Standard ML of New Jersey

Andrew W. Appel* Princeton University

David B. MacQueen AT&T Bell Laboratories

CS-TR-329-91, Dept. of Computer Science, Princeton University, June 1991

This paper appeared in Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming, Springer-Verlag LNCS 528, pp. 1–13, August 1991.

Abstract

The Standard ML of New Jersey compiler has been under development for five years now. We have developed a robust and complete environment for Standard ML that supports the implementation of large software systems and generates efficient code. The compiler has also served as a laboratory for developing novel implementation techniques for a sophisticated type and module system, continuation based code generation, efficient pattern matching, and concurrent programming features.

1 Introduction

Standard ML of New Jersey is a compiler and programming environment for the Standard ML language[26] that has been continuously developed since early 1986. Our initial goal was to produce a working ML front end and interpreter for programming language research, but the scope of the project has expanded considerably. We believe that Standard ML may be the best general-purpose programming language yet developed; to demonstrate this, we must provide high-quality, robust, and efficient tools for software engineering.

Along the way we have learned many useful things about the design and implementation of "modern" programming languages. There were some unexpected interactions between the module system, type system, code generator, debugger, garbage collector, runtime data format, and hardware; and some things were much easier than expected. We wrote an early description of the compiler in the spring of 1987[7], but almost every component of the compiler has since been redesigned and reimplemented at least once, so it is worthwhile to provide an updated overview of the system and our implementation experience.

Our compiler is structured in a rather conventional way: the input stream is broken into tokens by a lexical analyzer, parsed according to a context-free grammar, semantically analyzed into an annotated abstract syntax tree, type-checked, and translated into a lower-level intermediate language. This is the "front end" of the compiler. Then the intermediate language—*Continuation-Passing Style*—is "optimized," closures are introduced to implement lexical scoping, registers are allocated, target-machine instructions are generated, and (on RISC machines) instructions are scheduled to avoid pipeline delays; these together constitute the "back end."

2 Parsing

Early in the development of the compiler we used a hand-written lexical analyzer and a recursive-descent parser. In both of these components the code for semantic analysis was intermixed with the parsing code. This made error recovery difficult, and it was difficult to understand the syntax or semantics individually. We now have excellent tools[8, 32] for the automatic generation of lexical analyzers and error-correcting parsers. Syntactic error recovery is handled automatically by the parser generator, and semantic actions are only evaluated on correct (or corrected) parses. This has greatly improved both the quality of the error messages and the robustness of the compiler on incorrect inputs. We remark that it would have been helpful if the definition of Standard ML[26] had included an LR(1) grammar for the language.

There are two places in the ML grammar that appear not to be context free. One is the treatment of data constructors: according to the definition, constructor names are in a different lexical class than variable names, even though the distinction depends on the semantic analysis of previous `datatype` definitions. However, by putting constructors and variables into the same class of lexical tokens, and the same name space, parsing can be done correctly and the difference resolved in semantic analysis.

The other context-dependent aspect of syntax is

*Supported in part by NSF grant CCR-9002786.

the parsing of `infix` identifiers. ML allows the programmer to specify any identifier as infix, with an operator precedence ranging from 0 to 9. Our solution to this problem is to completely ignore operator precedence in writing our LALR(1) grammar; the expression $a+b*c$ is parsed into the list $[a,+,b,*,c]$ and the semantic analysis routines include a simple operator precedence parser (35 lines of ML).

Each production of our grammar is annotated by a semantic action, roughly in the style made popular by YACC[16]. Our semantic actions are written like a denotational semantics or attribute grammar, where each fragment is a function that takes inherited attributes as parameters and returns synthesized attributes as results. Within the actions there are occasional side-effects; e.g. when the type-checker performs unification by the modification of `ref`-cells.

A complete parse yields a function p parameterized by a static environment e (of identifiers defined in previous compilation units, etc.). No side-effects occur until p is applied to e, at which point e is distributed by further function calls to many levels of the parse tree. In essence, before p is applied to e it is a tree of closures (one pointing to the other) that is isomorphic to the *concrete* parse tree of the program. Yet we have not had to introduce a myriad of data constructors to describe concrete parse trees!

Delaying the semantic actions is useful to the error-correcting parser. If an error in the parse occurs, the parser might want to correct it at a point 10 tokens previous; this means discarding the last few semantic actions. Since the actions have had no side-effects, it is easy to discard them. Then, when a complete correct parse is constructed, its semantic value can be applied to the environment e and all the side-effects will go off in the right order.

Finally, the treatment of mutually-recursive definitions is easier with delayed semantic actions; the newly-defined identifiers can be entered into the environment before the right-hand-sides are processed.

There is one disadvantage to this arrangement. It turns out that the closure representation of the concrete parse tree is much larger than the annotated parse tree that results from performing the semantic actions. Thus, if we had used a more conventional style in which the actions are performed as the input is parsed, the compiler would use less memory.

Our parser-generator provides, for each nonterminal in the input, the line number (and position within the line) of the beginning and end of the program fragment corresponding to that nonterminal. These are used to add accurate locality information to error messages. Furthermore, these line numbers are sprinkled into the annotated abstract syntax tree so that the type checker, match compiler, and debugger can also give good diagnostics.

3 Semantic analysis

A *static environment* maps each variable of the program to a *binding* containing its *type* and its runtime *access* information. The type is used for compile-time type checking, and is not used at runtime. The access information is (typically) the name of a low-level λ-calculus variable that will be manipulated by the code generator. Static environments also map other kinds of identifiers—data constructors, type constructors, structure names, etc.—to other kinds of bindings.

Our initial implementation treated environments imperatively: the operations on environments were to add a new binding to the global environment; to "mark" (save) the state of the environment; to revert back to a previous mark; and, for implementation of the module system, to encapsulate into a special table everything added since a particular mark. We did this even though we knew better—denotational semantics or attribute grammars would have us treat environments as pure values, to be combined to yield larger environments—because we thought that imperative environments would be faster.

