Adding “Overloading Polymorphism” to “C”

by

David Ziegler

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Bachelor of Independent Studies
in
Independent Studies

Waterloo, Ontario, Canada, 1992

Contents

Chapter 1

Introduction

In "C" programs, new functions are defined in terms of specific types and the functions that are already defined for those types. Often programmers want to provide the same function for several different types (e.g, functions to manipulate matrices of `int` and functions to manipulate matrices of `double`). "C" programmers have no choice but to write separate copies of each function for each type.

"C" functions can be made more general by parameterizing them by type and by operation. Unfortunately, in systems where these functions are used extensively—particularly as operation parameters to other such functions—it becomes complicated and tedious to supply the type and operation parameters explicitly. In this paper we propose a variant of "C"—called "Sea"—that has functions that are explicitly parameterized by type and operation. It then uses the type-inference/overload-resolution algorithm described in [Cormack 90] to automatically infer bindings for these type and operation parameters at the application site.

For example in "C" we can define a `square` function that takes a `double` argument, uses the `double` multiply operator and returns a `double` result. Using "Overloading Polymorphism" we can define a `square` function that operates on any type that has a multiply operator:

```
forall a : a square(a x, exists a op*(a,a))
{
  return x*x;
}

square(5)   -> 5.0
square(5.0) -> 25.0
```

The goals of this project are to integrate the “overloading polymorphism” type system with “C”, explore implementation issues by building a compiler, and then to experiment with the resulting language.

Chapter 2

Language Definition

As the integration of the type system with "C" progressed, it became apparent that, while it would be easy to design a new language that was mostly compatible with "C", it would take a major design effort to get every feature of the new language exactly compatible with "C"'s already complex language definition. Because we wanted to explore implementation issues, and because there is a separate, parallel project where the integration of a similar type system with "C" is a major focus [Ditchfield 92], we have elected to use a mostly compatible language definition. The structure of this document reflects this choice, with little attention paid to designing an exactly compatible language definition and considerable space devoted to the implementation and use of this style of language.

2.1 Overloading

All declarations of objects with "Sea" function type are overloaded. Declarations of any other type of object (including pointers to "Sea" functions) use "C"'s normal declaration system. There are only two contexts in which the "Sea" function type occurs 1) external "Sea" function declarations and 2) operation parameters. By restricting the language so that only "Sea" functions can be overloaded, we make it possible to transform "Sea" programs so that no run-time closures are required. (see chapter 3)

Two overloaded declarations with the exact same name and type interact in the same way as normal "C" declarations with the same name. Thus overloaded declarations can be supplied with prototypes, and can be re-declared at inner scopes.

Note that in “Sea”, unlike in “C++”, we can declare overloaded functions that differ only in their return type.

The declaration of a normal (non-overloaded) name at an inner scope hides all overloaded definitions for that name. This is necessary to prevent the addition of a new external overloaded name from breaking the internal operation of an already defined function. An overloaded declaration for a name that already has a normal declaration (either at the current or an outer scope) is an error.

Overload resolution is done by the type inferencer; the algorithm used will be discussed in another section of this paper.

2.2 Overloaded Polymorphic Functions

We call the set of operations defined for a type its “algebra” (from the language Russell [DD 85]).

Using overloaded function definitions we can build algebras for different types that have a common set of operations in a common form.

```
The algebra for int is:
    [int op*(int,int), int op+(int,int) ...]
The algebra for double is:
    [double op*(double, double), double op+(double,double)...]
```

Both `int` and `double` have a multiplication operator of the form “`t op*(t,t)`” where `t` is either `int` or `double`.

“Sea” polymorphic functions are parameterized by a set of type variables and a list of operation parameters. The type variables can be bound to any type as long as the algebras for these types contain the operations required to satisfy the operation parameters. Thus the algebra described by the operation parameters for each type variable must be a subset of the algebra for the type we are binding to the type variable. (We consider operation parameters involving multiple type variables as describing an operation that is required for each of the type variables.)

The following `square` function can be applied to a parameter of any type `a`, provided that `a`’s algebra contains “`[a op*(a,a)]`”.

```
forall a : a square(a x, exists a op*(a,a))
```

```
{
  return x*x;
}
```

The square function can be applied to both ints and doubles, so the square function extends the algebras for both ints and doubles.

```
The algebra for int is now:
    [int op*(int,int), int op+(int,int), int square(int) ...]
The algebra for double is now:
    [double op*(double, double), double op+(double,double),
       double square(double)...]
```

The specialized `square` function is now available to satisfy the requirements of another polymorphic function. Note that we stored a specialized version of `square` in the int and double algebras; in order to generate this specialized `square` we need some sort of closure mechanism (see chapter 3.)

The compiler never actually has a list of all the operations in an algebra. Such a list would often not be finite in length. Instead, the type inferencer recursively constructs algebras on demand in order to satisfy the requirements of polymorphic function specializations. (Either due to an application to actual arguments, or in an attempt to create an algebra to satisfy a specialization already under way.)

2.2.1 Syntax

The syntax for a polymorphic function declaration is as follows: (Simplified version-the syntax becomes considerably more complex after integration with all of "C"'s artifacts.)

```
"forall" typevarlist : return_type function_name(parm_or_exist_decl, ...)
```

`parm_or_exist_decl` is either a normal parameter declaration or:

```
"exists" function_declaration
```

The type parameter declaration list was put at the beginning of the function declaration so that the type parameters will be available for declaring the function return value.

2.3 Type Generators

Every type other than void has a base set of operations. (If a type has no operations then it is equivalent to void.) The built-in types are defined with arithmetic and logical operations. For user-defined types, each type constructor has an associated set of "access" operations. In "C" these access operations are not normal functions, and as such are not part of the algebra of a type.

We would like to be able to write polymorphic functions that could operate over all constructed types with a similar structure (i.e. all vectors). One way to do this would be to make these access functions available as normal functions.

Using this technique we could write a binary search routine that would operate on any type of vector:

```
forall elem,elem_ptr : int binary_search(elem_ptr x, int len,
                              exists elem op*(elem_ptr),
                              exists int compare(elem,elem));
```

Or a print list routine that operated on any type that had a `next` and a `data` field:

```
forall list_ptr, elem : print(list_ptr list_head,
                              exists list_ptr get_next(list_ptr),
                              exists elem get_data(list_ptr),
                              exists void print(elem));
```

This is not a very convenient way to write polymorphic data structure manipulation routines.

Instead, in "Sea", parameter types can be declared in terms of type constructors applied to type parameters. The type inferencer will make sure the structure of the argument type matches the structure of the parameter type. Because type constructors are used to build the parameter type, appropriate polymorphic access functions are automatically available. The above functions can now be rewritten as follows:

```
forall elem: int binary_search(elem *x, int len,
                               exists int compare(elem,elem));


forall elem : print( struct l { l *next; elem data; } *list_head,
                               exists void print(elem));
```

`Print` is not a valid "Sea" function because "C"/"Sea" do not recognise structurally equivalent records as being the same type. We could alter "Sea" to recognise structural equivalence. A better solution, which also saves us from having to repeat the entire type declaration whenever it is used, is parameterized types. We call these "type generators"; two generated types are equivalent if the parameters to the type generator are equivalent.

```
type list(type elem)
{
  return struct {
    list(elem) next;
    elem data; } *;
}

forall elem : print(list(elem) list_head, exists void print(elem));
```

Type generators use field layout conventions that are optimized for access by polymorphic functions. (See chapter 3.)

Extensible typegens, allowing for single inheritance, would be a useful addition.

2.4 Overloaded Operators

"C" has a large number of heavily overloaded built-in operators. It is desirable to treat these operators as if they were functions so that 1) overloaded polymorphic functions can be defined in terms of these operations, 2) the functionality represented by the operator symbols can be applied to new types and 3) the inferencer can be simplified by not having to incorporate special rules for each of the built-in operators.

It is not possible to directly translate all "C" infix operators into a function call form because many operators have definitions that cannot be accommodated using normal function call semantics.

The following operators can be translated directly:

```
a+b   -> op+(a,b)
a-b   -> op-(a,b)
a*b   -> op*(a,b)
```

```
a/b   -> op/(a,b)
a%b   -> op%(a,b)
a>>b  -> op>>(a,b)
a<<b  -> op<<(a,b)
a>b   -> op>(a,b)
a<b   -> op<(a,b)
a>=b  -> op>=(a,b)
a<=b  -> op<=(a,b)
a==b  -> op==(a,b)
a!=b  -> op!=(a,b)
a&b   -> op&(a,b)
a|b   -> op|(a,b)
a^b   -> op^(a,b)
-a    -> op-(a)
+a    -> op+(a)
~a    -> op~(a)
!a    -> op!(a)
```

The logical `&&` and `||` operators can be translated directly, but user versions will not provide short-circuit evaluation.

```
&&    -> op&&(a,b)
||    -> op||(a,b)
```

The array indexing operator returns an lvalue which is not possible for a normal "C" function. We translate `op[]` so that the definer is expected to return a pointer to the assignable value, and then we add a dereference operator to translate this into an lvalue.

```
a[b]  -> *op[](a,b)
```

Two transformations are done to the assignment operators 1) They are given the address of the parameter that they are expected to modify and 2) Rather than having the assignment function return the assigned value (as defined in "C") we have the compiler arrange for access to this value. This is an optimization to avoid the cost of returning a value that is usually ignored.

```
x=y   -> (op=(&x,y),val(x))
x*=y  -> (op*=(&x,y),val(x))
x/=y  -> (op/=(&x,y),val(x))
x%=y  -> (op%=(&x,y),val(x))
x+=y  -> (op+=(&x,y),val(x))
x-=y  -> (op-=(&x,y),val(x))
x<<=y -> (op<<=(&x,y),val(x))
x>>=y -> (op>>=(&x,y),val(x))
x&=y  -> (op&=(&x,y),val(x))
x^=y  -> (op^=(&x,y),val(x))
x|=y  -> (op|=(&x,y),val(x))
```

It should be noted that the system header files contain a fully polymorphic version of the `op=` operator, and that unless the user explicitly makes `op=` an operation parameter to a polymorphic function, the default fully polymorphic assignment operator will be used.

Increment and decrement operators require treatment similar to that given to assignment operators:

```
++x   -> (op++(&x),val(x))
x++   -> (oldx=x,op++(&val(x)),oldx)
--x   -> (op++(&x),val(x))
x--   -> (oldx=x,op++(&val(x)),oldx)
```

There is no functional form for the following operators:

```
&a               -> special
*a               -> special
sizeof(a)        -> special
sizeof(typename) -> special
(typename)a      -> special
a.fieldname      -> special
a->fieldname     -> special
a?b:c            -> special
a,b              -> special
```

"C" built-in operators are heavily overloaded. All the possible versions of all the overloadable built-in operators are described in a header file/library containing 1584 function definitions. Thus any variant of any of the built-in operators is available for binding to an operation parameter.

The following excerpt from the "sea.h" header file describes part of the behaviour of the "C" binary `+` operator.

```
extern int op+ (int, int) __builtin("op+");
extern double op+ (int, double) __builtin("op+");
extern double op+ (double, int) __builtin("op+");
extern double op+ (double, double) __builtin("op+");
forall a : extern a *op+ (short, a *) __builtin("op+");
forall a : extern a *op+ (a *, short) __builtin("op+");
forall a : extern a *op+ (int, a *) __builtin("op+");
forall a : extern a *op+ (a *, int) __builtin("op+");
```

The `__builtin` attribute tells the compiler that this function is equivalent to one of "C"'s standard operators, and that it should use its inline version if possible.

2.5 Type Inference/Overload Resolution Algorithm

The following algorithm has been adapted from the "ForceTwo" inference algorithm presented in [Cormack 90].

As input the inferencer takes a "Sea" expression tree in which leaf nodes are overloaded identifiers and interior nodes are function applications. Each step in the inference process involves the application of one parameter, so we represent the expression tree in curried form.

Each leaf node is translated into a set of possible types for that identifier. We then find the set of possible types for each application by unifying the set of possible types for the function parameter with the set of possible types for the actual parameter. During this pass `exist` parameters are not resolved, but instead they are promoted to the result type.

Each potential solution now has a list of the `exist` parameters that it requires. To resolve each `exist` parameter we 1) convert it into a normal parameter, 2) apply the variable that has the same name as the exist parameter to this parameter and then 3) recursively use the type-inference/overload-resolution algorithm to find a solution. Note that new exist parameters can be introduced by this process.

It is possible to construct expressions where we can never resolve all the exist parameters. For example:

```
forall a: void f(exists f(a*));
```

To prevent the inferencer from not terminating, the language has a built-in limit on the maximum size of an expression it will generate. While this is a major philosophical black mark on the inferencer algorithm, it is not expected to be significant limitation when compiling real programs.

For many expressions the inferencer will find several sets of bindings for type variables and overloaded identifiers that each result in a valid expression. In some circumstances it is possible to choose a best expression in a way that is both useful and predictable. If no such "best" solution can be found, the expression is considered ambiguous and the inferencer will fail.

Currently we only have one disambiguating rule: If both a monomorphic and a polymorphic function are applicable, then the monomorphic function will always be chosen. If an expression has multiple such applications, but no solution can be found that incorporates all of them, then the expression will be considered ambiguous.

If there are multiple polymorphic functions that can supply the same functionality then the algorithm will fail. This situation occurs often when we are designing "abstract" polymorphic data structures. (Polymorphic data structures for which all operations are provided through overloaded functions.) Often a single concrete data structure will have the basic operations necessary to be manipulated as one of several "abstract" data structures. If two of these abstract data structures have the same operation defined then ambiguity will result. For example a `sorted vector` has a `find` operation that does a binary search. `Sorted vector` also has interface routines defined so that it can be used wherever a `sorted sequence` is required. If we subsequently define a `find` operation for `sorted sequence`, in terms of the basic `sorted sequence` operations, then we will have two polymorphic `find` operations that can be applied to `sorted vectors`. Situations like these can usually be avoided by careful design of sub-algebra relationships.

