same compilation strategy can be used to implement other language features that have generally relied on tags, including garbage collection.

The techniques proposed do have potential drawbacks. Obviously, polymorphic primitive operations will be slowed since they must interpret tags at run-time. But if polymorphic operations occur rarely or can be made to occur rarely, then this may not matter. Section 4.4 discusses techniques that can be used to eliminate polymorphism. I expect that the benefits from freeing representations will out-weigh the costs of interpreting tags in the polymorphic operations that will remain, once these optimizations are applied.

4.1 Lifted Tags

The traditional way to tag an object is to use a pair whose first component is a *constructor* and whose second component is the actual value of the object. The constructor only describes the "top-most" information about the value. For instance, the constructor might simply indicate that the value is an n-tuple, but not describe the components of the n-tuple, since each of the components will be self-describing. This scheme is pictured in Figure 2.

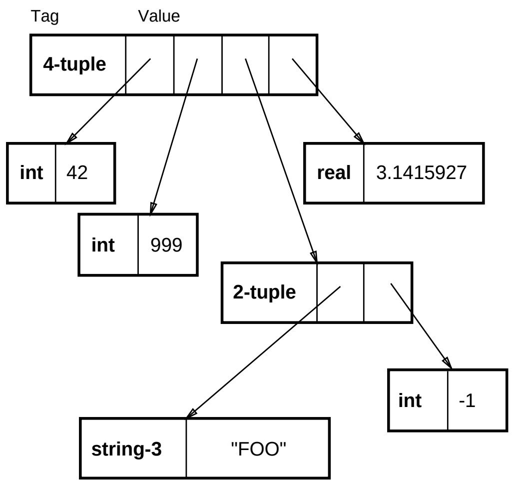

Figure 2: Traditional Tagging

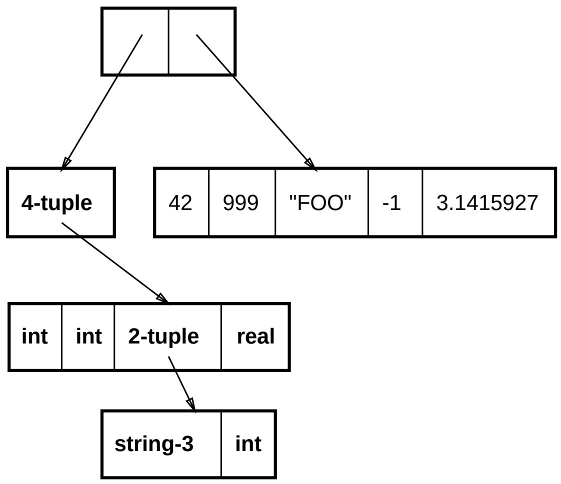

Figure 3: Lifted Tagging

The traditional approach either mixes the tags with the data or else requires an extra level of indirection for each component of an aggregate. Consequently, we cannot use this approach for "flattened" representations where components of an object are unboxed. However, if we allow tags to be compound data structures themselves, then we can *lift* the tags of the components into the outermost tag. Figure 3 shows how tags might be lifted in a system that chooses to flatten tuples.

Tagging data may seem unreasonable for statically typed languages if the only reason is to support flattened data structures. However, tagging is often used to implement other language features. For instance, the Modula-3 type-case statement lets one discover the type of a value at runtime. Tagging is a simple way to accommodate such a primitive. Implementations of languages such as Haskell, Modula-3, and SML must provide garbage collection. Most forms of *accurate* garbage collection (as opposed to conservative garbage collection) require that data be tagged so that the size of an object and any pointers it contains may be identified. As already stated, languages like Lisp and Scheme that are dynamically typed require some form of tagging. Thus, for many language implementations, lifted tags are a reasonable approach for dealing with a variety of data representations.

I have modified the SML/NJ compiler and runtime to support lifted tagging and flattened representations of integers (of various sizes), records, vectors, and refs. Modifying the compiler to support flattened representations through lifted tagging did not prove to be too difficult because SML/NJ tags all data for garbage collection and polymorphic-equality reasons. The FoxNet project at CMU is using this modified compiler to implement a variety of communications-oriented programs, including TCP/IP Internet protocol suite, Kerberos authentication, and Mach 3.0 services [3].

Lifted tags are not an ideal solution, however, because even if no polymorphic features are used, values continue to be tagged. In addition, top-level objects must be tagged with a rather cumbersome description which in turn imposes some restrictions on representations and calling conventions. Unboxing techniques like Leroy's [13] can be used to eliminate some of the tagging and indirection for monomorphic types. The basic idea is to convert values to the boxed and tagged state when passed to polymorphic code and to unbox and untag the values when returned from a polymorphic function. Unfortunately, it is not practical to unbox/untag lists (or any other recursive datatype) with this technique, and it is not even possible to unbox/untag mutable objects.

4.2 Separated Tags

In this section, I describe a tagging technique that *separates* tags and data entirely. Consequently, few restrictions are placed on the representations of polymorphic objects and tags are only manipulated in polymorphic code.