We have recently changed to a pure functional style of environments, in which the operations are to create an environment with a single binding, and to layer one environment on top of another nondestructively, yielding a new environment. The implementation of this abstract data type has side effects, as sufficiently large environment-values are represented as hash tables, etc. We made this change to accommodate the new debugger, which must allow the user to be in several environments simultaneously; and to allow the implementation of "make" programs, which need explicit control over the static environments of the programs being compiled. Though we were willing to suffer a performance degradation in exchange for this flexibility, we found "pure" environments to be just as fast as imperative ones.

This illustrates a more general principle that we have noticed in ML program development. Many parts of the compiler that we initially implemented in an imperative style have been rewritten piecemeal in a cleaner functional style. This is one of the advantages of ML: programs (and programmers) can migrate gradually to "functional" programming.

Type checking

The main type-checking algorithm has changed relatively little since our earlier description[7]. The representations of types, type constructors, and type variables have been cleaned up in various ways, but the basic algorithm for type checking is still based on a straightforward unification algorithm.

The most complex part of the type-checking algorithm deals with *weak* polymorphism, a restricted form of polymorphism required to handle mutable values (references and arrays), exception transmission, and communication (in extensions like Concurrent ML[28]). Standard ML of New Jersey implements a generalization of the imperative type variables described in the Definition[26, 34]. In our scheme, *imperative* type variables are replaced by *weak* type variables that have an associated degree of weakness: a nonnegative integer. A type variable must be weak if it is involved in the type of an expression denoting a reference, and its degree of weakness roughly measures the number of function applications that must take place before the reference value is actually created. A weakness degree of zero is disallowed at top level, which insures that top-level reference values (*i.e.* those existing within values in the top level environment) have monomorphic types. The type-checking algorithm uses an abstract type *occ* to keep track of the "applicative context" of expression occurrences, which is approximately the balance of function abstractions over function applications surrounding the expression, and the *occ* value at a variable occurrence determines the weakness degree of generic type variables introduced by that occurrence. The *occ* value at a let binding is also used to determine which type variables can be generalized.

The weak typing scheme is fairly subtle and has been prone to bugs, so it is important that it be formalized and proven sound (as the Tofte scheme has been [Tofte-thesis]). There are several people currently working on formalizing the treatment used in the compiler[17, 38].

The weak polymorphism scheme currently used in Standard ML of New Jersey is not regarded as the final word on polymorphism and references. It shares with the imperative type variable scheme the fault that weak polymorphism propagates more widely than necessary. Even purely internal and temporary uses of references in a function definition will often "poison" the function, giving it a weak type. An example is the definition

```
fun f x = !(ref x)
```

in which f has the type $1\alpha \rightarrow 1\alpha$, but *ought* to have the strong polymorphic type $\alpha \rightarrow \alpha$. This inessential weak polymorphism is particularly annoying when it interferes with the matching of a signature specification merely because of the use of an imperative style within a function's definition. Such implementation choices should be invisible in the type. Research continues on this problem[17, 22, 38], but there is no satisfactory solution yet.

The interface between the type checker and the parser is quite simple in most respects. There is only one entry point to the type checker, a function that is called to type-check each value declaration at top level and within a structure. However, the interface between type checking and the parser is complicated by the problem of determining the scope or binding point of explicit type variables that appear in a program. The rather subtle scoping rules for these type variables[26, Section 4.6][25, Section 4.4] force the parser to pass sets of type variables both upward and downward (as both synthesized and inherited attributes of phrases). Once determined, the set of explicit type variables to be bound at a definition is stored in the abstract syntax representation of the definition to make it available to the typechecker.

4 Modules

The implementation of modules in SML of NJ has evolved through three different designs. The main innovation of the second version factored signatures into a symbol table shared among all instances, and a small instantiation environment for each instance[23]. Experience with this version revealed problems that led to the third implementation developed in collaboration with Georges Gonthier and Damien Doligez.

Representations

At the heart of the module system are the internal representations of signatures, structures, and functors. Based on these representations, the following principal procedures must be implemented:

1. signature creation—static evaluation of signature expressions;

2. structure creation—static evaluation of structure expressions;

3. signature matching between a signature and a structure, creating an *instance* of the signature, and a *view* of the structure;

4. definition of functors—abstraction of the functor body expression with respect to the formal parameter;

5. functor application—instantiation of the formal parameter by matching against the actual parameter, followed by instantiation of the functor body.

It is clear that instantiation of structure templates (i.e. signatures and functor bodies) is a critical process in the module system. It is also a process prone to consume excessive space and time if implemented naively. Our implementation has achieved reasonable efficiency by separating the *volatile* part of a template, that which changes with each instance, from the *stable* part that is common to all instances and whose representation may therefore be shared by all instances. The volatile components are stored in an instantiation environment and they are referred to indirectly in the bindings in the shared symbol table (or static environment) using indices or paths into the instantiation environment. The instantiation environment is represented as a pair of arrays, one for type constructor components, the other for substructures.

The static representation of a structure is essentially an environment (i.e., symbol table) containing bindings of types, variables, etc., and an identifying stamp[26, 33, 23]. In the second implementation a signature was represented as a "dummy" instance that differs from an ordinary structure in that its volatile components contain dummy or *bound* stamps and it carries some additional information specifying sharing constraints. The volatile components with their bound stamps are replaced, or *instantiated*, during signature matching by corresponding components from the structure being matched. Similarly, a functor body is represented as a structure with dummy stamps that are replaced by newly generated stamps when the functor is applied.

The problem with representing signatures (and functor bodies) as dummy structures with bound stamps is the need to do alpha-conversion at various points to avoid confusing bound stamps. To minimize this problem the previous implementation insures that the sets of bound stamps for each signature and functor body are disjoint. But there is still a problem with signatures and functors that are separately compiled and then imported into a new context; here alpha-conversion of bound stamps is required to maintain the disjointness property. Managing bound stamps was a source of complexity and bugs in the module system.