Size of Inference Tree

At each step in the inference process, each of the (partial) possible solutions for the expression is represented as a separate node. For some expressions the number of possible solutions can be very large. For example, given the following function definitions:

```
char * f();
char ** f();
char *** f();
char **** f();
char ***** f();
char ****** f();
char ******* f();
char ******** f();
char ********* f();
char ********** f();

void q(char *, char *, char *, char *, char *,
       char *, char *, char *, char *, char *);

forall t1,t2,t3,t4,t5,t6,t7,t8,t9,t10 :
  void g(t1, t2, t3, t4, t5, t6, t7, t8, t9, t10,
         exists void q(t1, t2, t3, t4, t5, t6, t7, t8, t9, t10));
```

The expression:

```
g(f(), f(), f(), f(), f(), f(), f(), f(), f(), f());
```

has `10^10` possible solutions before we apply the exist parameter `q`.

This is clearly a manufactured example. In real programs large numbers of intermediate solutions are generated when we use functions that have many polymorphic definitions each of which is constrained only by its operation parameters. The inferencer does not bind operation parameters until after it is finished with the basic expression; up until this point all these polymorphic functions will be considered as possible solutions. For these situations, the size of the inference tree could be reduced considerably if we altered the inference algorithm to find (partial) bindings for `exist` parameters as they are introduced into the solutions.

Rather than expanding our inference tree until we are out of memory—which can result in very poor performance on machines with virtual memory—the compiler has a pre-set (machine-dependent) limit on the number of nodes it will allocate. On the machine which was used to develop the "Sea" compiler, we had access to at least 50Mb of real memory. Each inference node (including one type binding) uses about 200 bytes, so the inference tree was allowed to grow to 250000 nodes. It is not known whether this limit will be encountered often when compiling real programs. If a large number of nodes are actually required to compile real programs, then this machine-dependent limit will be a major portability problem: A program that compiles on a 25Mb computer may not compile on a 15Mb computer.

The inferencer could be re-designed to use a data structure where each possible solution does not have to be explicitly represented as a separate node.

If we restrict the language so that we ignore function return types when doing type-inference/overload-resolution then the inference tree will never grow very large. (This is what the designers of "C++" have elected to do.) While there are many examples of expressions that will not compile with this restriction in place, we do not have enough experience with "Sea" programming to know whether the restricted version would represent a useful and understandable language.

Chapter 3

The "Lake" Intermediate Language

"Sea" function declarations are overloaded and have explicit type and operation parameters. When these functions are used the "Sea" type inferencer will automatically select one of several overloaded versions and provide appropriate bindings for the type and operation parameters.

"Lake" is a language very similar to "Sea" but without overloading and with type and operation parameters explicit at both the declaration and the call site. The "Sea" type inferencer emits "Lake" code as its output.

Because "Lake" doesn't have overloading, the "Sea" compiler assigns unique names to overloaded functions and the "Sea" overload resolver can then resolve uses of an overloaded name to a specific unique name. The actual renaming process is fairly complicated in order to support traditional "C" style separate compilation; for this section unique names will be generated by adding a underscore followed by a unique number to the end of the overloaded name.

A "Sea" polymorphic function supplies a new operation to every type that has the required prerequisite operations. Before a polymorphic operation can be applied to a specific set of actual parameters, the type inferencer specializes it by binding its type and operation parameters. If this only happened just before a polymorphic function was applied, type and operation parameters could be passed along with the normal parameters. But in "Sea" we also specialize polymorphic functions in order to generate the operations needed to satisfy the "exist" requirements of other polymorphic functions. In order to allow type and operation parameters to be applied prior to the application of other parameters we borrow an idea from functional programming languages and describe all "Sea" functions as functions that take a tuple—containing type and operation parameters—as their only argument, and return a function that takes a tuple (the normal “C” argument list) to the final return type.

We use “Sea” declaration syntax to declare “Lake” functions both because the “Sea” syntax has been carefully designed to be compatible with normal “C” declaration syntax and because it makes it possible to translate a “Sea” program to a “Lake” program without structural changes to the source program. However using “Sea” syntax for “Lake” does introduce one ambiguity; the type for monomorphic functions that don't take any type or operation parameters, but still require an empty type and operation tuple, is indistinguishable from the type of a “Lake” function that has already had its type and operation tuple applied. To avoid this confusion, function types that still require a type and operation tuple will always be written using a “forall”, even if the forall list is empty.

The first argument to a “Lake” function—the type and operation tuple—can be applied anytime before the regular arguments tuple is applied. To visually distinguish these two tuples we use “<” and “>” to bracket the type and operation tuple, and normal brackets to surround the regular arguments tuple. For example:

Given the following implementation of `square` :

```
forall t : t square(t x, exists t op*(t,t))
{
  return x*x;
}
```

The “Sea” expression :

```
square(7)
```

Will get translated by the inferencer to the “Lake” expression :

```
square_1<int, op*_1<>>(7)
```

A more general version of this mechanism would allow for some type and operation parameters to be bound while others were left unbound. This is difficult and expensive (in terms of performance) to implement and no applications that could justify this cost were found.

3.1 Non-constant Closures

The "Sea" inferencer takes as input a expression tree consisting entirely of constants, non-overloaded identifiers and overloaded identifiers. "Sea" is statically scoped so the set of bindings for the overloaded and non-overloaded identifiers is a compile time constant and, because "Sea" is statically typed, the type of each of these identifiers is also a compile time constant. This is all the information needed by the type inferencer in order for inference to be done entirely at compile time.

While all the types and the set of operations are compile time constants, for forall and exist parameters the actual value is not constant. For example, within `square`, `x` has the type `a`, and `op*` has the type "`a (a,a)`". However, the attributes of type `a`, and the value of the `op*` parameter are not available at compile time. Non-constant closures are created when these type and operation parameters are used to construct a closure. For example in the following `fourth` function a non-constant `op*` and type `a` are used to build a closure for `square`.

```
forall a : a square(a x, exists a op*(a, a))
{
  return x*x;
}

forall a : a fourth(a x, exists a op*(a, a))
{
  return square (square (x));
}

int f()
{
  return fourth(7);
}
```

is translated by the inferencer to :

```
forall a : a square_1(a x, exists a op*_p1(a, a))
{
  return op*_p1(x,x);
}
```

```
forall a : a fourth_1(a x, exists a op*_p1(a, a))
{
  return square_1<a, op*_p1> (square_1<a, op*_p1> (x));
}

forall : int f_1()
{
  return fourth_1<int, op*_1>(7);
}
```

The closure `<a, op*_p1>` in `fourth_1` is not a constant.

The same phenomenon results when we write recursive polymorphic routines. For example the “Sea” routine:

```
forall a : void print (list(a) l, exists void print(a))
{
  if (l != NULL)
    {
      print (l->data);
      print (l->next);
    }
}
```

Is translated into the “Lake” routine:

```
forall a : void print_1 (list(a) l, exists void print_p1(a))
{
  if (l != NULL)
    {
      print_p1 (l->data);
      print_1<a, print_p1> (l->next);
    }
}
```

`<a, print_p1>` is not constant.

While the `fourth` example could have been rewritten by adding `exists square` to `fourth's` parameter list, there is no such conversion for recursive functions. We could define print as requiring a `print(list)` exist parameter, but the `print(list)` exist parameter would also require a `print(list)` operation parameter etc.

For simple recursive calls, like this one, where the closure required for the recursive call to `print` is the same as the closure of the instance of `print` making the recursive call, we can arrange things so that a function always receives its own closure as one of its parameters and use this to make the recursive call, thereby not needing to construct a variable closure for the recursive call. However, this will not work for all instances. For example, in the following recursive function the closure for the recursive call to `q` is not the same as the closure for the instance of `q` making the recursive call.

```
forall a, b : int q(a x, b y, exists int term(a), exists int term(b))
{
  if (term(a))
    return 0;
  else
    return q(y, x);
}
```

Is translated by the inferencer to :

```
forall a, b : int q_1(a x, b y, exists int term_p1(a), exists int term_p2(b))
{
  if (term_p1(a))
    return 0;
  else
    return q_1<b, a, term_p2, term_p1>(y, x);
}
```

Mutually recursive functions present a similar problem.

In the following section we look at the impact of having to support these variable closures on the range of implementation strategies available for "Sea". We then present an algorithm that transforms a "Lake" program with non-constant closures into one where all closures are constant.

3.1.1 Creating Closures at Runtime

Creating a non-constant closure at runtime is not very difficult, and takes about the same amount of time as it takes to construct a normal parameter list. Often, in order to complete one closure, other closures will need to be created, and a large tree of closures will get created in order to generate one call. As long as all the requested closures are used, despite the fact that they are all created at once, this cost is amortized across many function calls, and the runtime cost as a percentage of execution time remains acceptable. However, if a style of programming is adopted whereby large complex closures are constructed, but never completely used, or where closures are created prior to a majority of calls, closure construction could be a major runtime cost.

3.1.2 Non-Constant Operation Parameters

Non-constant operation parameters (ie. operation parameters from a non-constant closure) are not very difficult to implement, and are not inordinately expensive at runtime. The problem with non-constant operation parameters is that they cannot be inlined. Inlining is to an important optimization if we hope to get "C"-like performance out of polymorphic programs. (see chapter 5)

If we want to add "C++"-style constructors, destructors and overloaded assignments to "Sea" then inlining becomes vital. The inline expansion of a constructor that is bound to the default no-operation constructor is no-code and no-overhead, whereas a constructor that is received as a parameter must always be called. This is a problem because there are so many places in a "Sea" program where constructors must be called.

For example when the following function was processed by the AT&T "C++" compiler (bignum was defined with a constructor, a destructor, and an overloaded assignment operator) 16 calls were generated to one of these functions. Every "Sea" function that has polymorphic arguments would suffer a similar explosion, even if, as would usually be the case, it was called with null constructors and destructors.

```
// ``bignum'' is defined as a class with a constructor, a destructor
//    and an assignment operator.


bignum pythag(bignum x, bignum y)
{
  bignum r;
  r=bignum_sqrt(x*x+y*y);
  return r;
}
```

3.1.3 Non-Constant Type Parameters

"Sea" type parameters can be bound to any type. Different types have different sizes and possibly different alignment requirements. In order to support variable closures we have to generate object code that is parameterized at runtime by the size of its forall types. Supporting variables and parameters whose size is not known at compile time is difficult, not very efficient, and forces us to use a less efficient calling convention for all "Sea" functions (not just polymorphic functions).

When type parameters in a "Lake" program are compile time constants then we can use this to either 1) generate specialized monomorphic versions of polymorphic functions or 2) continue generating one object code function per source function, but generate much more elaborate closures that contain information such as parameter offsets and stack layout for local variables.

One "Sea" variant will only bind type parameters to pointer types. For this variant no type attributes are required at runtime and it is possible to build an efficient compiler even in the presence of runtime closures.

More detail on the advantages and pitfalls of these various implementation techniques can be found in chapter 4.

3.1.4 Other Applications

Transforming a "Lake" program so that all type and operation parameters are constant is also very helpful if we are going to construct the "Sea" variant (described at the end of this document) where type and operation parameters do not have to be specified at the function declaration site. With none of the parameter types specified, and none of the intermediate values constrained, almost every function call made in the body of a polymorphic function must be a separate operation parameter. The style of programming encouraged by this "Sea" variant will involve the construction of large numbers of large non-constant closures, and constructing these large closures at runtime will be prohibitively expensive.

3.1.5 Eliminating non-constant closures

Within polymorphic function `f` that has been specialized by closure `c`, a new closure `z` can be constructed using constant types, constant operations, type parameters from `c`, type constructors applied to type parameters from `c`, and operation parameters from `c`. If `z` is constructed using any parts of `c` then `z` will be a non-constant closure. Notice that any closure created in `f` will consist entirely of fields from `c` and constants, thus all the values needed to construct `z` are also available when `c` is created.

If `z` is a non-constant closure it can be eliminated by converting it into an additional field in `c`, then modifying the places where `c` is created to construct the extra field `z` out of constants and other members of `c`.

So, for example, the `fourth` function defined earlier could be converted to :

```
forall a : a square_1(a x, exists a op*_p1(a, a))
{
  return op*_p1(x,x);
}

forall a : a fourth_1(a x, exists a op*_p1(a, a), exists a square_p1(a))
{
  return square_p1 (square_p1 (x));
}

forall : int t_1()
{
  return fourth_1<int, op*_1, square_1<int, op*_1>>(7);
}
```

The algorithm as described so far has two problems 1) it will not terminate if it encounters a recursive polymorphic function and 2) we haven't defined the order in which expansions will be done when there are more than one non-constant closure in a program. What follows is a more detailed version of this algorithm which addresses these issues.

Convert all the non-constant closures in the "Lake" program into operation parameters. These are normal operation parameters that will be received through the normal closure mechanism, but we introduce a new syntax both to highlight their different role, and to provide information to the caller detailing how this parameter is to be filled in. Before the body of the function, in square brackets, we add a list of expressions of the form “name=closure body”. “closure body” describes how the closure is to be created (in terms of constants, type parameters, operation parameters and type constructors applied to type parameters). “name” is then used in the body of the function to refer to this closure.

`fourth` would be translated to :

```
forall a : a fourth_1(a x, exists a op*_p1(a, a)) [ c1=square_1<a, op*_p1> ]
{
  return c1 (c1 (x));
}
```

Starting with the set of constant closures, expand each closure by adding the new operation parameters required by the function to which the closure refers. Recursively expand any closures that are generated by this process. Each time a new closure is created, before that closure is expanded, check whether an identical closure has already been expanded, and if so use the previous expansion rather than generating a new one.