The essential idea behind this technique is to make the parameterization of a polymorphic object *explicit*. That is, as in the polymorphic (second-order) typed lambda calculus [19, 8, 7], all polymorphic definitions are converted into explicit functions from types to objects. Instantiating a polymorphic object will become an explicit application of a polymorphic object to a type.

As an example, consider the SML `pair` function and its use at two different types:

```
let val pair = fn x => (x,x)
in
   pair 1;
   pair true
end
```

Since pair is assigned the type $\forall\alpha.\alpha \rightarrow (\alpha * \alpha)$, the expression will be translated into:

```
let val pair = fn 'a => fn x => (x,x)
in
  pair int 1;
  pair bool true
end
```

Assuming tags can be computed from types, this approach will allow us to reconstruct the tag of a polymorphic object at runtime without actually tagging the object. Unlike previous approaches, no constraints will be placed on representations.

A strong advantage of this approach is that no new concepts are introduced into the language. If an implementation provides efficient support for functions, then it would seem possible to provide efficient support for polymorphism.

Unfortunately, this approach is not possible for languages that allow polymorphic objects to have computational effects including non-termination, since it is not always possible to turn polymorphic objects into functions. For example, consider the following code:

```
let val pair = (print "Hello"; fn x => (x,x))
in
    pair 1;
    pair true
end
```

Our translation would yield:

```
let val pair = fn 'a => (print "Hello"; fn x => (x,x))
in
    pair int 1;
    pair int true
end
```

The first program will print `Hello` one time while the second one will print `Hello` two times. By turning polymorphic objects into functions, the translation delays the effects of polymorphic objects until they are used (if ever). For so-called "lazy" languages such as Haskell where polymorphic objects do not have computational effects, the approach is sound.

Allowing polymorphic objects to have computational effects leads to other problems. For instance, assigning a polymorphic type to `ref` (mutable objects) or `callcc` is not sound. Various approaches to preventing this problem have been proposed, including Tofte's imperative types [22] and SML/NJ weak types [14, 10, 12]. Many people feel that the easiest solution to these problems is to only allow *values* (i.e. identifiers, functions, etc.) to be assigned a polymorphic type. If such a restriction were used, then our translation would be valid.

A key question is whether this "value restriction" is reasonable. That is, how often do programmers write code where non-values are assigned polymorphic types? Recently, Wright has shown that a large number of real SML codes, including the SML/NJ Library, ML-Lex, and SourceGroup only compute polymorphic values [25]. For other programs, all but a very few of the expressions were without effects. Thus, by eta-expanding these expressions (i.e. turning them into functions), he was able to assign them polymorphic types. The remaining programs were easily transformed so that only values were assigned polymorphic types. Thus, it appears that the value restriction is reasonable.

Another key question regarding this approach towards implementing polymorphism is what overhead is imposed by making polymorphic objects explicit functions. Section 4.4 discusses techniques that can be used to minimize this overhead.

Finally, since this technique allows us to eliminate tags in order to implement polymorphic operations, a key question is whether we can eliminate *all* tags. As noted earlier, efficient implementations of other language features seem to require tags. The following section discusses how many of these other features can be implemented if the separated tagging scheme is used.

4.3 Other Uses for Separated Tags

It is possible to implement a variety of other interesting language features if the separated tagging scheme is used. The features include but are not limited to:

1. **Structural equality**: In Lisp, Scheme, and SML, values may be compared structurally for equality. Doing so requires size, shape, and pointer information.
2. **Type dynamic and type-case**: Type dynamic is essentially a way to bundle an object with its type at runtime. The *type-case* expression allows one to branch on the type associated with a dynamic value. Obviously, creating a dynamic value requires type information.
3. **Polymorphic record calculi**: Polymorphic record calculi provide a form of subtyping where operations can be defined on records that contain at least a subset of some fields. This can be used to simulate some aspects of object-oriented "method-dispatch" programming. This requires information about the positions of fields in a record.
4. **Accurate garbage collection**: Accurate garbage collection ensures that only "reachable" objects are not garbage collected. This seems to require size, shape, and pointer information.

This section discusses how one of these features, namely garbage collection, can be implemented with separated tags.

As with compilation, it is possible to do accurate, tagless garbage collection for statically typed, monomorphic languages. That is, it is possible for a program to never tag data and still be able to collect all non-reachable data. The basic idea is to associate with the code for each procedure the types of all free variables in the closure (i.e. stack frame) of the procedure. From the types, the garbage collector is able to determine the size of objects and any pointers they hold.

Appel [2] and Goldberg & Gloger [9] have shown that, in theory, it is possible to do tagless garbage collection for *parametrically polymorphic* languages. The idea is to reconstruct the types that were substituted for polymorphic parameters by walking up the stack. Unfortunately, it is not possible to reconstruct the types of all reachable objects. The problem is that the closure for a function that created a polymorphic value may not be reachable, so it may not be possible to reconstruct the value's type. The following example from Goldberg & Gloger demonstrates the problem:

```
let fun f (x : 'a list) = fn z => if length x = 1 then z+1
                                  else z-1
    val g : int -> int = if ... then f [1,2,3] else f [[1,2],[3,4]]
in
  g (hd [1])
end
```

If garbage collection is initiated during the construction of the list [1], then the garbage collector will be unable to determine the type of the free variable x in g's closure. Fortunately, Goldberg & Gloger have proven that if you cannot reconstruct an object's type, that object is effectively garbage and can be collected. In the example above, x is not garbage, but it is a list of elements that are all garbage. Thus, only the spine of x needs to be preserved.