The usual way of avoiding the complications of bound variables is to replace them with an indexing scheme, as is done with deBruijn indices in the lambda calculus[13]. Since in the symbol table part we already used indices into instantiation arrays to refer to volatile components, we can avoid the bound stamps by using this *relativized* symbol table alone to represent signatures.

To drop the instantiation environment part of the signature representation, leaving only the symbol table part, we need to revise the details of how environments are represented. Formerly a substructure specification would be represented in the symbol table by a binding like

$$\texttt{A} \;\mapsto\; \texttt{INDstr}\; i$$

indicating that `A` is the ith substructure, and the rest of the specification of `A` (in the form of a dummy structure) would be found in the ith slot of the instantiation environment. Since we are dropping the dummy instantiation environment we must have all the information specifying `A` in the binding. Thus the new implementation uses

$$\texttt{A} \;\mapsto\; \texttt{FORMALstrb}\{\texttt{pos} = i, \texttt{spec} = \mathit{sig}_A\}$$

as the binding of `A`. This makes the substructure signature specification available immediately in the symbol table without having to access it indirectly through an instantiation environment.

Another improvement in the representation of signatures (and their instantiations) has to do with the *scope* of instantiation environments. In the old implementation each substructure had its own instantiation environment. But one substructure may contain relative references to a component of another substructure, as in the following example

```
signature S1 =
sig
  structure A : sig type t end
  structure B : sig val x : A.t end
end
```

Here the type of `B.x` refers to the first type component `t` of `A`. This would be represented from the standpoint of `B` as a relative path [*parent*, *first substructure*, *first type constructor*]. To accommodate these cross-structure references when each structure has a local instantiation environment, the first structure slot in the instantiation environment contains a pointer to the parent signature or structure. Defining and maintaining these parent pointers was another source of complexity, since it made the representation highly cyclical.

The new representation avoids this problem by having a single instantiation environment shared by the top level signature and all its *embedded* signatures. An embedded signature is one that is written "in-line" like the signatures of `A` and `B` in the example above. In the above example, the new representation of `A.t` within `B` is [*first type constructor*] since `A.t` will occupy the first type constructor slot in the shared instantiation environment.

A nonembedded signature is one that is defined at top level and referred to by name. The signature `S0` in the following example is a nonembedded signature.

```
signature S0 = sig type t end
signature S1 =
sig
  structure A : S0
  structure B : sig val x : A.t end
end
```

In this case the type `A.t` of `x` uses the indirect reference [*first substructure, first type constructor*] meaning the first type constructor in the local instantiation environment of `A`, which is the first structure component in the instantiation environment of `S1`. `S1` and `B` share a common instantiation environment because `B` is embedded in `S1`. But `S0`, the signature of `A`, is nonembedded because it was defined externally to `S1`. It therefore can contain no references to other components of `S1` and so it is given its own private instantiation environment having the configuration appropriate to `S0`.

Signature Matching

The goal of the representation of signatures is to make it easy to instantiate them via signature matching. A signature is a template for structures, and a structure can be obtained from the signature by adding an appropriate instantiation environment (and recursively instantiating any substructures with nonembedded signature specifications).

The signature matching process involves the following steps: (1) Create an empty instantiation environment of a size specified in the signature representation. (2) For each component of the signature, in the order they were specified, check that there is a corresponding component in the structure and that this component satisfies the specification. When this check succeeds it may result in an instance of a volatile component (e.g. a type constructor) that is entered into the new instantiation environment. (3) Finally, having created the instantiation structure, any sharing constraints in the signature are verified by "inspection."

Functors

The key idea is to process a functor definition to isolate volatile components of the result (those deriving from the parameter and those arising from generative declarations in the body) in an instantiation environment. Then the body's symbol table is *relativized* to this instantiation environment by replacing direct references by indirect paths. As in the case of signature matching, this minimizes the effort required to create an instance of the body when the functor is applied, because the symbol table information is inherited unchanged by the instance.

Defining a functor is done in three steps: (1) The formal parameter signature is *instantiated* to create a dummy parameter structure. (2) This dummy structure is bound to the formal parameter name in the current environment and the resulting environment is used to parse and type-check the functor body expression. If a result signature is specified the functor body is matched against it. (3) The resulting body structure is scanned for *volatile* components, identified by having stamps belonging to the dummy parameter or generated within the body, and references to these volatile components are replaced by indirect positional references into an instantiation environment.

The instantiation of the parameter signature must produce a structure that is *free* modulo the sharing constraints contained in the signature. In other words, it must satisfy the explicit sharing constraints in the signature and all implicit sharing constraints implied by them, but there must by no extraneous sharing. The algorithm used for this instantiation process is mainly due to George Gonthier and is vaguely related to linear unification algorithms. This instantiation process is also used to create structures declared as *abstractions* using the abstraction declaration of Standard ML of New Jersey (a nonstandard extension of the language).

Given this processing of the functor definition, functor application is now a fairly straightforward process. The actual parameter is matched with the formal parameter signature yielding an instantiation environment relative to the parameter signature. This is combined with a new instantiation environment generated for the functor body using freshly generated stamps in new volatile components.

5 Translation to λ-language

During the semantic analysis phase, all static program errors are detected; the result is an abstract parse tree annotated with type information. This is then translated into a *strict* lambda calculus augmented with data constructors, numeric and string constants, n-tuples, mutually-recursive functions; and various primitive operators for arithmetic, manipulation of `ref`s, numeric comparisons, etc. The translation into λ-language is the phase of our compiler that has changed least over the years.