Checking for duplicate closures not only reduces the number of closures that are created, but also allows most polymorphic recursive functions to be processed. (see later for details)

The `fourth` program would be processed as follows :

The closure template expansion algorithm doesn't need function bodies of types, only the set of constant closures and the set of templates.

```
Constant Closures
fourth_1<int, op*_1>

Closure Templates
fourth_1 : [ c1=square_1<a, op*_p1> ]

Index   Input Closure             Expanded Closure
------------------------------------------------------------
#0 :    {fourth_1, int, op*_1} -> {fourth_1, int, op*_1, #1}
#1 :    {square_1, int, op*_1} -> {square_1, int, op*_1}
```

The following sample program will be used to demonstrate the algorithm. This is a tricky example : `f` and `g` are mutually recursive functions, and the order of the parameters is switched with each iteration.

```
int q(int x) { ... };
int q(double x) { ... };

forall a, b : extern int g(a x, b y, exists int q(a), exists int q(b));

forall a, b : int f(a x, b y, exists int q(a), exists int q(b))
{
  g(y, x);
}


forall a, b : int g(a x, b y, exists int q(a), exists int q(b))
{
  f(x, y);
}

int main()
{
  print(f(5, 5.0));
}
```

Is translated by the inferencer to :

```
forall a, b : int f_1(a x, b y, exists int q_p1(a), exists int q_p2(b))
{
  g_1<b, a, q_p2, q_p1>(y, x);
}


forall a, b : int g_1(a x, b y, exists int q_p1(a), exists int q_p2(b))
{
  f_1<a, b, q_p1, q_p2>(x, y);
}

int main()
{
  print_1 (f_1<int, double, q_1, q_2>(5));
}
```

After non-constant closures have been turned into operation parameters :

```
forall a, b : int f_1(a x, b y, exists int q_p1(a), exists int q_p1(b))
  [ c1=g_1<b, a, q_p2, q_p1> ]
{
  c1(y, x);
}


forall a, b : int g_1(a x, exists int q_p1(a), exists int q_p1(b))
  [ c1=f_1<a, b, q_p1, q_p2> ]
{
  c1(x, y);
}

int main()
{
  print_1 (f_1<int, double, q_1, q_2>(5));
}

Constant Closures :
f_1<int, double, q_1, q_2>

Closure Templates :
f_1 : [ c1=g_1<b, a, q_p2, q_p1> ]
g_1 : [ c1=f_1<a, b, q_p1, q_p2> ]
```

The output of the algorithm is shown in the following table.

```
Index   Input Closure                       Expanded Closure
--------------------------------------------------------------------------
#0 :    {f_1, int, double, q_1, q_2} -> {f_1, int, double, q_1, q_2, #1}
#1 :    {g_1, double, int, q_2, q_1} -> {g_1, double, int, q_2, q_1, #2}
#2 :    {f_1, double, int, q_2, q_1} -> {f_1, double, int, q_2, q_1, #3}
#3 :    {g_1, int, double, q_1, q_2} -> {f_1, int, double, q_1, q_2, #0}
```

If no type generators are used, each template can only rearrange its input type and operation parameters, possibly introducing some constants. There is a finite number of arrangements of the initial closure and the constants that are introduced in subsequent levels. So for recursive polymorphic functions in which type generators are not involved in the recursion, our algorithm will generate all the possible variants, then introduce a loop in the closure graph. We can then compile this class of polymorphic recursive functions.

It is possible for there to be a large number of versions generated before a cycle is generated, but is felt that this will occur very seldom is real programs. The following example function will cause 40320 constant closures to be generated.

```
/* 8*7*6*5*4*3*2*1 = 40320 versions */

struct s1 { int i; };
struct s8 { int i; };

int o(struct s1 x) { ... };
int o(struct s8 x) { ... };

forall t1, t2, t3, t4, t5, t6, t7, t8 :
     int q(t1 x1, t2 x2, t3 x3, t4 x4, t5 x5, t6 x6, t7 x7, t8 x8,
        exists int o(t1), exists int o(t2), exists int o(t3), exists int o(t4),
        exists int o(t5), exists int o(t6), exists int o(t7), exists int o(t8))
{
  q(x1, x2, x3, x4, x5, x6, x7, x8);
  q(x2, x1, x3, x4, x5, x6, x7, x8);
  q(x2, x3, x1, x4, x5, x6, x7, x8);
  q(x2, x3, x4, x1, x5, x6, x7, x8);
  q(x2, x3, x4, x5, x1, x6, x7, x8);
  q(x2, x3, x4, x5, x6, x1, x7, x8);
  q(x2, x3, x4, x5, x6, x7, x1, x8);
  q(x2, x3, x4, x5, x6, x7, x8, x1);

  if (o(x1))
    return 0;
}

main()
{
  struct s1 v1;
  struct s8 v8;

  q(v1, v2, v3, v4, v5, v6, v7, v8);
}
```

For recursive functions where a type constructor is involved in the recursion, there is an infinite number of types and therefore closures, and our algorithm will fail. This is not felt to be a severe restriction. The following is an example of a function we cannot compile.

```
// pow2 : calculates 2^n

int f(char *x)
{
  return 1;
}

forall a : int f(a *xp, exists int f(a))
{
  return f(*xp) + f(*xp);
}

forall a : g(int n, a x, exists int f(a))
{
  if (n>0)
    return g(n-1, &x);
  else
    return f(x);
}

int pow2(int n)
{
  return g(n, "dummy parameter");
}
```

The above program is interesting because it makes use of the limited curried function parameter support—which we introduced to support inferred type and operation parameters—to construct, at runtime, a function that calculates `2^n`.

3.1.6 Separate Compilation Issues

To eliminate constant closures the compiler must have access to the templates for all polymorphic functions in the program, and normal "C" prototypes do not provide this information. All the implementations we propose eliminate constant closures in a global pass just before linking. (see chapter 4)

Chapter 4

Implementation of “Lake”

In the following section we present several ways of implementing “Lake”. Each of the implementations has different advantages, and places different constraints on the design of the language. “Lake” has been designed so as not to preclude any of the major implementation strategies.

4.1 Implementation 1 : Polymorphic object code, runtime variable closures

Each source-level polymorphic function is translated into a single polymorphic object-code function. This object-code function takes, as its first parameter, a closure containing bindings for its type and operation parameters, and uses this closure at runtime, to specialize its own behaviour.

This is the technique that was used for our prototype “Sea” compiler.

4.1.1 Overloading

The “Sea” language allows us to overload a single identifier with multiple function definitions, each with a different type. Whenever this identifier is used, the inferencer selects the function definition with the most appropriate type.

As overloaded functions are compiled we must assign each a unique name: the overload resolver can then translate a reference to an overloaded name into the unique name that refers to the most appropriate function. In "Sea" function definitions can be compiled separately from calls to those functions, thus we have no direct way of communicating the unique name generated when an overloaded function is compiled to the overload resolution algorithm processing a function call made from a separate file.

For example, in file 1 we have definitions for two `print` functions:

```
void print(int x)
{
...
}

void print(double x)
{
...
}
```

And in file 2 we attempt to use these :

```
extern void print(int);
extern void print(double);

main()
{
  print(7);
}
```

In "Sea", every overloaded definition for a name must have a unique type, and this type is available both when the function is compiled and, through function prototypes, when the function is called. By generating a name from the overloaded name as well as an encoding of the type of the function, then this name will uniquely refer to the appropriate function, and will be available everywhere the function name and type are available (ie., at both the definition and the call site). We call these "mangled" names. This technique originated with Bjarn Stroustrups "C++" compiler.

The actual algorithm we use to generate mangled names is derived from the name mangler in the GNU C++ compiler.

Some sample mangled names:

```
int op*(int x, int y);
```

Would be encoded as (note that `op*` has been translated into something more palatable to the assembler and linker) :

```
__multiply__FLiie_iZ

__multiply     Original function name
__             Punctuation to separate name from type encoding
F              Begin function type constructor
  L              Begin argument list
    i              First argument is integer
    i              Second argument is integer
  e              End argument list
  _              Punctuation to separate argument types from return type
  i              Return type is integer
Z              End function type constructor
```

and

```
forall a : a square(a x, exist a op*(a,a));
```

would be encoded as:

```
_square__FLI1aEFLI1aI1ae_I1aZe_I1aZ
```

Refer to the compiler documentation for a complete discussion of the algorithm.

In "Sea", "exist" parameters are part of the type of a function, and therefor must be included in the mangled function name. This can cause very large mangled names to be generated, which means the "Sea" compiler will not be portable to environments where the assembler or linker place do not support long names. All of the machines that we tested allowed names to be at least 512 characters long, which should be enough for most programs.

4.1.2 Closures

For each “Lake” source function, we generate a single object code function. A “Lake” function that has not yet been specialized by having its type and operation parameters bound is represented by a pointer to object code. When a “Lake” function is specialized with type and operation parameters, we represent this by building a closure containing a pointer to the function object code as well as the supplied bindings for the type and operation parameters. When this closure is then applied to list of regular arguments, we call the object code referenced by the closure, and arrange that it will receive a pointer to the closure as its first parameter. The polymorphic object code can then reference the contents of the closure to specialize its own behaviour.

The structure of this closure is as follows :

```
{Pointer to function object code,
 Attributes of first type arg, ..., Attributes of nth type arg,
 First operation parameter, ..., nth operation parameter}
```

For the current implementation the only attribute that is needed to describe each type parameter is the size of the type.

As was discussed in chapter 3, operation parameters are either pointers to another closure, or a pointer to function object code, depending on whether or not the operation parameter has been specialized yet.

When the `square` function :

```
forall a : a square(a x, exist a op*(a,a)) { return x*x; }
```

is applied to an integer argument, it will be specialized into the following closure :

```
; Closure for polymorphic square applied to an integer argument
LC1:
        .long _square__FLI1aEFLI1aI1ae_I1aZe_I1aZ ; pointer to square function
        .long 4                        ; sizeof(int)
        .long LC0                      ; pointer to closure for integer multiply

; Closure for integer multiply (there are no type or operation parameters)
LC0:
        .long _multiply__FLiie_iZ   ; pointer to integer multiply function
```

4.1.3 Operation Parameters

Operation parameters that have not yet been specialized can be specialized by building a closure. Most operation parameters, however, have already been specialized, and are received as pointers to closures.

Calling a closure is not a difficult or expensive operation. Details of the “Sea” calling convention can be found in the next section.

The only problem with receiving operations as parameters is that these operations cannot be inlined. This leads to significantly worse performance for things that are traditionally inlined like the basic integer and floating point operations.

4.1.4 Type Parameters

A “Sea” function can be declared with type parameters. These type parameters can then be used as parameters to type constructors, resulting in a family of variable-sized types and pointers to variable-sized types. These types can then be used to declare parameters and automatic variables.

The operations that the compiler must provide for these variables and parameters are 1) assignment, 2) member access for complex types, and 3) function calls. In this section we look at the compile-time and run-time mechanisms that this requires.

Representation of Type Parameters in the Closure

For the current implementation the only information that is contained in a type parameter is the size of the type it has been bound to.

Often, within our polymorphic routines, we need to calculate the size of a type rounded up to the next alignment boundary. Because our code uses closures much more frequently than it creates them, it would have been better to do this calculation at closure creation time and add the results to the closure as an extra field.

4.1.5 Polymorphic Data Structures

For compatibility with system libraries, monomorphic structs are laid out using the host “C”’s structure layout conventions. On many architectures, it is too complicated to emulate these structure layout rules at run-time for polymorphic types.

So “Sea” structures built using type parameters are not defined as having the same layout as an equivalent monomorphic structure definition; instead every field that could be instantiated with types with varying alignment requirements is given the most conservative alignment. This is not incompatible with the “C” language definition which does not guarantee that structurally identical type declarations have identical layout. As in standard “C”, the total size of a structure is padded so that if an array of structures is started on the most conservative alignment boundary, then every field in every struct in the array will have correct alignment.

If an application requires that both a polymorphic and a monomorphic routine can work on the same “struct”, then parameterized types—which were explicitly designed for this purpose—must be used.

Parameterized Types

Type parameters are defined such that the most conservative alignment requirements are applied to every field whose alignment requirements may vary with different instantiations of the type generator. This makes it possible for monomorphic and polymorphic functions to access the same data structure without forcing the polymorphic routine to do complicated and time-consuming alignment calculations.

For example :

```
struct {
  char x;
  char y; };
```

Would be laid out as :

```
Byte     : 00 01
Contents :  x  y

struct {
  t x;
  t y; };
```

In a polymorphic routine where “t” was bound to “char” would be laid out as :

```
Byte     : 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
Contents :  x ------Padding-------  y ------Padding-------
```

Whereas the generated type :

```
type pair(type t)
{
  return struct {
           t x;
           t y; };
}
```

Whether instantiated as `pair(char)` or `pair(t)` (where `t` is bound to `char`) would be laid out as :

```
Byte     : 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
Contents :  x ------Padding-------  y ------Padding-------
```

Arrays

Because all "Sea" types are defined such that as long as the first element of the array has correct alignment, all elements of the array have correct alignment, no alignment considerations need to be taken into account to locate array members. Thus polymorphic arrays require little runtime support and are compatible with monomorphic arrays.

Pointers

No additional runtime support is needed to allow pointers to point at polymorphic objects.

4.1.6 Calling Conventions

Because there are few built-in operations for objects with polymorphic type, almost every operation must be done through calls to monomorphic functions. Thus it is important that we have efficient polymorphic calling conventions.

All Functions Must Use The Same Calling Convention

Operation parameters to polymorphic functions can be bound to either monomorphic functions (specialized by an empty closure) or polymorphic functions. If different calling conventions were used for monomorphic and polymorphic functions, every time an operation parameter was called we would have to select one of two different calling conventions, and—as will be discussed below—building a monomorphic argument list from within a polymorphic function can be very costly. Every function in a "Sea" program could potentially be selected as an operation parameter, so every function must use the same calling convention. This makes it even more crucial that our polymorphic calling conventions be efficient.