Unfortunately, reconstructing an object's type by walking up the stack can be expensive. In addition, the inability to reconstruct all reachable objects' types makes garbage collection difficult. For instance, it might be possible to determine the type along one path to an object, but not along another. Thus, the object cannot be collected until all possible paths have been explored. This seems to require a second pass over the heap. But, the situation is even worse! If an object is discovered that must be preserved in a second pass, then it might have pointers to polymorphic objects that lead to a third pass, and so on.

Fortunately, it is possible to reconstruct the type of every object if separated tags are used. The reason is that types bound to type variables are carried in closures along with objects. Only if the object is not used is the type inaccessible. And if the object is not used, it is garbage. For instance, consider Goldberg & Gloger's example. This will be translated into:

```
let fun f 'a x = fn z => if length 'a x = 1 then z+1
                          else z-1
    val g = if ... then f int [1,2,3] else f (int list) [[1,2],[3,4]]
in
  g (hd int [1])
end
```

Note that now, the type of x can be reconstructed since the type of the components of the list is bound to the variable 'a.

4.4 Eliminating Polymorphism

Since both lifted tagging and separated tagging make polymorphic operations more expensive than their monomorphic counterparts, it is important that we eliminate as much polymorphism as possible. I propose the use of at least two techniques:

Duplicate: The simplest way to eliminate polymorphism is to duplicate definitions of polymorphic objects and instantiate them at different types at compile time. For instance, the pair example of Section 4.2 might be rewritten as:

```
let val pair_int = fn (x:int) => (x,x)
    val pair_bool = fn (x:bool) => (x,x)
in
  pair_int 1;
  pair_bool true
end
```

However, as noted earlier, this approach does not work across modules, top-level loop transactions, etc. because you cannot determine all of the types to which a polymorphic object will be instantiated without looking at the whole program. Nevertheless, the technique works within a compilation unit.

Infer Minimal Types: Languages such as SML that are based on the Hindley-Milner type system often assign types to expressions that are more general than needed. "Extra" polymorphism is introduced by the type inference algorithm W which generalizes unknown types at let-expressions. For example, in the following code, pair is assigned a polymorphic type even though it is only used at one type.

```
let val pair = fn x => (x,x)
in
    pair 1; pair 2
end
```

An alternative type-inference algorithm is to look at the uses of a let-bound variable, determine a type that satisfies each of the uses with as little polymorphism as possible, and assigns that type to the let-bound variable. As with duplicating, this approach only works within a compilation unit. Unlike inlining, it facilitates code sharing.

5 Proposed Work

In this section, I outline the work that I propose to do for this thesis. Like any "language" thesis, there are roughly four parts to the work:

1. Theory
2. Language Design
3. Implementation
4. Evaluation

Section 5.1 discusses the language formalism I will develop to model and prove correct the proposed language features and implementation techniques. Section 5.2 discusses the language features that I will develop in order to support programmer control over data representation in the presence of polymorphism and separate compilation. Section 5.3 discusses the language implementation work that I will do. Section 5.4 presents questions regarding the implementation that need to be addressed in order to evaluate and substantiate the thesis claim.

I have chosen SML as the language I will use to explore this thesis. The primary reasons for choosing SML is that it offers both module-level and core-level parametric polymorphism. Languages such as Haskell do not have modules while languages like Ada and Modula-3 do not have core-level polymorphism. I will be modifying and extending the Standard ML of New Jersey (SML/NJ) implementation. SML/NJ provides a freely available, portable, relatively high-quality, optimizing compiler.

5.1 Theory

I have been working on a framework that enables me to express and prove that implementation techniques for polymorphism are correct. The framework consists of a set of intermediate languages that make representation devices such as tags, boxing, and data structure layout explicit. An implementation technique is expressed as a translation from the source language into one of these intermediate languages. Correctness is proven by showing that if the original source program evaluates to a value v, then the translation of that program evaluates to the translation of v.

A small example is given in the appendix. It demonstrates how a simple ML-like language with nested tuples may be translated into an intermediate language that only has "right-associated" tuples. From an abstract perspective, these "right-associated" tuples are more primitive than their fully nested counter-parts.

Currently my intermediate languages are "untyped". The problem seems to stem from the fact that I need to use ad-hoc operations in the target language. Thus, parametric type systems, such as ML's cannot be used for these intermediate languages. Having a typed intermediate language is important for many reasons: First, proving that an implementation technique is correct may be simplified if we can show that types are preserved under translation. Second, if we are to expose these intermediate languages to give control over data structures, then we would like to have a sound type system. Third, if other implementation techniques, such as garbage collection, require that type information be propagated through translations, then the intermediate languages need to carry types. Finally, giving a type system to the intermediate languages helps preserve the distinction between computations on types and computations on values. These phase distinctions allow us to specialize computations at different times. Of course, we could use a fully dependent type system for the intermediate languages, but this does not help preserve phase distinctions. One of the goals of this thesis is to develop a type discipline for these intermediate languages that does preserve phase distinctions.