Though the λ language has data constructors, it does not have pattern-matches. Instead, there is a very simple case statement that determines which constructor has been applied at top level in a given value. The pattern-matches of ML must be translated into discriminations on individual constructors. This is done as described in our previous paper[7], though Bruce Duba has revised the details of the algorithm.

The dynamic semantics of structures and functors are represented using the same lambda-language operators as for the records and functions of the core language. This means that the code generator and runtime system don't need to know anything about the module system, which is a great convenience.

Also in this phase we handle equality tests. ML allows any hereditarily nonabstract, nonfunctional values of the same type to be tested for equality; even if the values have polymorphic types. In most cases, however, the types can be determined at compile time. For equality on atomic types (like integer and real), we substitute an efficient, type-specific primitive operator for the generic equality function. When constructed datatypes are tested for equality, we automatically construct a set of mutually-recursive functions for the specific instance of the datatype; these are compiled into the code for the user's program. Only when the type is truly polymorphic—not known at compile time—is the general polymorphic equality function invoked. This function interprets the tags of objects at runtime to recursively compare bit-patterns without knowing the full types of the objects it is testing for equality.

Standard ML's polymorphic equality seriously complicates the compiler. In the front end, there are special "equality type variables" to indicate polymorphic types that are required to admit equality, and signatures have an `eqtype` keyword to denote exported types that admit equality. The eqtype property must be propagated among all types and structures that *share* in a functor definition. We estimate that about 7% of the code in the front end of the compiler is there to implement polymorphic equality.

The effect on the back end and runtime system is just as pernicious. Because ML is a statically-typed language, it should not be necessary to have type tags and descriptors on every runtime object (as Lisp does). The only reasons to have these tags are for the garbage collector (so it can understand how to traverse pointers and records) and for the polymorphic equality function. But it's possible to give the garbage collector a map of the type system[1], so that it can figure out the types of runtime objects without tags and descriptors. Yet the polymorphic equality function also uses these tags, so even with a sophisticated garbage collector they can't be done away with. (One alternative is to pass an equality-test function along with every value of an equality type, but this is also quite costly[36].)

Finally, the treatment of equality types in Standard ML is irregular and incomplete[15]. The *Definition* categorizes type constructors as either "equality" or "nonequality" type constructors; but a more refined classification would more accurately specify the effects of the `ref` operator. Some types that structurally support equality are classified as nonequality types by the *Definition*.

6 Conversion to CPS

The λ-language is converted into *continuation-passing style* (CPS) before optimization and code generation. CPS is used because it has clean semantic properties (like λ-calculus), but it also matches the execution model of a von Neumann register machine: variables of the CPS correspond closely to registers on the machine, which leads to very efficient code generaton[18].

In the λ-language (with side-effecting operators) we must specify a call-by-value (strict) order of evaluation to really pin down the meaning of the program; this means that we can't simply do arbitrary β-reductions (etc.) to partially evaluate and optimize the program. In the conversion to CPS, all order-of-evaluation information is encoded in the chaining of function calls, and it doesn't matter whether we consider the CPS to be strict or non-strict. Thus, β-reductions and other optimizations become much easier to specify and implement.

The CPS notation[30] and our representation of it[5] are described elsewhere, as is a detailed description of optimization techniques and runtime representations for CPS[4]. We will just summarize the important points here.

In continuation-passing style, each function can have several arguments (in contrast to ML, in which functions formally have only one parameter). Each of the actual parameters to a function must be *atomic*—a constant or a variable. The operands of an arithmetic operator must also be atomic; the result of the operation is bound to a newly-defined variable. There is no provision for binding the result of a function call to a variable; "functions never return."

To use CPS for compiling a programming language—in which functions are usually allowed to return results, and expressions can have nontrivial sub-expressions—it is necessary to use *continuations*. Instead of saying that a function call $f(x)$

returns a value a, we can make a function $k(a)$ that expresses what "the rest of the program" would do with the result a, and then call $f_{\mathrm{cps}}(x, k)$. Then f_{cps}, instead of returning, will call k with its result a.

After CPS-conversion, a source-language function call looks just like a source-language function return—they both look like calls in the CPS. This means it is easy to β-reduce the call without reducing the return, or vice versa; this kind of flexibility is very useful in reasoning about (and optimizing) tail-recursion, etc.

In a strict λ-calculus, β-reduction is problematical. If the actual parameters to a function have side effects, or do not terminate, then they cannot be safely substituted for the formal parameters throughout the body of the function. Any actual parameter expression could contain a call to an unknown (at compile time) function, and in this case it is impossible to tell whether it does have a side effect. But in CPS, the actual parameters to a function are always atomic expressions, which have no side effects and always terminate; so it's safe and easy to perform β-reduction and other kinds of substitutions.

In our optimizer, we take great advantage of a unique property of ML: records, n-tuples, constructors, etc., are *immutable*. That is, except for `ref` cells and arrays (which are identifiable at compile time through the type system), once a record is created it cannot be modified. This means that a *fetch* from a record will always yield the same result, even if the compiler arranges for it to be performed earlier or later than specified in the program. This allows much greater freedom in the partial evaluation of fetches (e.g. from pattern-matches), in constant-folding, in instruction scheduling, and common subexpression elimination than most compilers are permitted. (One would think that in a pure functional language like Haskell this *immutable record* property would be similarly useful, but such languages are usually *lazy* so that fetches from a lazy cell will yield different results the first and second times.)

A similar property of ML is that immutable records are not distinguishable by address. That is, if two records contain the same values, they are "the same;" the expressions

```
[(x,y), (x,y)]
let val a = (x,y) in [a,a] end
```

are indistinguishable in any context. This is not the case in most programming languages, where the different pairs `(x,y)` in the first list would have different addresses and could be distinguished by a pointer-equality test.

This means that the compiler is free to perform common sub-expression elimination on record expressions (i.e. convert the first expression above to the second); the garbage collector is free to make several copies of a record (possibly useful for concurrent collection), or to merge several copies into one (a kind of "delayed hash-consing"); a distributed implementation is free to keep separate copies of a record on different machines, etc. We have not really exploited most of these opportunities yet, however.