For example :

```
// put integer
void put(FILE *f, int x)
{
  fwrite(&x, sizeof(int), 1, f);
}

int f(int x)
{
  put(stdout, 4);
}
```

However, lurking somewhere else in the system we may have :

```
// put list
forall streams,elem :
void put(stream s, list(elem) l, exists void put(stream, elem))
{
  while (1)
    {
      put(s, l->data);
      l = l->next;
    }
}

void g(list(int) l)
```

```
{
  put(stdout, 1);
}
```

`put(integer)` is passed as an operation parameter to `put(list)`, so `put(integer)` must be compiled with standard calling conventions, which means the call to `put(integer)` in `f` has to use standard calling conventions.

Host “C” calling conventions

The “C” standard allows machine designers and compiler writers considerable freedom in the design of their calling conventions, and RISC machines (almost every current processor design other than the i386) have highly tuned register-based calling conventions.

It would be very good, for both performance and portability, if “Sea” could use these calling conventions. The following is a typical RISC calling convention :

```
For Each Arg :
  If Arg is Integer Then
    If more registers are available
      Put in next available Register
    else
      Put on stack (aligned to 32 bits)
    endif
  endif

  If Arg is Double or Float Then
    If more floating point registers are available then
      Put in next available Floating Point Register
    else
      Put on Stack (aligned to 64 bits)
    endif
  endif

  If Arg is struct or union Then
    Put as many words of struct as will fit in rest of register file
    Put balance on stack
  Endif
  etc.

EndFor
```

While it is possible to implement such an algorithm at runtime, the resulting code would require many conditional branches per argument, and would have a difficult time even loading the registers. (There is no way to access the register file via an index on most machines, so the code would have to build an image of the register file in memory, then load the image—thereby defeating the whole point of passing arguments in registers.)

Many compilers/libraries are capable of unpacking their parameter lists at runtime in order to support calling vararg functions like `printf` when no prototype has been supplied. The code to do this is not compact, efficient or pretty.

Return conventions are also complex and parameterized by type, and thus pose similar problems.

Because "Sea" cannot use the native calling conventions on all machines, we do not define the "Sea" language as having compatible calling conventions with the host "C". Instead we have provided the "cdecl" mechanism that allows "Sea" and "C" programs to call each other. This is not much of a loss because "Sea" overloaded function definitions were in a different linker name space (by virtue or their mangled names) than "C" functions.

Stack-Based calling conventions

Using stack-based calling conventions would make "Sea" perform significantly (perhaps a factor of 3) worse than "C" on RISC machines.

Variable-sized argument lists are simple to construct. Each parameter is either pushed or copied onto the stack.

If the processor has strict alignment requirements for any type, we can either 1) add type alignment information to our closures and calculate appropriate alignment for each polymorphic parameter as they are pushed (or retrieved) at the cost of considerable runtime overhead for each parameter, or 2) push all parameters with the strictest alignment (remember we gave up on being compatible with host calling conventions) which reduces overhead when pushing polymorphic parameters, but forces us to push an extra word of padding every time a monomorphic routine wants to pass an "int" or a pointer.

“GCC” has some built in support for stack-based variable-sized arguments, however this code is not operational in the distributed compiler. We fixed this code to the point where we could experiment with it and got the following results :

```
int g(int w, struct { char c[w]; int m; } kk,
             struct { char c[w]; int m; } kkk, int q)
{
printf("w=%X, &w=%X, &kk=%X, &kkk=%X, &q=%X\n", w, &w, &kk, &kkk, &q);

return 0;
}
```

Compiled to about 100 lines of assembly. However, most of this was repeated calculations, so when we re-compiled with optimization we got the following output :

```
LC1:
        .ascii "w=%X, &w=%X, &kk=%X, &kkk=%X, &q=%X\12\0"
        .align 1
.globl _g
_g:
        .word 0x0
        movl 4(ap),r0
        addl3 r0,$3,r1
        movl $2,r3
        movl $30,r4
        extzv r3,r4,r1,r2
        incl r2
        moval 3[r2],r2
        extzv r3,r4,r2,r2
        moval 0[r2],r2
        extzv r3,r4,r1,r1
        incl r1
        moval 3[r1],r1
        extzv r3,r4,r1,r1
        moval 8(r2)[r1],r1
        addl3 ap,r1,-(sp)
        pushab 8(ap)[r2]
        addl3 ap,$8,-(sp)
```

```
addl3 ap,$4,-(sp)
pushl r0
pushab LC1
calls $6,_printf
clrl r0
ret
```

Most of this code is needed to calculate the offsets of the input parameters in the stack frame, similar calculations are required when a polymorphic parameter list is being passed. These calculations could be simplified considerably if we provided, in the closure, the size for each type rounded up to the next word boundary, rather than having the generated code do this calculation at runtime.

On machines with strict alignment requirements, the calculations needed to build and disassemble a variable-sized parameter list get even more complicated. On such machines “GCC” currently gives up and calls `abort()`.

We were going to use this calling convention for the first version of “Sea” so as to get a working compiler more rapidly. However as work progressed it became apparent that “GCC”’s support for variable sized arguments had many subtle problems, and it was felt that it would be quicker to implement the more efficient calling convention we wanted for the final version of our compiler than to track down and repair all the problems with the existing system.

The problems with this calling convention are :

1. Stack-based calling conventions are a big loss for RISC machines.

2. When passing polymorphic arguments, the caller does complex calculations in order to place all the parameters in the parameter list, and then the called function repeats the same calculations in order to access the arguments.

3. There are few built-in operations for polymorphic types. Polymorphic functions do most of their actual work by calling monomorphic functions. Most of the time and code in polymorphic functions is spent doing variable-sized copies of input parameters to subfunction parameters. It would be useful if there were some way to reduce the number of such copies done.

4. On machines with strict alignment requirements there is even more runtime overhead.

"Sea" Calling Conventions

With the following goals in mind we set out to design a new calling convention for the "Sea" language. This should:

1. Use register-based calling conventions on RISC machines.
2. Not be much slower than the native calling conventions for monomorphic functions.
3. Be as fast as possible for polymorphic functions. This is done primarily by avoiding conditions that must checked at runtime and by reducing the number of variable-sized copies needed.

For the following discussion we assume that "int"s and pointers are 32 bits and that "longs" and "doubles" are 64 bits. The same discussion applies to other machines, however some of the particulars will be slightly different.

- The first parameter to a function is always a pointer to the closure that contains bindings for any type or operation parameters. If there are no fields in the closure (apart from the pointer to the functions object code) then this parameter does not have to be valid.
- The second parameter is a pointer to the stash in which to write the return value. If no such stash is required then this parameter can be indeterminate. (More details on the use of this "return value pointer" will be given later).

  Usually, the above two parameters will be passed in registers, thus if valid values are not required there will be no cost apart from the loss of two register parameters.
- For each argument we do the following:
  - If an argument is the size of a word then we pass it directly.
  - If an argument is not the size of a word then we store the argument in memory (If it is not already stored there) and pass a pointer to the argument. We call the place where the argument resides in memory the "stash".

    So every parameter is a word, containing either the actual value or a pointer to the actual value. We call this type of parameter a MA. (Multiplexed Argument)

  Having every parameter in the same format—regardless of type—allows us to use RISC register-based calling conventions to pass these parameters.

  This calling convention is as efficient as the host calling convention for pointers and integers, the two most common types in "Sea" programs. Unfortunately performance for "chars", "shorts", "longs", and "doubles" is significantly degraded.

  - The caller must guarantee that the stash areas referenced by the parameters it is passing be constant until either 1) the function returns or 2) all or part of the stash area is overwritten because it is the return area for the same call.
  - The callee must not modify any stash areas referenced by its input parameters. If an input parameter is in danger of being modified, the compiler must make a copy of the input parameter, and use this copy in place of the original.

    These two rules allow many parameters to be passed without having to copy their data to a stash. In particular they allow parameters to be used as parameters to subsequent function calls with very little overhead. This is very important because almost every operation a polymorphic function wants to perform on its arguments must be done through a function call.

    With this scheme when copies are done they tend to be done either in the monomorphic function that made the initial polymorphic call, or in the monomorphic functions that do the actual operations. It is much cheaper to copy an object in a monomorphic function—when its size and layout are known—than in a polymorphic function.

- To return a value from a function:
  - The callee must assume that the area referred to by the return area pointer may be an alias for any memory in the system, including the stashes of its input parameters.

    This rule allows expressions of the form "`var=func(...)`" to be compiled so that the return value of "`func`" can be written directly into "`var`". This is a useful optimization in polymorphic functions, because allocating and copying variable-sized temporaries is so expensive.
  - Return values that are the size of a word are returned in the return value register.

  - Return values that are not the size of a word are written to the stash referenced by the "return value stash pointer" input parameter. The "return value stash pointer" is then copied to the "return value register".

    Having values returned in the same format that is required for subsequent function calls (or storage in a local variable), allows the return value of one function call to be used as a parameter to a subsequent function call with a minimum of overhead.

    Also, returning small values directly in registers is very important if we want to have "C"-like performance for monomorphic pointer and integer functions.

- Polymorphic local variables (and temporaries) are represented by a word that contains either 1) For types that are the size of a word: the variable's value or 2) For other types: a pointer to a separately allocated stash that contains the variable's value.

  This is the same format that is required for parameters (and return values) by our new calling conventions. Keeping variables in this format saves us from having to do an expensive run-time conversion prior to every call (or every use as a return area).

The following examples are written in the assembly language of a fictitious processor that combines the instruction set of the VAX with the register windows of the SPARC. (This was done to demonstrate the algorithm's proficiency with register-based calling conventions, while retaining a human-readable assembly language.)

Performance for monomorphic functions

For parameters that are a word or smaller in size, apart from the loss of two registers, our new calling convention performs identically to the host calling convention on most machines.

```
int add3(int x, int y, int z)
{
  return x+y+z;
}

int g()
{
  int q;
  q = add3(1,2,3);
  return q;
}

_add3__FLiiie_iZ:
        addl i2,i3              ; add up parameters received in regs and
        addl i3,i4,i1           ;  leave result in return reg (i1)
        ret

_g__Fle_iZ:
        movl 1,o2               ; pass arguments in output registers
        movl 2,o3
        movl 3,o4
        call _add3__FLiiie_iZ
        movl o1,l1              ; copy return from ``add3'' to ``q''
        movl l1,i1              ; return ``q'' in return reg
        ret
```

For `double` and `long` parameters (which are typically larger than a word), we have to pass and return values via pointers, resulting in significant performance degradation.

```
double add3(double x, double y, double z)
{
  return x+y+z;
}

double g()
{
  double q;
  q = add3(1.0, 2.0, 3.0);
  return q;
}

_add3__FLddde_dZ:
        addf (i2),(i3),f0       ; access parameter values indirectly
        addf (i4),f0,(i1)       ; write return value into area pointed by i1
        ret
```

```
_g__Fle_dZ:
        subl 8,sp               ; allocate space for ``q'' (can't use regs)
        movl sp,l1              ; store pointer to ``q'' in reg

        subl sp,24              ; allocate space for stash
        movl sp,l2              ; we will use ``l2'' to fill stash

        movd 1.0,(l2)           ; put float value 1.0 in stash
        movl l2,o2              ; pass pointer to stashed ``1.0'' as first arg
        addl 8,l2               ; advance stash pointer

        movd 2.0,(l2)           ; stash and pass 2.0 as second arg
        movl l2,o3
        addl 8,l2

        movd 3.0,(l2)           ; stash and pass 3.0 as third arg
        movl l2,o4
        addl 8,l2

        movl o1,l1              ; pass pointer to place to store return value
        call _add3__FLddde_dZ   ; call function

        movd (l1),(i1)          ; copy ``q'' to return area pointer by i1
        ret
```

The same function compiled using normal “C” calling conventions might read as follows: (SPARC’s use very inefficient floating point calling conventions that would probably perform even worse than our new calling conventions. This example is compiled using well tuned floating point calling conventions.)

```
_add3__FLddde_dZ:

        addf f1,f2,f5           ; add up parameters received in regs and
        addf f5,f3,f0           ;   leave result in f0
        ret

_g__Fle_dZ:
        movd 1.0,f1             ; pass arguments in output registers
        movd 2.0,f2
        movd 3.0,f3
        call _add3__FLddde_dZ   ; call function
        movd f0,f16             ; copy return from ``add3'' to ``q''
        movd f16,f0             ; return ``q'' in floating return reg
        ret
```

The `add3` program is 20 instructions long when compiled using our new calling conventions and 10 instructions long when compiled using more conventional calling conventions.

The above example represents the worst case example for monomorphic floating-point code. There are a number of simple optimizations we can apply to common cases: 1) Constant parameters (like the above) can be stashed at compile time as constant data 2) we can often pass the address of floating point parameters and variables without having to re-stash them and 3) if we set up a number of stash locations we can often re-use them for several calls.

One solution to this problem of expensive floating point parameters would be to alter our calling convention to pass all values less than the size of a `double` directly. We have chosen not to do this because it degrades performance substantially for integer and pointer code (by either wasting half the registers for a register-based calling convention, or by forcing us to push garbage words for a stack-based calling convention as well as numerous other complications—see later for details).

For `struct` parameters, which are seldom used in normal "C" programs, our new calling conventions are at least as efficient as the usual host calling conventions.

So for monomorphic functions our calling convention is as efficient as the host "C" calling convention for all types except doubles and longs. Programs that make extensive use of `double` or `long` parameters may experience significant performance degradation.