The proofs of correctness for a translation are usually straightforward but tedious. I plan to use the Elf system [17] to verify some of the translations. That is, I will encode the source and intermediate languages, their type systems, their dynamic semantics, and the translation itself into Elf. I expect that Elf will help eliminate some of the tedium from the proofs of correctness.

5.2 Language Design

As argued in Section 3, many applications require control over data representations. In Section 4, I presented techniques that may be used to free representations from constraints imposed by polymorphism. Yet, even if these techniques are used, there are a host of practical problems involved in exposing control over data representations to the programmer. In particular, it is difficult to expose control in such a way that programs may be written in a portable yet efficient manner. Architectural differences such as endianess, alignment, and word size are primary culprits.

One approach to these problems is to provide many versions of a type constructor, one for each architecture together with a default. For instance, we might define a set of integer type constructors of various fixed sizes, like 8,16,32, and 64 bits. The type constructor `int` would be an alias for one of these types depending upon the architecture. If programmers need control over the size of an integer, they would use the appropriate type constructor directly. If programmers do not care, then they would use the `int` type with the expectation that this is the most "efficient" type to use. This is the approach that I have used to provide support for the FoxNet project. I give the programmer 8,16, and 32 bit ints as well as default ML ints, flat records and ML records, etc.

Unfortunately, this approach leads to an exponential blow up of type constructors. For instance, to write completely portable network code requires a cross product of 8,16,24, and 32-bit size, versus little and big endian. This results in 8 different integer types that the compiler must implement. Currently, I do not have a good solution to this problem.

Mutable data also interacts badly with explicit control over data representations. For instance, since an ML ref cell is a first-class item, it can be placed in multiple data structures. This requires implementing the ref cell as a pointer in order to preserve sharing. But how do we provide large, flattened data structures whose components are updatable? The components cannot be refs, because these must be implemented as pointers. My current solution to this problem is to provide *locatives*. Conceptually, a locative is a ref to an object together with a *path* to one of its components. Updating a locative can be thought of as dereferencing the object, pulling it apart by interpreting the path, putting it back together with the new component, and then assigning the result back to the ref. Of course, this is really implemented by an in-place update of the component. Locatives can be used to express many data structures including arrays and CAML-style records. I plan to investigate efficient techniques for implementing locatives.

Another aspect of giving users control over data representations is allowing them to view a value of one type as if it has another type as long as the two types are isomorphic (or somehow equivalent) and represented the same way. For instance, it should be possible to view a flat 4-tuple of ints as a flat 2-tuple of flat 2-tuples of ints. Providing these view operations allows one to convert an object from one type to another without having to copy the data. This is important for many of the applications mentioned in Section 3, especially communications software. I plan to investigate the design and type-theoretic aspects of these view operations.

5.3 Implementation

In order to substantiate the thesis claim, I will be adding the following functionality to the SML/NJ compiler:

1. Implement the lifted tagging approach to polymorphism (see Section 4.1).
2. Implement the separated tagging approach to polymorphism (see Section 4.2).
3. Add features that give users control over data representations.

This will produce three compilers, the original, a lifted tagging version, and a separated tagging version. Only the latter two will provide support for user control over data representations.

A prototype of the lifted-tagging approach has already been implemented together with some features that give users control over data structures. This implementation needs more work to support flattened data structures efficiently. The current implementation does not support views in a type-safe manner.

In order to answer the questions posed in Section 5.4, I will measure benchmark programs running on each of the compilers. Some of the programs will be taken from the SML/NJ benchmark suite. These are general purpose programs that range from the SML/NJ compiler itself to scientific codes. None of these programs requires special data representations. The FoxNet communications suite, including the TCP, UDP, and IP protocols, will serve as representative for those programs that require control over data representation.

5.4 Evaluation

In this section, I present a set of questions to be answered about the language design and implementation. I expect that the answers to these questions will provide strong support for the thesis claim. I plan to use a combination of wall-clock time, heap size, instruction counts, and cache-miss counts to answer the questions. The instruction counts and cache-miss information will be collected via the simulation tools of Tarditi, Diwan, and Moss [21]. This information should serve as a predictor of performance for future architectures.

- Representation of types:
  - Is too much sharing lost by flattening a particular type?
  - How is program time affected by choosing to flatten a type?
  - How is program locality affected by choosing to flatten a type?
  - How is garbage collection time affected by choosing to flatten a type?
  - How is garbage collector locality affected by choosing to flatten a type?
- Lifted tagging:
  - How does lifted tagging affect type representation choices (see above)?
  - How do we represent lifted tags?
    - Should we minimize space or time?
    - How is the program affected by tag representation?
    - How is the garbage collector affected?
- Separated tagging:
  - How does separated tagging affect type representation choices (see above)?
  - How often does the value restriction get in the way?
  - How much is polymorphic code slowed down (compared to complete or lifted tagging) by having the extra type parameter?
  - How many type applications are eliminated by a standard optimizer?
  - If tags are to be retained for GC, then what representation do we use for the tags?
  - How much space is saved by eliminating all tags?
- Eliminating polymorphism:
  - How effectively does minimal type inference eliminate polymorphism?
  - When should we inline?