7 Closure conversion

The conversion of λ-calculus to CPS makes the control flow of the program much more explicit, which is useful when performing optimizations. The next phase of our compiler, *closure conversion*, makes explicit the access to nonlocal variables (using lexical scope). In ML (and Scheme, Smalltalk, and other languages), function definitions may be nested inside each other; and an inner function can have *free variables* that are bound in an outer function. Therefore, the representation of a function-value (at runtime) must include some way to access the values of these free variables. The *closure* data structure allows a function to be represented by a pointer to a record containing

1. The address of the machine-code entry-point for the body of the function.

2. The values of free variables of the function.

The code pointer (item 1) must be kept in a standardized location in all closures; for when a function f is passed as an argument to another function g, then g must be able to extract the address of f in order to jump to f. But it's not necessary to keep the free variables (item 2) in any standard order; instead, g will simply pass f's closure-pointer as an extra argument to f, which will know how to extract its own free variables.

This mechanism is quite old[19] and reasonably efficient. However, the introduction of closures is usually performed as part of machine-code generation; we have made it a separate phase that rewrites the CPS representation of the program to include closure records. Thus, the output of the closure-conversion phase is a CPS expression in which it is guaranteed that no function has free variables; this expression has explicit record-creation operators to build closures, and explicit fetch operators to extract code-pointers and free variables from them.

Since closure-introduction is not bundled together with other aspects of code generation, it is easier to introduce sophisticated closure techniques without breaking the rest of the compiler. In general, we have found that structuring our compiler with so many phases—each with a clean and well-defined interface—has proven very successful in allowing work to proceed independently on different parts of the compiler.

Initially, we considered variations on two different closure representations, which we call *flat* and *linked*. A *flat* closure for a function f is a record containing the code-pointer for f and the values of each of f's free variables. A *linked* closure for f contains the code pointer, the value of each free variable *bound by the enclosing function*, and a pointer to the enclosing function's closure. Variables free in the enclosing function can be found by traversing the linked list of closures starting from f; this is just like the method of *access links* used in implementing static scope in Pascal.

It would seem that linked closures are cheaper to build (because a single pointer to the enclosing scope can be used instead of all the free variables from that scope) but costlier to access (getting a free variable requires traversing a linked list). In fact, we investigated many different representational tricks on the spectrum between flat and linked closures[6], including tricks where we use the *same* closure record for several different functions *with several different code-pointers*[5, 4].

In a "traditional" compiler, these tricks make a significant difference. But in the CPS representation, it appears that the pattern of functions and variable access narrows the effective difference between these techniques, so that closure representation is not usually too important.

There are two aspects of closures that are important, however. We have recently shown that using linked or merged closures can cause a compiled program to use much more memory[4]. For example, a program compiled with flat closures might use $O(N)$ memory (i.e. simultaneous live data) on an input of size N, and the same program compiled with linked closures might use $O(N^2)$. Though this may happen rarely, we believe it is unacceptable (especially since the programmer will have no way to understand what is going on). We are therefore re-examining our closure representations to ensure "safety" of memory usage; this essentially means sticking to flat closures.

We have also introduced the notion of "callee-save registers."[9, 4] Normally, when an "unknown" function (e.g. one from another compilation unit) is called in a compiler using CPS, all the registers (variables) that will be needed "after the call" are *free variables of the continuation*. As such, they are stored into the continuation closure, and fetched back after the continuation is invoked. In a conventional compiler, the *caller* of a function might similarly save registers into the stack frame, and fetch them back after the call.

But some conventional compilers also have "callee-save" registers. It is the responsibility of each function to leave these registers undisturbed; if they are needed during execution of the function, they must be saved and restored by the *callee*.

We can represent callee-save variables in the original CPS language, without changing the code-generation interface. We will represent a continuation not as one argument but as $N+1$ arguments $k_0, k_1, k_2, \ldots, k_N$. Then, when the continuation k_0 is invoked with "return-value" a, the variables $k_1, \ldots, k_N$ will also be passed as arguments to the continuation.

Since our code generator keeps all CPS variables in registers—including function parameters—the variables $k_1, \ldots, k_N$ are, in effect, callee-save registers. We have found that $N = 3$ is sufficient to obtain a significant (7%) improvement in performance.

8 Final code generation

The operators of the CPS notation—especially after closure-conversion—are similar to the instructions of a simple register/memory von Neumann machine. The recent trend towards RISC machines with large register sets makes CPS-based code generation very attractive. It is a relatively simple matter to translate the closure-converted CPS into simple abstract-machine instructions; these are then translated into native machine code for the MIPS, Sparc, VAX, or MC68020. The latter two machines are not RISC machines, and to do a really good job in code generation for them we would have to add a final peephole-optimization or instruction-selection phase. On the RISC machines, we have a final instruction-scheduling phase to minimize delays from run-time pipeline interlocks.

One interesting aspect of the final abstract-machine code generation is the register allocation. After closure-conversion and before code generation we have a *spill* phase that rewrites the CPS expression to limit the number of free variables of any subexpression to less than the number of registers on the target machine[5, 4]. It turns out that very few functions require any such rewriting, especially on modern machines with 32 registers; five spills in 40,000 lines of code is typical.

Because the free variables of any expression are guaranteed to fit in registers, register allocation is a very simple matter: when each variable is bound, only K other variables are live (i.e. free in the continuation of the operation that binds the variable), where $K < N$, the number of registers. Thus, any of the remaining $N - K$ registers can be chosen to hold the new value.

The only place that a register-register **move** is ever required is at a procedure call, when the actual parameters must be shuffled into the locations required for the formal parameters. For those functions whose call sites are all evident to the compiler (i.e. those functions that are not passed as parameters or stored into data structures), we can choose the register-bindings for formal parameters to eliminate any **move**s in at least one of the calls[18]. By clever choices of which register to use for the bindings described in the last paragraph, we can almost eliminate any remaining register-register moves that might be required for the other procedure calls.