Examples of Polymorphic Functions

First we show the assembly code for the simple `square` function we have been using as an example throughout this document.

```
forall a : a square(a x, exists a op*(a,a))
{
  return x*x;
}

void f()
{
  print(square(5));
  print(square(5.0));
}
```

Is compiled to:

```
_square__FLI1aEFLI1aI1ae_I1aZe_I1aZ:
        movl i2,o2              ; pass MA for ``x'' as first arg to op*
        movl i2,o3              ; pass MA for ``x'' as second arg to op*
        movl i1,o1              ; pass our return area pointer to ``op*''
        call *12(i0)            ; call ``op*'' operation parameter
        movl o1,i1              ; return MA that was returned by ``op*''
        ret

_closure_0:
        .long _multiply__FLiie_iZ

_closure_1:
        .long _square__FLI1aEFLI1aI1ae_I1aZe_I1aZ
        .long 4
        .long _closure_0

_closure_2:
        .long _multiply__FLdde_dZ

_closure_3:
        .long _square__FLI1aEFLI1aI1ae_I1aZe_I1aZ
        .long 8
        .long _closure_1

LC0:
        .double 5.0
```

```
_f__FLe_Z:
        movl 5,o2              ; pass integer 5 in MA
        movab _closure_1,o0    ; pass pointer to closure for square(int)
        call *(o0)             ; call ``square'' via pointer in closure

        movl o1,o2             ; pass MA returned from square(int)
        call _print__FLie_iZ   ; call print(int)

        movab LC0,o2           ; pass pointer to stashed ``5.0'' as first arg

        subl sp,8              ; allocate space to stash ``square'' return
        movl sp,l2
        movl l2,o1             ; and pass as return area
        movab _closure_1,o0    ; pass pointer to closure for square(double)
        call *(o0)             ; call ``square'' via pointer in closure

        movl l2,o2             ; pass stash containing ``square'' return
        call _print__FLde_iZ   ; call print(double)

        ret
```

While these examples demonstrate good performance for small-sample polymorphic routines, they get this performance through an array of optimizations that cannot be applied in all cases. One way to compare calling conventions would be to compile, then benchmark a large program using each of the proposed calling conventions. This technique, however, does not provide much insight into how to design a good calling convention.

Fortunately, it is possible to enumerate all the sources of polymorphic values, and all the operations that can be performed on these values. This in a useful tool when designing and evaluating calling conventions.

The following are all the sources of variable-sized values :

1. Input parameter
2. Automatic variable
3. Function return value

4. Dereferenced value (Includes pointer, array and field access)

These are all the operations that can be applied to a variable sized value :

1. Pass as Parameter
2. Return from function
3. Assign to variable
4. Assign to parameter
5. Assign to Dereferenced lvalue (Includes pointer, array and field access)
6. Take address of

In the following table we explore all the transactions that a "Sea" compiler is required to support, and how a compiler using our new calling conventions would implement them:

1. Create polymorphic local variable (or temporary polymorphic variable)

   We create polymorphic local variables in the same format (using a stash for objects that are not a word in size) that is used for parameters. This allows polymorphic local variables to be efficiently 1) passed as parameters and 2) used to store the results of function calls.

   The following code sample creates a local variable with stash size specified by a field in the closure. The MA for the local variable is stored in register "`l0`", and the stash is allocated from the stack.

   ```
           addl 8(i0),sp
           movl sp,l0
   ```

   We execute the same code for word-sized variables, but in this case the stash area will never be used. We do this because it is considerably cheaper to allocate a bogus stash than it is to check(at run-time) whether a stash is really needed.

2. Input parameter or automatic variable passed as parameter

   If the stash for a local variable or parameter cannot be guaranteed constant for the duration of the call (usually because a pointer to the variable has been passed outside the function), then the compiler must make a safe copy of the variable before passing it.

   Once we have a safe variable in MA form, our calling convention allows us to pass just the MA, without making a new copy of the stash (if there is one). The code that is required to pass the MA is the same as would be required to pass a word-sized argument using the host "C"'s calling convention. For our assembly language this is:

   ```
           movl i2,o2
   ```

3. Input parameter or Automatic variable returned from function

   ```
   if sizeof(t) == sizeof(int)
     return variable's MA in return reg
   else
     copy variable's stash to area pointed by "return area pointer"
   ```

4. Input parameter or automatic variable assigned to variable or parameter

   ```
   if sizeof(t) == sizeof(int)
     copy MA
   else
     copy stash
   ```

5. Input parameter or Automatic variable assigned to dereferenced lvalue

   ```
   if sizeof(t) == sizeof(int)
     copy MA to target
   else
     copy area referenced by MA to target
   ```

6. Function return value passed as Parameter

```
Allocate stash for return value (re-use same stash for multiple calls)
Pass pointer to stash as return value area
Call first function
Pass return register as parameter to second function (It will either
  contain a value if the sizeof(t) == sizeof(int) or it will contain a
  pointer to the already allocated stash, and the value will have been
  written in this stash.)

        addl 8(10),sp
        movl sp,i2
        call _func
        movl i2,i3
        call _func2
```

7. Function return value returned from function

   By passing the “return area pointer” on to a subfunction, and then copying the subfunction’s “return register” to the current function’s “return register”, it can be arranged that the current function does not have to recopy and stash-based portion of the return value.

```
        movl i1,o1
        call f
        movl o1,i1
        ret
```

8. Function return value assigned to variable or parameter

```
Pass the MA for the variable or pointer as the ‘‘return area pointer’’
Call the function
Store the return register from the function in the MA for the variable
  (If sizeof(t) == sizeof(int) the return register will contain a
  value, otherwise it will contain the original stash pointer)

        movl l1,o1              ; l1 is the MA for the variable
        call f
        movl o1,l1              ; o1 will either contain a value, or
                                ;  will still contain l1
```

9. Function return value assigned to dereferenced lvalue

```
call function with pointer to lvalue as return area pointer
if (sizeof(t) == sizeof(int))
  copy return value reg to lvalue
```

10. Dereferenced value passed as parameter

    If we can determine that there are no aliases to the object then we can avoid copying the object's data:

    ```
    if (sizeof(t) == sizeof(int))
      load pointed at object into MA
    else
      load address of object into MA
    ```

    If there may be aliases to the object then we must copy the object's data:

    ```
    if (sizeof(t) == sizeof(int))
      load pointed at object into MA
    else
      allocate stash for pointed to object
      copy object into stash
      load address of stash into MA
    ```

11. Dereferenced value returned from function

    ```
    if (sizeof(t) == sizeof(int))
      return value in return register
    else
      copy value to area pointer by "return value pointer"
    ```

12. Dereferenced value assigned to variable or parameter

    ```
    if (sizeof(t) == sizeof(int))
      copy value to MA
    else
      copy value to stash
    ```

13. Dereferenced value assigned to Dereferenced lvalue

```
copy value
```

14. Take address of parameter or variable

```
if (sizeof(t) == sizeof(int))
  return address of MA (we may have to move MA to memory first)
else
  return contents of MA
```

Evaluation of our Calling Convention

Putting word sized values inside the MA has little effect on the performance of polymorphic functions: it forces us to add many conditionals, but these probably improve performance by allowing a very common case (word size arguments) to execute without having to do do a call to "bcopy" (the variable-sized copy function. For monomorphic functions, however, allowing word sized arguments to be passed and returned directly in registers can improve performance significantly.

Our elaborate rules to reduce the need for copying arguments allow the following cases to be implemented very cheaply (No conditionals and no variable-sized copies):

2. Input parameter or automatic variable passed as parameter (no alias case)

6. Function return value passed as parameter

7. Function return value returned from function

8. Function return value assigned to variable or parameter

These cases can be implemented with one conditional:

10. Dereferenced value passed as parameter (no alias case)

9. Function return value assigned to dereferenced lvalue

The following cases still need to do a variable sized copy:

3. Input parameter or automatic variable returned from function

4. Input parameter or automatic variable assigned to variable or parameter

5. Input parameter or automatic variable assigned to dereferenced lvalue
11. Dereferenced value returned from function
12. Dereferenced value assigned to variable or parameter
13. Dereferenced value assigned to Dereferenced lvalue

No copies are required under either scheme for the following transactions:

1. Create polymorphic local variable (or temporary polymorphic variable)
14. Take address of parameter or variable

Note that all the cases that still need to do a variable-sized copy are either assignment statements or return statements. Unless we depart from "C"'s model of "a variable is a name for a region of storage" it is impossible to eliminate copies for assignment or return statements (Although there are some tricks we can play with return statements: for example see "returning a return value".)

So we have efficient calling conventions for monomorphic code, we have reduced the number of variable sized copies to an absolute minimum, and we have very low overhead for the various glue operations like locating input parameters.

Comparison Of Calling Conventions

Calling conventions can be compared by comparing the run-time algorithms they require to implement each of the above 14 points.

In the following list we compare our new calling convention with the simpler "always copy, no MA" calling convention, which is the same as our calling convention except that arguments are always copied prior to a call and values are never stored in the MA:

Much Better (No conditionals and no variable-sized copies regardless of type)

2. Input parameter or automatic variable passed as parameter
6. Function return value passed as parameter
7. Function return value returned from function
8. Function return value assigned to variable or parameter

Similar (Check for MA-sized argument allows this common case to execute faster)

9. Function return value assigned to dereferenced lvalue

3. Input parameter or automatic variable returned from function

4. Input parameter or automatic variable assigned to variable or parameter

5. Input parameter or automatic variable assigned to dereferenced lvalue

10. Dereferenced value passed as parameter

11. Dereferenced value returned from function

12. Dereferenced value assigned to variable or parameter

Identical

1. Create polymorphic local variable (or temporary polymorphic variable)

13. Dereferenced value assigned to Dereferenced lvalue

Slightly Worse

14. Take address of parameter or variable

Monomorphic code uses the same calling conventions as polymorphic code, but can have quite different characteristics because all conditionals are evaluated at compile time. For monomorphic code our new calling conventions provide the same performance as the host "C" calling conventions for integer and pointer types, and perform the same as the "always copy, no MA" calling conventions.

In summary our new calling convention never performs much worse than the "always copy, no MA" calling convention, and in many important cases it gives much better performance.

Why we use the stash for less-then-word-sized arguments

It is possible to put less-than-word-sized arguments inside a word and pass them inside the MA rather than using the stash.

If we only allow word-sized arguments to be stored in the MA, then whenever we assign a polymorphic variable to a non stash-format object we have to, at run-time, execute the following conditional code.

```
if (sizeof(t) == sizeof(int))
  copy MA word to target
else
  copy stash to target
```

If we allow less-than-word-sized arguments to be stored in the MA, then this conditional becomes much more complex, significantly degrading both performance and code size.

```
if (sizeof(t) == sizeof(int))
  copy MA word to target
else if (sizeof(t) == 1)
  copy low order byte of MA to target
else if (sizeof(t) == 2)
  copy low order 2 bytes of MA to target
else if (sizeof(t) == 3)
  copy low order 3 bytes of MA to target
else
  copy stash to target
```

When we pass less-than-word-sized integers in registers it is desirable to pass them in the a format that the processors integer instructions can operate on. Usually this means we want to pass small integers in the integer registers, as if they were full width integers.

With processors that store the most significant byte of a word in the lowest numbered address (big-endian machines) the layout in storage of a small integer stored in a full width integer and a native small integer are different :

The integer “42” stored at memory location 10

```
Address : 10 11 12 13
Value   : 00 00 00 42
```

The "short integer" 42 stored at memory location 10

```
Address : 10 11 12 13
Value   : 00 42 XX XX    (XX = don't care)
```

The different storage formats become a problem when the address of a polymorphic variable is taken. If we only allow word-sized arguments to be stored in the MA, then the address of a polymorphic variable is either the address of the MA or the contents of the MA:

```
if (sizeof(t) == sizeof(int))
  return address of MA (we may have to move MA to memory first)
else
  return contents of MA
```

If we allow less-than-word-sized arguments to be stored in the MA, then we must add code to calculate the offset of the argument within the MA:

```
if (sizeof(t) == sizeof(int))
  return address of MA + (4 - sizeof(t)) (may have to move MA first)
else
  return contents of MA
```

Whenever we want to use the result of a pointer dereference (pointers, arrays, or field accesses) another set of complications arises. The current version of the compiler has to do the following:

```
if (sizeof(t) == sizeof(int))
  load pointed at object into MA
else
  allocate stash for pointed to object
  copy object into stash
  load address of stash into MA
```

While a version that allows less-than-word-sized objects to be stored in the MA would have to do:

```
if (sizeof(t) == sizeof(int))
  load pointed at object into MA
else if (sizeof(t) == 1)
  load one byte into low order bytes of MA
else if (sizeof(t) == 2)
  load two bytes into low order bytes of MA
else if (sizeof(t) == 3)
  load three bytes into low order bytes of MA
else
  allocate stash for pointed to object
  copy object into stash
  load address of stash into MA
```

Keep in mind that these algorithms must be evaluated—at run-time—whenever a polymorphic memory object is accessed.

If we allow small objects to be loaded in the MA we must load small structures into the MA in the same manner that we would load small integers. Otherwise we would have to add extra conditions to the unpacking and address calculation routines.

Less-than-word-sized arguments are rarely used in “C” programs; instead they are converted to integers before being passed. So having a less efficient calling convention for smaller arguments is not expected to have much impact on the performance “Sea” programs.

One way to get better performance for double parameters would be to expand the MA to the size of a double. However if this were done, the above complications would force us to pass int’s and pointers using the stash.