6 Related Work

The most directly related work to what I propose is that of Morrison et al. [15]. They describe an implementation of Napier88 that uses ad-hoc polymorphism to implement parametric polymorphism. Like separated tagging, their technique is to encode the source language into the second-order polymorphic lambda calculus. However, they still seem to require transforming data when it is passed to a polymorphic routine (see page 353). Thus, it is not possible to retain complete control over data representations. Furthermore, they do not seem to have a good comparison between their technique and other techniques.

Encoding a language into the polymorphic lambda calculus in order to implement ad-hoc features is not a new idea. For instance, Haskell overloading is generally implemented by passing "method dictionaries" as hidden parameters to polymorphic operations, in much the same way I pass tags. Ohori has shown that a primitive form of record-subtyping can be implemented by passing index mappings into polymorphic functions [16]. Tolmach [23] and I have independently shown that copying garbage collection can be implemented using this approach.

Leroy's boxing/unboxing technique for SML [13] is another closely related technique to separated tagging. Leroy suggests compiling code as if polymorphic components are in the "universal" representation (i.e. boxed). At the *use* of a polymorphic value, the value is converted to the unboxed representation at the appropriate type. The conversion requires pulling apart the object, recursively converting the components, and then putting the object back together.

Leroy's approach does not require a restriction like the value restriction for separated tagging and works in the presence of separate compilation. However, the approach does not work for mutable values and appears to be practical only for flattening small tuples, since it relies on copying and modifying the representation of a value. Finally, Leroy's approach does not give complete freedom in representation choices. That is, it is not possible to provide objects that are always flattened.

The separated tagging approach can be seen as a dual of Leroy's. Leroy eagerly transforms polymorphic values into monomorphic values at their uses. Separated tagging lazily transforms polymorphic

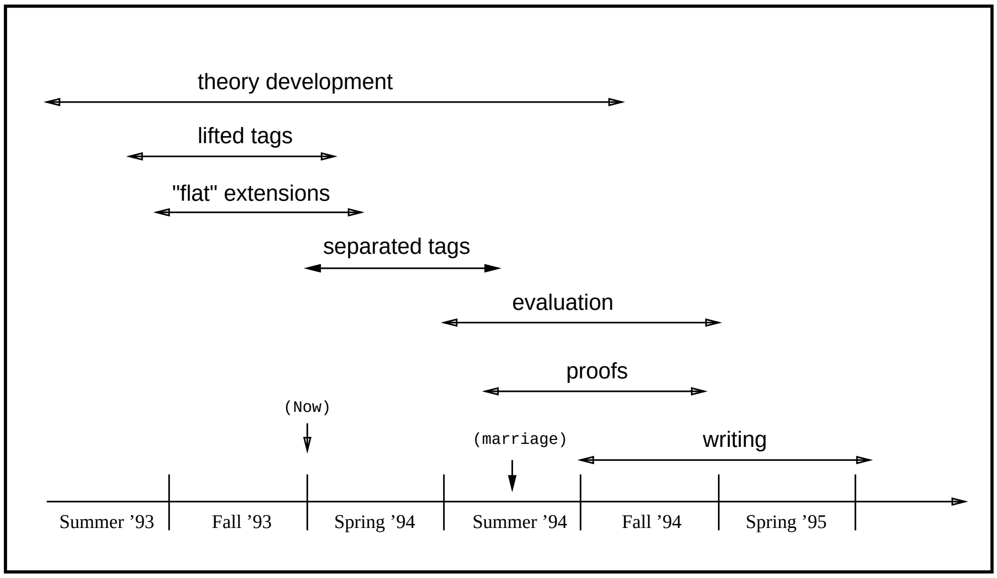

Figure 4: Schedule

values by transforming the primitive operations that manipulate the values. Since operations are transformed instead of values, this technique places no restrictions on the representations that a value can have.

Poulsen describes an extension of Leroy's approach that can be used to specialize more representations, such as lists [18]. An escape analysis is needed to determine whether a value's representation can be specialized. As with other restrict and eliminate techniques, this approach does not work in the presence of separate compilation.

Collberg describes practical techniques for eliminating polymorphism that retain the appearance of separate compilation [6]. Like others, he delays compilation of polymorphic objects until link time. However, he concentrates on reducing the time spent in compilation at link-time by using parallel and incremental re-compilation techniques. Unfortunately, he assumes that the language has been designed so that all polymorphism can be eliminated at link time, so his techniques do not cover ML-style polymorphism.