9 The runtime system

The absence of function returns means that a runtime stack is not formally required to execute programs. Although most CPS-based compilers introduce a runtime stack anyway[30, 18], we do not. Instead, we keep all closures (i.e. activation records) on the garbage-collected heap. This not only simplifies some aspects of our runtime system, but makes the use of first-class continuations (`call-with-current-continuation`) very efficient.

Because all closures are put on the heap, however, SML/NJ allocates garbage-collected storage at a furious rate: one 32-bit word of storage for every five instructions executed, approximately[4]. This means that the most important requirement for the runtime system is that it support fast storage allocation and fast garbage collection.

To make heap allocations cheap, we use a generational copying garbage collector[2] and we keep the format of our runtime data simple[3]. Copying collection is attractive because the collector touches only the live data, and not the garbage; we can arrange that almost all of a particular region of memory is garbage, then just a few operations can reclaim a very large amount of storage. Another advantage of copying collection is that the free area (in which to allocate new records) is a contiguous block of memory; it is easier to grab the first few words of this block than it would be to manage a "free list" of different-sized records.

Indeed, we keep pointers to the beginning and end of the free area in registers for fast access. Allocation and initialization of an n-word record requires n **store** instructions at different offsets from the free-space register, followed by the addition of a constant (the size of the new record) to the register. We perform allocations in-line (without a procedure call), and we use just one test for free storage exhaustion to cover all the allocations in a procedure (remember that in CPS, procedures don't have internal loops). Furthermore, we can perform this test in one single-cycle instruction by clever use of the overflow interrupt to initiate garbage collection[4].

Overall, garbage-collection overhead in Standard ML of New Jersey (with memory size equal to 5 times the amount of live data) is usually between 5 and 20%; this means that for each word of memory allocated, the amortized cost of collecting it is about 1/4 to 1 instruction. Thus, copying a data structure (reading it and writing a new copy) takes only two or three times as long as traversing it (examining all the fields). This encourages a more side-effect-free, functional style of programming.

In addition to the garbage collector, the runtime system provides an interface to operating system calls[3]. Higher-level services like buffered I/O are provided by a "standard library" written in Standard ML. There are also many C-language functions in the runtime system callable from ML; but we have not yet provided an easy interface for users to link their own foreign-language functions to be called from ML. Since the overhead for calling a C function is rather high, we have implemented half a dozen frequently-used functions (e.g. allocation of an array or a string) in assembly language.

There is also an ML interface to operating system signals[27] that uses the call/cc mechanism to bundle up the current state of execution into a "continuation;" to be resumed immediately, later (perhaps from another signal handler), never, or more than once.

A snapshot of the executing ML system may be written to a file; executing that file will resume execution just at the point where the snapshot was taken. It is also possible to remove the compiler from this snapshot, to build more compact standalone applications.

Our reliance on operating system signals for garbage collection, our direct connection to system calls, our snapshot-building utility, and other useful features of the runtime system have turned out to be quite operating-system dependent. This makes it hard to port the runtime system from one machine (and operating system) to another. Perhaps as different versions of Unix become more standardized (e.g. with System V/R4) these problems will largely disappear.

10 Performance

We had several goals for Standard ML of New Jersey:

- A complete and robust implementation of Standard ML.
- A compiler written in Standard ML itself, to serve as a test of ML for programming-in-the-large.
- A reasonably efficient compiler with no "bottlenecks."
- Very fast compiled code, competitive with "conventional" programming languages.
- A testbed for new ideas.

We believe we have achieved these goals. While our compiler has a few minor bugs (as does any large software system), they don't substantially detract from the usability of the system. We have found that ML is an excellent language for writing real programs. Our compiler's front end is quite carefully designed to be fast, but the back end needs (and is receiving) further work to make it compile faster. The quality of our compiled code is extremely good, as figures 1 and 2 show.

We tested Poly/ML[24] and SML/NJ on six real programs[4], whose average size was about 2000 nonblank noncomment lines of source. Figure 1 shows the results on a SparcStation 2 (the only modern platform on which they both run). Indeed, Poly/ML compiles about 43% faster (when it doesn't blow up); but SML/NJ programs run *five times faster* than Poly/ML programs, on the average (geometric mean). SML/NJ reportedly uses about 1.5 times as much heap space for execution[10]; and on a 68020-based platform (like a Sun-3), SML/NJ may not do relatively as well (since we don't generate really good code for that machine). So on obsolete machines with tiny memories, Poly/ML may do almost as well as SML/NJ.

Figure 2 compares implementations of several programming languages on a Knuth-Bendix benchmark. Standard ML of New Jersey does quite well, especially on the RISC machine (the DECstation 5000 has a MIPS processor).

	Poly/ML 1.91 Compile Time	Poly/ML 1.91 Run Time	SML/NJ 0.69 Compile Time	SML/NJ 0.69 Run Time
Life	10	128	13	27
Lex	41	95	66	20
Yacc	abort	—	531	10
Knuth-B	19	116	30	25
Simple	44	461	124	60
VLIW	abort	—	839	45

Figure 1: Comparison of Poly/ML and SML/NJ

This table shows compile time and run time in seconds of elapsed time for each benchmark on a SparcStation 2 with 64 megabytes of memory. SML/NJ was run with the optimization settings normally used for compiling the compiler itself, and with all the input in one file to enable cross-module optimization (which makes things about 9% faster). Note that the callee-save representation is not yet implemented for the Sparc and might save an additional 7% runtime. On two of the benchmarks (as shown), the Poly/ML compiler aborted after several minutes; we believe this is caused by complicated pattern-matches tripping over an exponential-time algorithm in the Poly/ML front end.