4.1.7 Actual Code Samples

When we were implementing the calling convention for our “Sea” compiler the priority was on getting a working compiler, and the efficiency of the first version was a lesser concern. Thus the current implementation of the polymorphic calling convention generates fairly inefficient code. The following is the VAX code emitted by our compiler for a polymorphic square program.

```
/* square.d */
```

```
forall a : a square(a x, exist a op*(a,a))
{
  return x*x;
}

cdecl int main()
{
  print (square (5));
  print (square (5.0));
  return 1;
}

#NO_APP
gcc_compiled.:
.text
        .align 1
.globl _square__FLI1aEFLI1aI1ae_I1aZe_I1aZ
_square__FLI1aEFLI1aI1ae_I1aZe_I1aZ:
        .word 0x0
        movl *4(ap),r0          ; load the pointer to our return area into r0
        movl *8(ap),r1          ; load the pointer to our closure into r1
        subl2 $24,sp            ; allocate space for stashing arguments
        movl 8(r1),r1           ; get pointer to ``op*'' closure through r1
        movl 12(ap),12(sp)      ; pass MA for ``x'' as first arg to ``op*''
        movl 12(ap),8(sp)       ; pass MA for ``x'' as second arg to ``op*''
        movl r1,20(sp)          ; pass pointer to ``op*'' closure to ``op*''
        movl r0,16(sp)          ; pass our return area pointer to ``op*''
        calls $10,*(r1)         ; call ``op*'' using ``op*'' closure
        ret                     ; return value that was returned by ``op*''
        .align 2

LC0:    ; closure for int op*(int,int)
        .long _multiply__FLiie_iZ
        .align 2

LC1:    ; closure for square specialized to int square(int)
        .long _square__FLI1aEFLI1aI1ae_I1aZe_I1aZ
        .long 4
        .long LC0
        .align 2

LC2:
        .long _print__FLie_iZ
        .align 2

LC3:
        .long _multiply__FLdde_dZ
        .align 2

LC4:
        .long _square__FLI1aEFLI1aI1ae_I1aZe_I1aZ
        .long 8
        .long LC3
        .align 2

LC5:
        .long _print__FLde_iZ
        .align 1

.globl _main
_main:
        .word 0x3c0
        subl2 $20,sp
        subl3 $4,fp,r8
        movl sp,r7
        subl2 $24,sp
        movl sp,r9
        subl2 $24,sp
        movl $5,20(sp)
        addl3 sp,$20,8(sp)
        movab LC1,16(sp)
        addl3 sp,$16,4(sp)
        subl3 $8,fp,12(sp)
        addl3 sp,$12,(sp)
        movab _square__FLI1aEFLI1aI1ae_I1aZe_I1aZ,r6
        calls $9,(r6)
        movl r9,sp
        movl -8(fp),20(sp)
        addl3 sp,$20,8(sp)
        movab LC2,16(sp)
        addl3 sp,$16,4(sp)
        movl r8,12(sp)
        addl3 sp,$12,(sp)
        calls $9,_print__FLie_iZ
        movl r7,sp
        subl3 $12,fp,r7
        subl2 $28,sp
        movl sp,r8
        subl2 $28,sp
        movd $0d5.0000000000000000000e+00,20(sp)
        addl3 sp,$20,8(sp)
        movab LC4,16(sp)
        addl3 sp,$16,4(sp)
        subl3 $20,fp,12(sp)
        addl3 sp,$12,(sp)
        calls $10,(r6)
        movl r8,sp
        movd -20(fp),20(sp)
        addl3 sp,$20,8(sp)
        movab LC5,16(sp)
        addl3 sp,$16,4(sp)
        movl r7,12(sp)
        addl3 sp,$12,(sp)
        calls $10,_print__FLde_iZ
        movl $1,r0
        ret
```

Longer examples are given in the appendices.

For comparison purposes, the following is a monomorphic version of the same functions. If we had written `square_int` and `square_double` using the inline `*` operator the resulting code would have been even better.

```
/* square.c */

int square_int(int x)
{
  return multiply_int(x, x);
}

double square_double(double x)
{
  return multiply_double(x, x);
}

int main()
{
  print_int (square_int (5));
  print_double (square_double (5.0));
  return 1;
}
```

```
#NO_APP
gcc_compiled.:
.text
.globl _square_int
_square_int:
        .word 0x0
        movl 4(ap),r0
        pushl r0
        pushl r0
        calls $2,_multiply_int
        ret
        .align 1
.globl _square_double
_square_double:
        .word 0x0
        movd 4(ap),r0
        movd r0,-(sp)
        movd r0,-(sp)
        calls $4,_multiply_double
        ret
        .align 1
.globl _main
_main:
```

```
.word 0x0
pushl $5
calls $1,_square_int
pushl r0
calls $1,_print_int
movd $0d5.00000000000000000000e+00,-(sp)
calls $2,_square_double
movd r0,-(sp)
calls $2,_print_double
movl $1,r0
ret
```

A compiler that utilizes the above implementation strategy can use normal “C” style separate compilation :

```
    .c+.h              .c+.h
      |                  |
      |                  |
      V                  V
--------------     --------------
|            |     |            |
|  Compiler  |     |  Compiler  |
|            |     |            |
--------------     --------------
      |                  |
      |                  |
      | .o               | .o
      |                  |
      |                  |
      V                  V
    ------------------------
    |                      |
    |   Standard Linker    |
    |                      |
    ------------------------
               |
               |
               |
               |
          V
        a.out
```

4.2 Implementation 2 : Polymorphic object code, all closures constant

For this implementation we use the algorithm proposed in chapter 3 to transform the "Lake" program so that all closures are constant.

A closure contains a pointer to a function as well as a set of type and operation parameters bindings for a specific invocation of that function. If closures are known at compile time then we can apply, at compile time, the closure to the polymorphic function and thereby generate a specialized version of the polymorphic function. While it is possible to completely specialize the polymorphic function to be a monomorphic function (and this is explored in the next section), this is not the only form of specialization that is possible. We propose retaining one object-code function that is referenced by all closures, but specializing bits of this function by moving constant calculations and bits of code to the closure.

When reading code generated by our "Sea" compiler, it was observed that many complex expressions, consisting entirely of constants and closure fields, were being evaluated at runtime. These expressions are used to access parameters, lay out local variables, locate fields in parameterized types, and during the construction of parameter lists.

For example : (`a` and `b` are type parameters)

```
(((sizeof(a)+3)/4)*4)*2 + ((sizeof(b)+3)/4)*4 + 8
```

Instead of generating runtime code for such expressions, we propose replacing these expressing with references to new closure members. We then add notes to the function definition detailing the (compile-time) calculations we want done. As constant closures are generated we can look up these notes, do the requested calculations, and store the results as new closure fields.

Pointers to chunks of code, like copy operations, can be similarly moved in the closure.

An additional benefit of having all closures constant is that we don't have to create closures at runtime. This can result in significant runtime savings for some styles of programming that involve the construction of many non-constant closures.

A compiler that utilizes this technique needs a global “Closure Processing” pass before linking. This pass uses a simple, fast algorithm, and does not have to process much data, so it is not expected to add significantly to compile time.

As each “`.c`” file is compiled to a “`.o`” file the compiler also outputs a “`.clo`” file that contains : 1) constant closures that have been created during the compilation of that file 2) “closure expansion templates” for all polymorphic functions declared in that file and 3) for each polymorphic function, a list of expressions that can be expanded at compile time.

```
    .c+.h              .c+.h
      |                  |
      |                  |
      V                  V
--------------     --------------
|            |     |            |
|  Compiler  |     |  Compiler  |
|            |     |            |
--------------     --------------
   |      |           |      |
   |.o    |.clo   .clo|    .o|
   |      |           |      |
   |      V           V      |
   |     ---------------     |
   |     |   Closure   |     |
   |     |  Processor  |     |
   |     |             |     |
   |     ---------------     |
   |            |            |
   |            |.o          |
   |            |            |
   V            V            V
----------------------------------
|                                |
|       Standard Linker          |
|                                |
----------------------------------
               |
               |
               |
                  |
                  V
                a.out
```

"`.clo`" files contain constant closures, closure expansion templates, and whatever information is needed about polymorphic functions to allow us to calculate stack offsets for polymorphic automatic variables, etc.

4.3 Implementation 3 : Translation to monomorphic object code

Once we have transformed a "Lake" program so that all closures are constant, we apply these constant closures to the original polymorphic functions and generate specialized monomorphic versions.

A compiler using this technique would read each "Sea" program file, do all type-checking and type inference, output a file containing closure information, and output a file containing "Lake" parse trees for every function. The "closure processor" global pass would then calculate the constant closures for the program. These constant closures can then be combined with the polymorphic function represented as a "Lake" parse tree to generate a specialized "Lake" function. This function can then be compiled to object code.

Specialized "Lake" functions would be standard "C" functions, and could be compiled using standard "C" compilation techniques, or, in fact, we could output these "Lake" functions as "C" code and feed this through the stock "C" compiler. This would result in a portable, high performance (in execution time) compiler.

Because a large body of "Sea" code has not been written, it is unknown how many specialized versions of polymorphic functions would be generated by typical programs. Because we replicate all the code for each version this algorithm could potentially generate very large executable files.

If we generated object code for specialized versions of polymorphic functions anew with every compile (including for libraries) we could not use this compiler on large systems. The solution is to cache object code for already expanded versions of functions. Every expansion is added to this cache after it is generated, and all the expansions generated from a specific file are flushed when the parse-tree generated from that file changes.

A compiler with parse trees for all the functions in a system would also be able to inline any function.

As was discussed earlier, this is the only compilation scheme that can efficiently support “C++”-style constructors and destructors.

Another advantage of this compilation scheme is that it would force library distributors to distribute their libraries in something close to source form.

The structure of a compiler employing this technique would be as follows :

```
    .c+.h                  .c+.h
      |                      |
      V                      V
--------------     --------------
|  Compiler  |     |  Compiler  |
--------------     --------------
   |.t    |.clo   .clo|    .t|
   |      |           |      |
   |      V           V      |
   |    -----------------    |
   |    |    Closure    |    |
   |    |   Processor   |    |
   |    -----------------    |
   |            |            |
   V            V            V
---------------------------------
|       Function Expander       |
---------------------------------
      ^                |.c
      |expansion       |
      |queries         V
      |        ------------------
      |        |  `C' compiler  |
      |        ------------------
      |                |.o
      V                V
---------------------------------
|       Expansion Cache         |
---------------------------------
               |.o+.a
               V
---------------------------------
|       Standard Linker         |
---------------------------------
               |
               V
             a.out
```

“`.t`” files contain “Lake” parse trees. “`.clo`” files contain closure information.

4.4 Implementation 4 : Only pointer types can be bound to type parameters

Almost all the complexity in our polymorphic object code compiler is there in order to support parameters, return values, and aggregate type members whose size is not a compile-time constant. For a variant of "Sea" where only (data) pointer types can be bound to type variables (and typegen parameters), all polymorphic objects have the same representation, and the resulting language system is much simpler, more reliable, more portable, and more efficient.

In the next chapter we will demonstrate that, due to the way "C" variables are defined, we derive surprisingly little advantage out of allowing types other than pointer to be bound to type parameters. We will then argue that the advantages of a "pointers only" definition outweigh the benefits of the more complete system.

What follows is a discussion of how a "pointers only" "Sea" compiler could be implemented :

The first step in compiling "pointers only" "Sea" is to do type inference and overload resolution, translating the "Sea" program into a "Lake" program. For example, the following "Sea" program fragment:

```
// Type generator for ``list''
type list(type elem) {
  return struct {
    list(elem) *next;
    elem data; } *;
}

// put string
void put(FILE *f, char *x)
{
  fputs(x, f);
}

// put list
forall streams,elem :
void put(stream s, list(elem) l, exists void put(stream, elem))
{
  while (l)
```

```
    {
      put(s, l->data);
      l = l->next;
    }
}

// test function
void g(list(char *) l)
{
  put(stdout, l);
}
```

Would be translated by the inferencer to the following “Lake” program: (a real translation would use the mangled names described for implementation 1)

```
// Type generator for ‘‘list’’
type list(type elem) {
  return struct {
    list(elem) *next;
    elem data; } *;
}

// put integer
void put_1(FILE *f, char *x)
{
  fputs(x, f);
}

// put list
forall streams,elem :
void put_2(stream s, list(elem) l, exists void put_p1(stream, elem))
{
  while (l)
    {
      put_p1(s, l->data);
      l = l->next;
    }
}
```

```
// test function
void g_1(list(char *) l)
{
  put_2<FILE *, char *, put_1<>>(stdout, l);
}
```

"Lake" has two features that are missing from "C":

1. Polymorphic "Lake" functions are specialized to be monomorphic "Lake" function by the application of a tuple of type and operation parameters.

   An unspecialized "Lake" function can be represented in "C" as a pointer to a "C" function, that is prepared to take, as its first parameter, a pointer to a structure containing bindings for its type and operation parameters. A specialized "Lake" function is represented in "C" as a closure containing a pointer to the polymorphic "C" function, as well as bindings for all its operation parameters. When a specialized "C" closure is applied to a set of actual arguments, we call the polymorphic function mentioned in the closure, supplying a pointer to the closure as its first parameter. The polymorphic function can then specialize its own behaviour by referring to the contents of the closure. No runtime support is needed for type parameters because we have restricted all type parameters to one representation.

   We declare a new closure "struct" for each polymorphic function definition, because this allows us to give types to the closure members, drastically reducing the number of type casts we have to do.

2. Type generators

   Type generators can be expanded to "C" types by applying, at compile time, the type generator parameters to the type generator definition. This translation loses the special type equivalency rules defined for type generators, fortunately all types generated from a single typegen will have the same representation (in "only pointers" "Sea"), and can be made type compatible when appropriate, through judicious application of type casts.

In addition to these two translations, a liberal sprinkling of `(void *)` casts is required to keep the "C" type-checker quiet. The resulting "C" program is:

```
typedef struct _list_poly {
  struct list *next;
  void *data; } *list_poly;

typedef struct _list_int {
  struct _list_int *next;
  char *data; } list_int;

/* The first parameter to every function is a pointer to its closure. */

/* put integer */

struct put_1_closure {
  void (*obj)(struct put_1_closure *, FILE *, char *);
};

void put_1(put_1_closure *c, FILE *f, char *x)
{
  fputs(x, f);
}

/* put list */

struct put_2_closure {
  void (*obj)(struct put_2_closure *, void *);
  void (**put_p1)(void *, void *, void *);
};

void put_2(struct put_2_closure *c, list_poly l)
{
  while(l)
    {
      *c->put_p1((void *)c->put_p1, s, l->data);
      l = l->next;
    }
}

struct put_1_closure C1 = {put_1};
struct put_2_closure C2 = {put_2,
                           void (**)(void *, void *, void *)&C1};
```

```
/* test function */
void g_1(void *c, list_int l)
{

  C2->obj(&C2, (void *)stdout, l);
}
```

Which can then be run through the host "C" compiler.