7 Expected Contributions, and Schedule

The expected contributions of this work are:

- Techniques (lifted and separated tagging) for implementing polymorphic languages that do not restrict representations nor separate compilation.
- Intermediate language framework that provides formalization of representation techniques (i.e. tagging, boxing, etc.)
- Proofs of correctness and other interesting properties regarding these techniques.
- Quantitative evaluation of these techniques versus each other and previous techniques in an existing system (SML/NJ).
- Extensions to SML to support user-controlled data representations.
- Recommendations to language designers regarding the interactions between polymorphism and data representations.

A schedule for the thesis work is in Figure 4.

$$
\begin{array}{rcll}
\tau & ::= & \alpha \mid & \text{type variables} \\
 & & \mathsf{int} \mid & \text{integers} \\
 & & \tau_1 \rightarrow \tau_2 \mid & \text{functions} \\
 & & \tau_1 * \tau_2 & \text{tuples} \\
 & & & \\
\sigma & ::= & \forall \alpha_1, \ldots, \alpha_n . \tau & \text{type schemes}
\end{array}
$$

$$
\begin{array}{rcll}
e & ::= & x \mid & \text{identifier} \\
 & & i \mid & \text{integer literal} \\
 & & \mathsf{fn}\ x \mathrel{=>} e \mid & \text{function} \\
 & & e_1\ e_2 \mid & \text{application} \\
 & & \mathsf{let\ val}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end} \mid & \text{let} \\
 & & (e_1, e_2) \mid & \text{tuple} \\
 & & \pi_i\ e & \text{projection}
\end{array}
$$

$$
v \quad ::= \quad x \mid i \mid \mathsf{fn}\ x \mathrel{=>} e \mid (v_1, v_2) \qquad \text{values}
$$

Figure 5: Mini-ML Abstract Syntax

A Example: Flattening Tuples

In this appendix I show how a simple polymorphic language, Mini-ML, with arbitrary tuples can be translated to an intermediate language, Pre-ML, with only "flat" tuples. By "flat", I mean that the intermediate language does not allow a nested tuple in the first component of the tuple.

The source language is described by the definitions of Figure 5. The static semantics (type system) of Mini-ML is given by Figure 6 as a series of typing-judgements of the form $\Gamma \vdash e : \tau$, where Γ is a type environment mapping identifiers to type schemes, e is an expression, and τ is a type. Note that I impose the value restriction upon this language (see Section 4.2) by only generalizing let-bound value expressions. Type generalization is expressed in terms of a $\mathsf{close}_\Gamma(\tau)$ function, which quantifies type variables free in τ but not free in Γ. Type scheme instantiation is expressed in terms of a partial function, ρ, from type variables to types.

The dynamic semantics of Mini-ML is given by Figure 7 as a set of inference rules of the form $e \hookrightarrow v$ where e is an expression and v is a non-identifier value. I use $e[v/x]$ to denote capture-avoiding substitution of the value v for the identifier x in the expression e.

Pre-ML is the same as Mini-ML except that values are restricted so that the first component of a tuple cannot contain a tuple:

$$
\begin{array}{rcll}
v^\flat & ::= & x \mid i \mid \mathsf{fn}\ x \mathrel{=>} e & \text{flat values} \\
 & & & \\
v & ::= & v^\flat \mid (v^\flat, v) & \textit{values}
\end{array}
$$

The translation from Mini-ML to Pre-ML is given by the set of rules in Figure 9 and makes use of the type derivation of the Mini-ML expression. It is essentially a separated tagging implementation (hence the value restriction.) However, instead of passing tags into polymorphic definitions, methods for the instantiated type are passed in. The methods for a particular type are defined in Figure 8.

The following theorem states that the translation is meaning preserving:

Theorem 1 *If $\Gamma \vdash e : \tau \Rightarrow e'$ and $e \hookrightarrow v$, then $\Gamma \vdash v : \tau \Rightarrow v'$ and $e' \hookrightarrow v'$.*

The theorem can be proven by induction on the derivation of $e \hookrightarrow v$.

$$\frac{\Gamma(x) = \forall\alpha_1,\ldots,\alpha_n.\tau \qquad \{\alpha_1,\cdots,\alpha_n\} \subseteq \mathsf{Dom}(\rho)}{\Gamma \vdash x : \rho(\tau)} \qquad\qquad \frac{}{\Gamma \vdash i : \mathsf{int}}$$

$$\frac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \mathsf{fn}\ x \mathrel{=>} e : \tau_1 \rightarrow \tau_2}\quad (x \notin \mathsf{Dom}(\Gamma)) \qquad\qquad \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}$$

$$\frac{\Gamma \vdash v : \tau_1 \qquad \sigma = \mathsf{close}_\Gamma(\tau_1) \qquad \Gamma[x \mapsto \sigma] \vdash e : \tau_2}{\Gamma \vdash \mathsf{let\ val}\ x = v\ \mathsf{in}\ e\ \mathsf{end} : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1,e_2) : \tau_1 * \tau_2} \qquad\qquad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash \pi_i\, e : \tau_i}\quad (1 = 1, 2)$$