11 Continuations

One of the more significant language innovations in Standard ML of New Jersey is typed first-class continuations[14]. It turned out to be possible to add a major new capability to the language merely by introducing a new primitive type constructor and two new primitive functions. The signature for first-class continuations is:

```
type 'a cont
val callcc : ('a cont -> 'a) -> 'a
val throw : 'a cont -> 'a -> 'b
```

The type `int cont` is the type of a continuation that is expecting an integer value. The `callcc` function is similar to `call-with-current-continuation` or `call/cc` in Scheme — it is the primitive that captures continuation values. The function `throw` coerces a continuation into a function that can be applied to invoke the continuation. Since the invocation of a continuation does not return like a normal function call, the return type of `throw k` is a generic type variable that will unify with any type.

The runtime implementation of first-class continuations was also quite easy and very efficient, because of the use of continuation passing style in the code generation, and the representation of continuations as objects in the heap. Bundling up the current continuation into a closure is just like what is done on the call to an escaping function, and throwing a value to a continuation is like a function call. So continuations are as cheap as ordinary function

	Sun 3/280 16 Mbytes		DEC 5000/200 16 Mbytes	
	Run	G.C.	Run	G.C.
CAML V2-6.1	14.5	14.8	6.2	6.2
CAML Light 0.2	28.3		6.5	
SML/NJ 0.65	9.6	0.3	1.7	0.1
SML/NJ 0.65 x	8.5	0.3	1.4	0.1
LeLisp 15.23	4.1		1.4	
SunOS 3.5, cc -O	4.35			
gcc 1.37.1, gcc -O	4.22			
Ultrix 4.0, cc -O2			0.90	

Figure 2: Comparison of several different compilers

Xavier Leroy translated Gerard Huet's Knuth-Bendix program into several different languages, and ran them on two different machines[21]. This table shows non-GC run time and GC time in seconds for each version of the program. Since the program uses higher-order functions, Leroy had to do manual lambda-lifting to write the program in Lisp and C, and in some places had to use explicit closures (structures containing function-pointers).

CAML is a different version of the ML language (i.e. not Standard ML) developed at INRIA[11]; CAML V2-6.1 is a native-code compiler that shares the LeLisp runtime system, and CAML Light[20] is a compiler with a byte-code interpreter written in C. **SML/NJ x** refers to Standard ML of New Jersey with all modules placed in "super-module" to allow cross-module optimization.

calls.

Continuations are not necessarily a good tool for routine programming since they lend themselves to tricky and contorted control constructs. However, continuations have an important "behind the scenes" role to play in implementing useful tools and abstractions. They are used in the implementation of the interactive ML system to construct a barrier between user computation and the ML system. This makes it possible to export an executable image of a user function without including the ML compiler. Another application of continuations is Andrew Tolmach's replay debugger[35], where they are used to save control states. This is the basis of the *time travel* capabilities of the debugger.

It is well known that continuations are useful for implementing coroutines and for simulating parallel threads of control[37]. Using continuations in conjunction with the signal handling mechanisms implemented by John Reppy[27] (themselves expressed in terms of continuations), one can build light-weight process libraries with preemptive process scheduling entirely within Standard ML of New Jersey. Two major concurrency systems have been implemented at this point: Concurrent ML by John Reppy[28] is based on CCS/CSP-style primitives (synchronous communication on typed channels) but introduces the novel idea of *first-class events*. ML Threads is a system designed by Eric Cooper and Greg Morrisett[12] that provides mutual exclusion primitives for synchronization. A version of ML Threads runs on shared-memory multiprocessors, where threads can be scheduled to run in parallel on separate physical processors. Both Concurrent ML and ML Threads are implemented as ordinary ML modules, requiring no enhancements of the language itself—except that ML Threads required modification of the runtime system to support multiprocessing.

12 Related projects

A number of very useful enhancements of the Standard ML of New Jersey system are being carried out by other groups or individuals. One such project is the SML-to-C translator done by David Tarditi, Anurag Acharya, and Peter Lee at Carnegie Mellon[31]. This provides a very portable basis for running ML programs on a variety of hardware for which we do not yet have native code generators, with very respectable performance.

Mads Tofte and Nick Rothwell implemented the first version of separate compilation for Standard ML of New Jersey. Recently Gene Rollins at Carnegie Mellon has developed a more sophisticated and efficient system called SourceGroups for managing separate compilation. SourceGroups builds on the primitive mechanisms provided by Tofte and Rothwell but gains efficiency by doing a global analysis of dependencies among a set of modules and minimizing redundancy when loading or recompiling the modules.

John Reppy and Emden Gansner have developed a library for interacting with the X window system. This system is based on Concurrent ML and provides a much higher-level of abstraction for writing graphical interfaces than the conventional conventional C-based libraries.

13 Future Plans

The development of Standard ML of New Jersey and its environment is proceeding at an accelerating pace. John Reppy is implementing a new multi-generation, multi-arena garbage collector that should significantly improve space efficiency. Work is in progress to improve code generation and significantly speed up the back end. Exploratory work is being done on new features like type *dynamic*, extensible datatypes, and higher-order functors.

14 Acknowledgments

Many people have worked on Standard ML of New Jersey. We would like to thank **John H. Reppy** for many improvements and rewrites of the runtime system, for designing and implementing the signal-handling mechanism[27], improving the call/cc mechanism, designing the current mechanism for calls to C functions, implementing a sophisticated new garbage collector, generally making the runtime system more robust, and implementing the SPARC code generator; and for designing and implementing the Concurrent ML system[28] and its X-windows interface[29].

Thanks to **Trevor Jim** for helping to design the CPS representation[5]; and for implementing the match compiler and the original closure-converter, the original library of floating point functions, and the original assembly-language implementation of external primitive functions.