4.4.1 Why not allow integers as well as pointers ?

Integers usually have the same representation as pointers, so for most machines it is possible, without compromising efficiency, to allow both integer and pointer types to be bound to type parameters.

Two minor compatibility problems would be introduced by this change:

1. We would not be able to compile "Sea" programs on machines that used a different representation for pointers and integers. Fortunately, however, such machines are rapidly becoming extinct.
2. On a machine that has separate integer and pointer registers (ie. the Motorola 68000), a register based calling convention might load integer parameters into integer registers and pointer parameters in pointer registers. Thus if we attempted to call a monomorphic function from within a polymorphic function, passing the integer arguments as if they were pointers, these arguments could end up in the wrong registers. The solution to this problem is to pass all integer arguments as if they were pointers, but this is a nuisance.

We are opposed to this addition because it adds little power to the language while adding confusion to the language definition.

A more interesting variant would allow all the built-in types as well as pointers, to be bound to type parameters. This would provide most of the power of our truly polymorphic version, without having to support runtime variable parameters larger than a `double`. We could implement this variant by passing all parameters as fixed sized chunks large enough to hold a value of any of the basic types. See the section entitled "Why we use the stash for less-then-word-sized arguments" for a discussion of the pitfalls of this technique. (In short, this ends up being just as complicated, and inefficient as the fully general case.)

Chapter 5

Experience Using “Sea”

After implementing the prototype “Sea” compiler we attempted to write a number of libraries and sample programs to test the expressiveness of the new features. In this section we explore the limitations we ran into as well as some unexpected capabilities that emerged.

5.1 Values

The following section is an exploration of how “values” are created and manipulated in “C”, and the implications of this for “Sea”.

A “C” variable or parameter of type T is a name for a region of storage large enough to hold a value of type T.

Object : “A region of data storage in the execution environment, the contents of which can represent values ...” ANSI 1.6

“A declaration that also causes storage to be reserved for an object or function named by an identifier is a definition” ANSI 3.5

Parameter : “An object declared as part of a function declaration or definition that acquires a value on entry to the function ...” ANSI 1.6

“If a return statement with an expression is executed, the value of the expression is returned to the caller as the value of the function call expression ...” ANSI 3.6.6.4

So values of any “C” type can be stored in variables, passed as function parameters, and returned from functions.

Unfortunately "C" variables are not capable of representing very complex values. They work fine for integers, floating point numbers, pointers, and small fixed-size structures (like those that might be used to represent complex numbers). A "C" variable has a size that is fixed at compile time, cannot be very big or it will be too expensive to pass as a parameter, and can only have a very limited substructure (e.g. no recursively defined substructure).

So straight "C" is a very limited language that cannot represent complex values like "matrices" and "lists". In practice this problem is circumvented by building complex values out of multiple chunks of storage allocated from areas of memory that are not used by the "C" language system, and then accessing these chunks of storage as typed objects through "C" pointers.

While this technique does allow us to represent complex abstractions in "C", because they are being created outside of the control of the language, the language's built-in operations are oblivious to this substructure, so assignment, parameter passing and value return all operate only on the top-level value. So when the "C" programmer wants to work with a large value (like a matrix) he or she must explicitly allocate and release storage for the value, explicitly create temporaries to hold intermediate values, and take into account when the sharing of the body of the value will be a problem and explicitly make copies.

We were able to write polymorphic routines that manipulated the small (in storage requirements and complexity) values that can fit within a single "C" variable. We were also able to write routines that modified the state of large polymorphic heap-based data structures (like linked lists). However, when we tried to implement routines that operated on large heap-based values (like strings, bignums or matrices), we found that our routines became polluted with explicit storage-management operations, value-copying operations, return value conventions, parameter-passing conventions, and sharing assumptions that were different for each abstraction.

The following code samples show a number of different ways a simple sample function could be implemented in "C". Each uses slightly different storage management conventions.

```
/* If a bignum can fit within a single 'C' variable then we can implement
   ''pythag'' as follows.
*/

bignum pythag(bignum x, bignum y)
{
  bignum r;
  r=bignum_sqrt(x*x+y*y);
  return r;
}

/* Conventions :
     1) Return values are stored on the heap and deallocation is the
        responsibility of the caller.
     2) Input parameters are never modified by functions (unless explicitly
        stated).
*/

bignum pythag(bignum x, bignum y)
{
  bignum t1, t2, t3, t4;

  t1 = bignum_mult(x, x);
  t2 = bignum_mult(y, y);
  t3 = bignum_add(t1, t2);
  bignum_free(t1);
  bignum_free(t2);
  t4 = bignum_sqrt(t3);
  bignum_free(t3);

  return t4;
}

/* Conventions :
     1) A pointer to an already created bignum will be passed in for
        storing a return value.  We may also choose to stipulate that this
        output parameter not point to the same area as any of the input
        parameters.
     2) Input parameters are never modified by functions (unless explicitly
        stated).

     This version generates fewer calls to the storage manager (the previous
     version needed at least one malloc/free per function call to handle
     the return value.)

     Note :
        Although there has only been a minor change in the conventions since
        the previous version, every line of code is different.
*/

void pythag(bignum out, bignum x, bignum y)
{
  bignum a, t1, t2, t3;

  bignum_init(a);
  bignum_init(t1);
  bignum_init(t2);
  bignum_init(t3);

  bignum_mult(t1, x, x);
  bignum_mult(t2, y, y);
  bignum_add(t3, x, y);
  bignum_sqrt(a, t3);

  bignum_copy(out, a);

  bignum_free(t3);
  bignum_free(t2);
  bignum_free(t1);
  bignum_free(a);

  return out;
}
```

5.2 Constructors and Destructors

When confronted with a similar problem, the designers of “C++” created a system of overloaded constructors, overloaded destructors and an overloaded assignment operator that allows the programmer to execute a code stub every time an object—of a specific type—is created, copied, or goes out of scope. The “C++” programmer typically uses these hooks to either 1) maintain a separate deep substructure for each variable or parameter, and recover the storage occupied by this substructure when the variable or parameter goes out of scope, or 2) implement a reference counting system for the object so multiple top-level objects can share one substructure, yet the storage occupied by the substructure can be reclaimed when there are no more references to it.

'C++"s definition of constructors and destructors is complex, confusing, and full of grey areas. However, most of this complexity appears to be the result of integrating the system into an already existing language ("C"). Rather than attempting to repeat their perilous (design) journey, thereby creating a fresh set of subtle problems, "Sea" will use "C++"'s constructor/destructor system.

If constructors/destructors were applied to every object in a system the resulting program would run very slowly, much slower than if garbage collection (a more general, fully automatic, storage reclamation system) had been used. The advantage to constructors/destructors is that there is no cost unless they are actually used, which is typically for only a few types in a system. Another advantage of constructors over garbage collection is that they allow for the reclamation of system resources like file handles.

"C++" constructors are defined as part of class, whereas "Sea" constructors are defined as separate overloaded functions that take a pointer to the type they are meant to construct.

We can use "Sea"s polymorphic overloading system to define constructors/destructors that work for any type that provides the base operations required for that style of constructor/destructor. For example, the following library implements a reference-counting constructor, destructor, and assignment operator for any type that has a "`reference_count`" field.

```
forall a : void construct(a *target,
            exists void first_construct(a *), exists int *reference_count(a *))
{
  *reference_count(target)++;
  create(target);
}

forall a : void init(a *target, a *source,
            exists int *reference_count(a *))
{
  *reference_count(source)++;
  *target = *source;
}

forall a : void op=(a *target, a source,
            exists int *reference_count(a *))
{
  *reference_count(&source)++;
  *target = source;
}

forall a : void destroy(a *target,
            exists void final_destroy(a *), exists int *reference_count(a *))
{
  if(!--*reference_count(target))
    final_destroy(target);
}
```

Then when we want to define a reference-counted “`bignum`” type, we can provide just the basic definitions required by the above library. In “C++” programs, the basic reference-counting algorithm must be re-implemented for each new type.

```
typedef struct _bignum {
  int reference_count;
  char *digits;
} *bignum;

void first_construct(bignum *x)
{
  *x->digits = NULL;
}

void final_destroy(bignum *x)
{
  if (*x->digits != NULL)
    free (*x->digits);
}

int *reference_count(bignum *x)
{
  return &(*x->reference_count);
}
```

Unfortunately, constructors and destructors are incompatible with the preferred implementation strategies for “Sea”, as will be illustrated by the following example:

We can define a polymorphic version of the `pythag` function in "Sea" as follows:

```
forall a : a pythag(a x, a y,
                    exists a op*(a,a), exists a op+(a,a), exists a sqrt(a))
{
  a r;
  r=sqrt(x*x+y*y);
  return r;
}
```

After the compiler adds the implicit constructor and destructor operation parameters `pythag` will read:

(We are using the same constructor/destructor system as "C++", so in order to accurately generate the following example we defined a pythag function for a class that had a constructor, destructor, and a assignment operator, and ran this program through the "cfront" (the AT&T "C++" to "C" translator.))

(Constructors and destructors are integrated into the function calling conventions, so it is not possible to do a truly accurate source-level representation of a "Sea" function with constructors and destructors in place. For the following example we have replaced parts of the function calling and return conventions with similar source level constructs.)

```
forall a : void pythag(a *result, a x, a y,
                    exists void mult(a *, a, a), exists void add(a *, a, a),
                    exists void sqrt(a *, a),
                    exists void construct(a *), exists void init(a *, a),
                    exists void assign(a *, a), exists void destroy(a *))
{
  a result;
  a 1r;
  a V4;
  a V5;
  a R6;
  a V7;
  a V8;
  a R9;
  a R10;
  a R11;
  construct (&1r);

  mult (&R6, init (&V4, x), init (&V5, x));
  mult (&R9, init (&V7, y), init (&V8, y));
  plus (&R10, &R6, &R7);
  sqrt (&R11, &R10);
  assign (&1r, &R11);

  destroy (&R11);
  destroy (&R10);
  destroy (&R9);
  destroy (&V8);
  destroy (&V7);
  destroy (&R6);
  destroy (&V5);
  destroy (&V4);

  init (result, &1r);
  destroy (&1r);

  return;
}
```

Often, when pythag is called, no constructors or destructors will be defined for the polymorphic type `a`. (For example no constructors or destructors are defined for `int`.) In this case the operation parameters `construct`, `init`, `destroy` and `assign` will be bound to the default no-operation constructor.

A "Sea" compiler that translated polymorphic "Sea" programs into equivalent monomorphic programs (Implementation 3), could remove all these null function calls resulting in no performance degradation except when constructors were actually used. (as in "C++")

Unfortunately, with any of the "Sea" implementations where operation parameters are received at runtime, for every variable declaration, every parameter passed, every return value, and every assignment operator we would have to generate a call to a (possibly null) operation parameter. (Or, more efficiently, evaluate a conditional to determine if the call is really needed.) Thus the performance of all polymorphic functions would suffer tremendously.

5.2.1 Garbage Collection

A garbage collector does automated storage reclamation by releasing the storage occupied by all objects that a program cannot reach though any chain of pointer dereferences. Because the program had no way of accessing this data, the fact that it is not longer available can have no effect on the program execution.

Traditional garbage collectors find all live data by marking all the data accessible from the program's variables (the roots), then recursively marking all data accessible from currently marked objects. (The actual algorithms that are used are much more complex, but they are all based on this shell.)

In order for this type of garbage collector to operate it must be able to: 1) identify all the "root" pointers in program variables 2) identify all the pointers in each heap object.

In many languages pointers are located either by storing pointers in a special format that no other data object can have (tagged pointers) or by tagging every data object (including stack frames) with enough type information to locate all pointers within that object.

Tagging Data Objects

A technique described by [DMH92] for tagging the runtime stack in a GCC-based Modula-3 compiler can also be used for "C". As functions are compiled we generate a data structure that describes, for each point in a function's execution, the location of all pointers or derived pointers. When a garbage collection is then done, pointers can be located by walking up the call chain, and for each function, looking in the table associated with its current state of execution (as identified by the current value of the program counter within that function.) (There are limitations and complications associated with this technique; see [DMH92] for details.)

To locate pointers in the heap, the compiler can be modified to generate tables describing the location of pointers in every type. These tables can be associated with each type through our overloading system. We can then write a polymorphic `alloc` function that will tag each allocated block with a reference to the appropriate pointer layout table.

Static data can be tagged using a method similar to that we propose for the heap, except that the tagging will be done at compile time.

There is no way for the garbage collector to determine which access path into a union was last used. Thus we must either add a tag bit to unions or we cannot allow unions that have pointers in different locations in different arms.

If the garbage collection algorithm is designed such that pointers that do not point into a heap object are left alone, then the garbage-collected heap can co-exist with a conventional "C" heap. Members of the "C" heap would not be allowed to point into the garbage collected heap. Allowing the two heaps to co-exist is important if we want to have access to the large library of "C" code available, much of which will not play by our new rules.

"C" pointers regularly point into the middle of objects, and therefore, our garbage collector must be capable of recognising such references.

Using the above tagging information it would be possible to use most of the high-performance garbage collectors described in the literature.

Unfortunately, this technique requires substantial compiler support, so it is not appropriate if we want to use "C" as an intermediate language.

Tagged Pointers

Tagging "C" pointers is not practical because there is no way we could reserve specific bit patterns for pointers only, without severe (and incompatible) modifications to how aggregate data structures are constructed and manipulated.

In response to these problems Hans Boehm invented conservative pointer-finding garbage collectors. The observation he made was that a garbage collector can be written even with a pointer-identifier that occasionally mistakes non-pointers for pointers, and that such a pointer-identifier can be implemented simply and efficiently by checking if the prospective word points to the beginning of a heap object.