Figure 6: Mini-ML Static Semantics

$$\frac{}{i \hookrightarrow i} \qquad\qquad \frac{}{\mathsf{fn}\ x \mathrel{=>} e \hookrightarrow \mathsf{fn}\ x \mathrel{=>} e}$$

$$\frac{e_1 \hookrightarrow \mathsf{fn}\ x \mathrel{=>} e \qquad e_2 \hookrightarrow v' \qquad e[v'/x] \hookrightarrow v}{e_1\, e_2 \hookrightarrow v} \qquad\qquad \frac{e_1 \hookrightarrow v' \qquad e_2[v'/x] \hookrightarrow v}{\mathsf{let\ val}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end} \hookrightarrow v}$$

$$\frac{e_1 \hookrightarrow v_1 \qquad e_2 \hookrightarrow v_2}{(e_1,e_2) \hookrightarrow (v_1,v_2)} \qquad\qquad \frac{e \hookrightarrow (v_1,v_2)}{\pi_i\, e \hookrightarrow v_i}\quad (i = 1, 2)$$

Figure 7: Mini-ML Dynamic Semantics

$$\begin{aligned}
\mathsf{cons}_\alpha &= \pi_1\, \alpha \\
\mathsf{cons}_{\mathsf{int}} &= \mathsf{fn}\ x \mathrel{=>} \mathsf{fn}\ y \mathrel{=>} (x,y) \\
\mathsf{cons}_{(\tau_1 \rightarrow \tau_2)} &= \mathsf{fn}\ x \mathrel{=>} \mathsf{fn}\ y \mathrel{=>} (x,y) \\
\mathsf{cons}_{(\tau_1 * \tau_2)} &= \mathsf{fn}\ x \mathrel{=>} \mathsf{fn}\ y \mathrel{=>} \mathsf{cons}_{\tau_1}(\mathsf{car}_{\tau_1}\ x)(\mathsf{cons}_{\tau_2}(\mathsf{cdr}_{\tau_1} x)\ y)
\end{aligned}$$

$$\begin{aligned}
\mathsf{car}_\alpha &= \pi_1\, (\pi_2\, \alpha) \\
\mathsf{car}_{\mathsf{int}} &= \mathsf{fn}\ x \mathrel{=>} \pi_1\, x \\
\mathsf{car}_{(\tau_1 \rightarrow \tau_2)} &= \mathsf{fn}\ x \mathrel{=>} \pi_1\, x \\
\mathsf{car}_{(\tau_1 * \tau_2)} &= \mathsf{fn}\ x \mathrel{=>} \mathsf{cons}_{\tau_1}(\mathsf{car}_{\tau_1}\ x)(\mathsf{car}_{\tau_2}(\mathsf{cdr}_{\tau_1}\ x))
\end{aligned}$$

$$\begin{aligned}
\mathsf{cdr}_\alpha &= \pi_2\, (\pi_2\, \alpha) \\
\mathsf{cdr}_{\mathsf{int}} &= \mathsf{fn}\ x \mathrel{=>} \pi_2\, x \\
\mathsf{cdr}_{(\tau_1 \rightarrow \tau_2)} &= \mathsf{fn}\ x \mathrel{=>} \pi_2\, x \\
\mathsf{cdr}_{(\tau_1 * \tau_2)} &= \mathsf{fn}\ x \mathrel{=>} \mathsf{cdr}_{\tau_2}(\mathsf{cdr}_{\tau_1}\ x)
\end{aligned}$$

$$\mathsf{methods}(\tau) = (\mathsf{cons}_\tau,(\mathsf{car}_\tau,\mathsf{cdr}_\tau))$$

Figure 8: Type Methods

$$\frac{\Gamma(x) = \forall \alpha_1, \ldots, \alpha_n.\tau \qquad \{\alpha_1, \cdots, \alpha_n\} \subseteq \mathsf{Dom}(\rho) \quad \mathsf{methods}(\rho(\alpha_1)) = e_1 \qquad \cdots \qquad \mathsf{methods}(\rho(\alpha_n)) = e_n}{\Gamma \vdash x : \rho(\tau) \Rightarrow x\ e_1\ \cdots\ e_n}$$

$$\frac{}{\Gamma \vdash i : \mathsf{int} \Rightarrow i}$$

$$\frac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2 \Rightarrow e'}{\Gamma \vdash \mathsf{fn}\ x \texttt{=>} e : \tau_1 \rightarrow \tau_2 \Rightarrow \mathsf{fn}\ x \texttt{=>} e'} \quad (x \notin \mathsf{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \Rightarrow e_1' \qquad \Gamma \vdash e_2 : \tau_1 \Rightarrow e_2'}{\Gamma \vdash e_1\, e_2 : \tau_2 \Rightarrow e_1'\, e_2'}$$

$$\frac{\begin{array}{c}\Gamma \vdash v : \tau_1 \Rightarrow v' \\ \sigma = \mathsf{close}_\Gamma(\tau_1) = \forall \alpha_1, \ldots, \alpha_n.\tau_1 \\ \Gamma[x \mapsto \sigma] \vdash e : \tau_2\end{array}}{\Gamma \vdash \mathsf{let\ val}\ x = v\ \mathsf{in}\ e\ \mathsf{end} : \tau_2 \Rightarrow \mathsf{let\ val}\ x = \mathsf{fn}\ \alpha_1 \texttt{=>}\ \cdots\ \mathsf{fn}\ \alpha_n \texttt{=>} v'\ \mathsf{in}\ e'\ \mathsf{end}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \Rightarrow e_1' \qquad \Gamma \vdash e_2 : \tau_2 \Rightarrow e_2'}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2 \Rightarrow \mathsf{cons}_{\tau_1}\ e_1'\ e_2'}$$