Thanks to **Bruce F. Duba** for improvements to the match compiler, the CPS constant-folding phase, the in-line expansion phase, the spill phase, and numerous other parts of the compiler; and for his part in the design of the call-with-current-continuation mechanism[14].

Thanks to **James W. O'Toole** who implemented the NS32032 code generator, and **Norman Ramsey** who implemented the MIPS code generator.

We thank **Andrew P. Tolmach** for the SML/NJ debugger[35], and for the new pure-functional style of static environments; and **Adam T. Dingle** for the debugger's Emacs interface.

Thanks to **James S. Mattson** for the first version of the ML lexical analyzer generator; and to **David R. Tarditi** for making the lexer-generator production-quality[8], for implementing a really first-class parser generator[32], for helping to implement the type-reconstruction algorithm used by the debugger[35], and for the the ML-to-C translator he implemented with **Anurag Acharya** and **Peter Lee**[31].

We appreciate **Lal George**'s teaching the code generator about floating point registers and making floating-point performance respectable; and his fixing of several difficult bugs not of his own creation. Thanks to **Zhong Shao** for the common-subexpression eliminator, as well as the callee-save convention that uses multiple-register continuations for faster procedure calls[9].

We thank **Nick Rothwell** and **Mads Tofte** for the initial implementation of the separate compilation mechanism, and **Gene Rollins** for his recent improvements.

Finally we thank our user community that sends us bug reports, keeps us honest, and actually finds useful things to do with Standard ML.

References

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

[2] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software—Practice and Experience*, 19(2):171–183, 1989.

[3] Andrew W. Appel. A runtime system. *Lisp and Symbolic Computation*, 3(343-380), 1990.

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

[5] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In *Sixteenth ACM Symp. on Principles of Programming Languages*, pages 293–302, 1989.

[6] Andrew W. Appel and Trevor T. Y. Jim. Optimizing closure environment representations. Technical Report 168, Dept. of Computer Science, Princeton University, 1988.

[7] Andrew W. Appel and David B. MacQueen. A Standard ML compiler. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture (LNCS 274)*, pages 301–324. Springer–Verlag, 1987.

[8] Andrew W. Appel, James S. Mattson, and David R. Tarditi. A lexical analyzer generator for Standard ML. distributed with Standard ML of New Jersey, December 1989.

[9] Andrew W. Appel and Zhong Shao. Callee-save registers in continuation-passing style. Technical Report CS-TR-326-91, Princeton Univ. Dept. of Computer Science, Princeton, NJ, June 1991.

[10] David Berry. SML resources. sent to the SML mailing list by db@lfcs.ed.ac.uk, May 1991.

[11] CAML: The reference manual (version 2.3). Projet Formel, INRIA-ENS, June 1987.

[12] Eric C. Cooper and J. Gregory Morrisett. Adding threads to Standard ML. Technical Report CMU-CS-90-186, School of Computer Science, Carnegie Mellon University, December 1990.

[13] N. G. deBruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Math.*, 34:381–392, 1972.

[14] Bruce Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ML. In *Eighteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 163–173, Jan 1991.

[15] Carl A. Gunter, Elsa L. Gunter, and David B. MacQueen. An abstract interpretation for ML equality kinds. In *Theoretical Aspects of Computer Software*. Springer, September 1991.

[16] S. C. Johnson. Yacc – yet another compiler compiler. Technical Report CSTR-32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[17] James William O'Toole Jr. Type abstraction rules for references: A comparison of four which have achieved noteriety. Technical Report 380, MIT Lab. for Computer Science, 1990.

[18] David Kranz. *ORBIT: An optimizing compiler for Scheme*. PhD thesis, Yale University, 1987.

[19] P. J. Landin. The mechanical evaluation of expressions. *Computer J.*, 6(4):308–320, 1964.

[20] Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report No. 117, INRIA, February 1990.

[21] Xavier Leroy. INRIA, personal communication, 1991.

[22] Xavier Leroy and Pierre Weis. Polymorphic type inference and assignment. In *Eighteenth Annual ACM Symp. on Principles of Prog. Languages*, Jan 1991.

[23] David B. MacQueen. The implementation of Standard ML modules. In *ACM Conf. on Lisp and Functional Programming*, pages 212–223, 1988.

[24] David C. J. Matthews. Papers on Poly/ML. Technical Report T.R. No. 161, Computer Laboratory, University of Cambridge, February 1989.

[25] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, Cambridge, Massachusetts, 1991.

[26] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Mass., 1989.

[27] John H. Reppy. Asynchronous signals in Standard ML. Technical Report TR 90-1144, Cornell University, Dept. of Computer Science, Ithaca, NY, 1990.

[28] John H. Reppy. Concurrent programming with events. Technical report, Cornell University, Dept. of Computer Science, Ithaca, NY, 1990.

[29] John H. Reppy and Emden R. Gansner. The eXene library manual. Cornell Univ. Dept. of Computer Science, March 1991.

[30] Guy L. Steele. Rabbit: a compiler for Scheme. Technical Report AI-TR-474, MIT, 1978.

[31] David R. Tarditi, Anurag Acharya, and Peter Lee. No assembly required: Compiling Standard ML to C. Technical Report CMU-CS-90-187, Carnegie Mellon Univ., November 1990.

[32] David R. Tarditi and Andrew W. Appel. ML-Yacc, version 2.0. distributed with Standard ML of New Jersey, April 1990.

[33] Mads Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, Edinburgh University, 1988. CST-52-88.

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

[35] Andrew P. Tolmach and Andrew W. Appel. Debugging Standard ML without reverse engineering. In *Proc. 1990 ACM Conf. on Lisp and Functional Programming*, pages 1–12, June 1990.

[36] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *Sixteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 60–76, Jan 1989.

[37] Mitchell Wand. Continuation-based multiprocessing. In *Conf. Record of the 1980 Lisp Conf.*, pages 19–28, August 1980.

[38] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical Report COMP TR91-160, Rice University, April 1991.