The runtime heap has to be specially constructed so that we can rapidly identify words pointing to the beginning of an allocated heap object. The way we do this is by defining a table that has an entry pointing to the beginning of every heap object. Searching this table as we are trying to identify pointers would be too time consuming, so we place a pointer in the header of each allocated object that points to its entry in the table. To check if a word points to the beginning of a heap object, we check if the pointer in the header of the object points into the table, and if it does, we check if the pointer in the table refers back into the heap object. (This is a simplification of the algorithm presented in [Boehm] that uses slightly more memory, but runs faster and does not require a custom storage allocator.)

Now that we have a way of identifying pointers we can implement a conventional mark-sweep garbage collector with every word in the stack and external variable area as the set of roots.

Because we will identify some non-pointers as pointers, we cannot use any of the garbage collection techniques that move data. (Without elaborate virtual memory tricks.) This is particularly disappointing because we would like to use our garbage collector data structures to store and retrieve data structures from secondary storage.

The addition of garbage collection makes "C" a much more powerful and pleasant language to program in. For example we can provide a much better string abstraction if we don't have to worry about storage reclamation:

```
char *concat(char *a, char *b)
{
  char *c = malloc(strlen(a)+strlen(b)+1);
  strcpy(c, a);
  strcat(c, b);
  return c;
}
```

We can then evaluate expressions like "`a = concat(concat("a", "b"))`" with no storage management considerations.

Such strings can be passed as parameters, returned from functions, and stored in variables, all without explicit storage allocation grief. See above for a discussion of why this is particularly important when we are defining polymorphic functions.

There are several problems with adding garbage collection to "Sea" :

1. Pauses in execution: There is a significant pause in execution while the garbage collector searches the heap to find and mark live data. The duration and frequency of these pauses depends on the application, computer, and amount of memory. Many "C" applications are interactive or real-time in nature and any pauses are unacceptable.

2. Significant time overhead: The exact overhead depends on the program and the amount of memory available.

3. Large memory requirements: Garbage-collected programs that do a lot of storage allocation (as is the preferred style with garbage collection) should expect to use several times more storage than they have live data.

4. Portability: The current versions of Boehm's conservative garbage collector are not compatible with the optimizations done by most modern compilers when they are told to generate "optimized code". Not using compiler optimizations on modern RISC machines leads to significant performance degradation. This is only an issue if our compiler generates "C" code (Eventually someone will add a GC-friendly switch to GCC).

For the above reasons, rather than making garbage collection a standard part of the "Sea" language definition, we propose two variants of "Sea": "Sea level 1" would not have garbage collection and would be suitable for applications where performance or real-time considerations were an issue, and "Sea" level 2 would have garbage collection, and would feature much more powerful polymorphic (and non-polymorphic) libraries.

5.2.2 Explicit Storage Management

It should be noted that even in the absence of some form of storage reclamation, many interesting polymorphic routines can still be written. For example most data-structure libraries (like lists or dictionaries) and data-structure manipulation routines (like sort) operate on a large, already existing structure, and thus can usually be implemented without much storage management.

It is also possible to write routines that work for any type that use one specific style of storage allocation, and then standardize on this style for most types. For example, we can define "pythag" for any type that has a storage "release" routine:

```
forall a : a pythag(a x, a y,
                    exists a mult(a,a), exists a add(a,a),
                    exists a sqrt(a), exists void release(a))
{
  a t1, t2, t3, t4;

  t1 = mult(x, x);
  t2 = mult(y, y);
  t3 = add(t1, t2);
  release(t1);
  release(t2);
  t4 = sqrt(t3);
  release(t3);
  return t4;
}
```

This is much less elaborate (and limited) than the full constructor/destructor scheme, but it does not create the same implementation problems.

5.2.3 Implementation Selection

As has already been discussed, we propose 2 levels of the “Sea” language 1) without garbage collection, suitable for limited memory environments or applications with real-time constraints and 2) with garbage collection, a much more powerful language with much more powerful libraries. We define these as two different variants of the language so that the real-time programmer can know which libraries are safe for his or her applications.

Earlier, we introduced a variant of “Sea” in which only pointer types can be bound to type variables. A compiler for this variant can be implemented as a translator to “C”, and such a compiler could probably be written in 15000-20000 lines of portable “C” code.

Because of the elaborate calling conventions and data-structure layout rules required for an efficient, unrestricted implementation of “Sea”, a compiler for such a language has to compile directly to the target assembly language. To make such a compiler portable to a wide range of machines requires considerable additional work. Our current “Sea” compiler is portable and, with some tuning, can generate high-performance code. It is implemented as 175000 lines of “C” code. (almost all of this is stock GCC).

Which implementation is preferred depends on the application. The applications of early, experimental versions of “Sea” are as follows:

1. Developmental : Experiment with and refine language features.

   “Sea” is one of the first languages to use “overloading polymorphism”, so as the language is used it is expected that the language design will evolve rapidly. This rapid evolution makes now a bad time to invest too heavily in implementation technology.

2. Evangelical : Allow other programmers/researchers to experiment with this style of programming.

It is much easier to get other people to experiment with a small, reliable, totally portable pre-processor than it is to get them to install and experiment with a 175000 line compiler.

3. Software Development : Use to construct actual programs that are meant to be useful in their own right.

   If someone writes a program in "pointers-only" "Sea" their program will be easy to port to any environment that has a "C"-compiler. All that is required is that the user first compile a small, totally portable pre-processor.

   A program written in full "Sea" will be dependent on a 175000 line compiler that takes considerable time and space (20Mb) to bring up on one of the supported architectures.

So for early, experimental versions of "Sea"-like languages we would argue that the pointers-only implementation is more appropriate. We make this argument because the implementors of this project feel that it would have been a more useful experiment to have written a small portable compiler, and experimented with language features, than to have devoted so much effort to making the compiler fully polymorphic.

5.3 Language Usage

5.3.1 Polymorphic Data Structures

The following is a sample polymorphic data-structure implementation:

```
type list(type elem)
{
  return struct {
    list(elem) *next;
    elem data; } *;
}

type hashnode(type key, type contents)
{
  return struct {
    key k;
    contents c; } *;
}

type hashtable(type key, type contents)
{
  return struct {
    list(hashnode(key,contents)) *table;
    int size;
    int (*hash)(key); } *;
}

forall key,contents:
  hashtable(key, contents) create(int hashsize, int (*hash)(key))
{
  hashtable(key,elem) h;
  h = alloc(1);
  h->table = alloc(hashsize);
  h->size = hashsize;
  h->hash = hash;
  return h;
}

forall key,contents:
  int lookup(hashtable(key,contents) d, key k, contents *ep,
        exists int compare(key, key))
{
  int hash;
  list(hashnode(key,contents) l;

  hash = d->hash(k) % d->hashsize;

  for (l = h->table[hash]; l != NULL; l = l->next)
    {
      if (compare(l->data->k, k) == 0)
        *ep = l->data->c;
      return 1;
    }

  return 0;
```

```
}

forall key,contents:
  void insert(hashtable(key,contents) d, key k, elem e)
{
  ...
}

forall key,contents:
  void print(hashtable(key,elem) d,
         exists void print(key), exists void print(elem))
{
  ...
}

forall key,contents:
  void destroy(hashtable(key,contents) d,
               void (*free_key)(key *), void (*free_contents)(contents *))
{
  int i;
  list(hashnode(key,contents) l,t;

  // Free storage occupied table, hashlists, hashnodes and hashtable
  for (i=0; i<h->size; i++)
    {
      for (l = h->table[i]; l != NULL; l = t)
        {
          t = l->next;
          free_key (&l->k);
          free_contents (&l->c);
          free(l->data);
          free(l);
        }
    }
  free(h->table);
  free(h);
}
```

Notes:

1. This data structure will work with "pointers-only" "Sea". In order to accommodate the "only pointers" restriction, we couldn't store the key and contents directly in the list node. This forced us to add an extra level of indirection, with the associated extra overhead and extra storage management activity.

2. The garbage-collected version of "Sea" is not required in order to use this abstraction. We have had to parameterize the hashtable destroy function with storage release functions for the types stored in the table.

3. We could have made the "hash" function an "exist" parameter. But in that case it would be more difficult if we wanted to use a different hash function, for the same type, in a different context (perhaps we want a different hash function for a keyword symbol table that we do for an identifier symbol table). Because "Sea" currently lacks nested functions the only way we could define two different overloaded `hash` functions for the same type would be to compile them in separate "`.c`" files, each with `static` visibility. (This is also why the `free` function parameters to `destroy` were not made into `exist` parameters.)

4. The `print` function above is an example of a type of overloaded utility function we propose providing for all built-in types and library data structures. It is expected that many data structures in "Sea" programs will be stored using the polymorphic data-structure libraries. If such data structures could be read and written in text and binary this would be a substantial convenience for the programmer. (Note how easily functions like `print` can be defined for complex data structures in terms of the `print` functions for their member types. For example to print a hashtable where each element was also a hashtable would require no additional code.)

5. All the basic operations required for the "hashtable" functions are provided by the "hashtable" type generator. This makes it very convenient to pass a polymorphic hashtable as a function parameter. (Compare this with "Abstract Polymorphic Data Structures" described below.)

Polymorphic versions of all the basic data structures can be similarly defined. This single addition makes "C" a much more powerful language.

5.3.2 Abstract Polymorphic Data Structures

Abstract polymorphic data-structure parameters are defined entirely in terms of their operations. Using this technique we can write polymorphic routines in terms of the characteristics they require from their data-structure parameters, rather than in terms of a specific data structure.

For example, the following browse routine will operate on any "sequence" that can be stepped through in both directions:

```
forall seq,pos,elem: void browse(seq s, pos p,
   exists pos next(seq, pos, elem *),
   exists pos prev(seq, pos, elem *),
   exists void format(char *, elem),
   exists int browse_view(elem))
{
  ...
}
```

Using this browser we can browse:

```
Array of menu options
Doubly linked list of field definitions in a database structure editor.
B-tree of filenames matching search criterion
Lines in the password file
Student records from a database server
Functions in a `C' file
Chunks of ``help'' text from a ``help'' file
Filenames in current directory
A copy of the operating systems run-able process queue
```

Abstract input/output devices have similar broad application.

5.3.3 Polymorphic "printf"

In [OCD 92] a polymorphic variable-argument-length `print` function is described. The following is a translation of this function into "Sea":

```
void print()
{
  return;
}
```

```
forall a,b: a print(b x, exists void put(b), exists a print)
{
  put(x);
  return print;
}

int main()
{
  put(4)(5.0)("red");
  return 0;
}
```

Unfortunately, because our overload resolution algorithm is restricted so that it will only consider external function definitions or `exist` parameters (in order to allow for programs to be transformed so that all closures are constant), this program will not compile under "Sea".

For "Sea" we will borrow the "streams" package from "C++".

Chapter 6

Conclusions

We have integrated "overloading polymorphism" with "C"; no fundamental incompatibilities were encountered.

In most polymorphic programming languages, all values are constants on a garbage-collected heap, and function parameters and return values are pointers to these constants. Because all function parameters and return values have the same representation (a pointer) regardless of type, polymorphic calling conventions for these languages are not complex.

"C"/"Sea" is defined in such a way that passing a parameter to a function or returning a value from a function involve passing the value directly (usually by copying it). This is a substantial contributor to the high performance of "C", because we don't need an additional level of indirection to access every value. Defining an efficient polymorphic calling convention that can work within the constraints of "C" was a major challenge.

Our new calling convention results in 1) slightly worse performance for monomorphic functions, and 2) polymorphic functions that, apart from the effects of no inlined arithmetic functions, perform not much worse than equivalent monomorphic functions.

We had expected to use "C++"-style constructors/destructors for storage management. However, it was discovered that if we were generating polymorphic object code, we would not know until runtime which type parameters had constructors/destructors defined for them. Allowing for this flexibility at runtime would have had a large negative impact on the performance of polymorphic "Sea" functions. So the only storage management alternatives available for "Sea" are explicit storage management or garbage collection.

In early experimentation with the "Sea" language we found we were able to write define a large class of useful functions without having to specify them in terms of a specific type. The principal problem that arose was that the list of operation parameters was cumbersome to construct, easy to get slightly wrong (thereby limiting the domain of the function), and not very useful to a human reader attempting to determine whether a specific type has the required operations.

Bibliography

[OCD 92] Ophel, J., Cormack, G., and Duggan, D. *Combining Overloading and Parametric Polymorphism in ML* Draft Copy, 1992.

[Aho 86] Aho, A.V., Sethi, R., and Ullman, J.D. *Compilers—Principles, Techniques, and Tools.* Addison Wesley, 1986.

[Cardelli 85] Cardelli, L. and Wegner, P. *On Understanding Types, Data Abstraction, and Polymorphism.* Computing Surveys 17:4, Dec. 1985, 471-522.

[Cormack 90] Cormack, G., and A.K. Wright. *Type-dependent Parameter Inference.* Proceedings of ACM Sigplan 90 Symposium on Programming Language Design and Implementation.

[Stallman 88] Stallman, R. *Internals of the GNU C Compiler.* Free Software Foundation.

[Stroustrup 90] Stroustrup, B. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[Wright 86] Wright, A.K. *Reference manual for the language ForceOne.* Masters Thesis, University of Waterloo, 1986.

[Ditchfield 92] Ditchfield, G. *Cforall Reference Manual* Phd Thesis, University of Waterloo, In process.

[DD 85] Donahue, J. and Demers, A. *Data Types Are Values* ACM Transactions on Programming Languages and Systems, July 1985. Pages 426-445.

[Boehm 87] Boehm, H and Weiser, M. *Garbage Collection in an uncooperative environment.* Software: Practice and Experience, vol 18, pages 807–820, Sept 1988.

[DMH 92] Diwan, A., Moss, E. and Hudson, R. *Compiler Support for Garbage Collection in a Statically Typed Language.* Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, pages 273–282.

[ANSI 90] X3J11 Technical Committee *Programming Language C — ANSI X3.159–1989* American National Standards Institute, 1989.