$$\frac{\Gamma \vdash e : \tau_1 * \tau_2 \Rightarrow e'}{\Gamma \vdash \pi_1\, e : \tau_1 \Rightarrow \mathsf{car}_{\tau_1}\ e'}$$

$$\frac{\Gamma \vdash e : \tau_1 * \tau_2 \Rightarrow e'}{\Gamma \vdash \pi_2\, e : \tau_2 \Rightarrow \mathsf{cdr}_{\tau_1}\ e'}$$

Figure 9: Translating Mini-ML to Pre-ML

References

[1] A. Aiken, E. L. Wimmers, and T. Lakshman. Soft typing with conditional types. In *Proceedings of the Twenty-First Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 1994. To appear.

[2] A. W. Appel. Runtime tags aren't necessary. *Journal of Lisp and Symbolic Computation*, 2:153–162, 1989.

[3] E. Biagioni, R. Harper, and P. Lee. Standard ML signatures for a protocol stack. Technical Report CMU-CS-93-170, School of Computer Science, Carnegie Mellon University, Oct. 1993.

[4] G. E. Blelloch, S. C. J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego*, pages 102–111, May 1993.

[5] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 278–292, June 1991.

[6] C. S. Collberg. *Flexible Encapsulation*. PhD thesis, Department of Computer Science, Lund University, Sweden, 1992.

[7] J.-Y. Girard. Une extension de l'interpretation de Gödel à l'analyse, et son application a l'elimination des coupures dans l'analyse et la théorie des types. In *Proceedings of the Second Scandinavian Logic Symposium, edited by J.E. Fenstad. North-Holland, Amsterdam*, pages 63–92, 1971.

[8] J.-Y. Girard. *Interprétation Fonctionnelle et Elimination des Coupures dans l'Arithmétique d'Ordre Supérieur*. PhD thesis, Université Paris VII, 1972.

[9] B. Goldberg and M. Gloger. Polymorphic type reconstruction for garbage collection without tags. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 53–65, June 1992.

[10] J. Greiner. Standard ML weak polymorphism can be sound. Technical Report CMU-CS-93-160R, School of Computer Science, Carnegie Mellon University, Sept. 1993.

[11] F. Henglein. Global tagging optimization by type inference. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 205–215, June 1992.

[12] M. Hoang, J. Mitchell, and R. Viswanathan. Standard ML weak polymorphism and imperative constructs. In *Eighth Symposium on Logic in Computer Science*, pages 15–25. IEEE Computer Society Press, June 1993.

[13] X. Leroy. Unboxed objects and polymorphic typing. In *Proceedings of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 177–188, Jan. 1992.

[14] D. MacQueen. Source code for SML/NJ type inference algorithm. Avaialable via anon. ftp from research.att.com:/dist/ml.

[15] R. Morrison, A. Dearle, R. C. H. Connor, and A. L. Brown. An ad hoc approach to the implementation of polymorphism. *ACM Trans. Prog. Lang. Syst.*, 13(3):342–371, July 1991.

[16] A. Ohori. A compilation method for ML-style polymorphic record calculi. In *Proceedings of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 154–165, Jan. 1992.

[17] F. Pfenning. Elf: A language for logic definition and verified meta-programming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322. IEEE Computer Society Press, June 1989.

[18] E. R. Poulsen. Representation analysis for efficient implementation of polymorphism. Technical report, Department of Computer Science (DIKU), University of Copenhagen, Apr. 1993. Master Dissertation.

[19] J. C. Reynolds. Towards a theory of type structure. In *Proceedings, Colloque sur la Programmation. Lecture Notes in Computer Science*, volume 19, pages 408–425. Springer-Verlag, Berlin, 1974.

[20] C. Strachey. Fundamental concepts in programming languages. In *Lecture Notes for International Summer School in Computer Programming, Copenhagen*. Aug. 1967.

[21] D. Tarditi, A. Diwan, and E. Moss. Memory subsystem performance of programs using copying garbage collection. In *Conference Record of the 21st Annual ACM Conference on Principles of Programming Languages*, pages 1–14, Jan. 1994.

[22] M. Tofte. Type inference for polymorphic references. *Information and Computation 89*, 1:1–34, Nov. 1990.

[23] A. Tolmach. Tag-free garbage collection using explicit type parameters. Submitted to 1993 ACM Conference on Lisp and Functional Programming, Nov. 1993.

[24] United States Department of Defense. *Reference Manual for the Ada Programming Language*, Feb. 1983. U.S. Government Printing Office, ANSI/MIL-STD-1815A-1983.

[25] A. K. Wright. Polymorphism for imperative languages without imperative types. Technical Report TR93-200, Department of Computer Science, Rice University, Feb. 1993.