Type-safe Stack Traversal for Garbage Collector Implementation

Colin Stebbins Gordon
Brown University
colin@cs.brown.edu

ABSTRACT

Garbage collectors are an important part of many modern language runtimes. Essentially all tools for developing and debugging programs using garbage collection assume the correctness of the collector, and therefore provide no means for detecting garbage collector errors. As a result it is especially important that garbage collector implementations be free of errors. This goal is even more challenging in the face of the typical implementation strategy for collectors: implementation in C, making error-prone inferences from complex bit patterns, where an error could result in dereferencing an invalid pointer or corrupting program data.

One approach to reducing errors in collector implementation is to improve both the type-safety and memory-safety of garbage collector implementations. Prior work [8, 17] in this direction has focused on the use of modern type systems to statically detect errors in the collector code at compile time, but has practical shortcomings. The prior work replaces the standard machine stack with a heap allocated data structure to avoid unsafe walks of the native stack. Traversal of the runtime stack is normally not possible in higher-level languages because they trade the flexibility of arbitrary memory access — typically used to gather a root set from a runtime stack — for the safety of being unable to cause memory access errors.

We present a method for addressing the safe stack traversal problem at the compiler level, by lifting actual machine stack frames up to the level of explicit data structures in Standard ML, such that complete stack traversal can be performed with minimal unsafe code. We implement a garbage collector in the ML Kit [14] using the techniques described and provide details on key parts of the implementation.

General Terms

Garbage collection, Compilers

1. INTRODUCTION

Garbage collectors serve an important role in many programming languages by abstracting away memory management. For collectors to work effectively, they must manage memory without the knowledge of the running program. To do this safely, they must preserve and interpret unspoken invariants of the user program. There are basic safety issues such as not replacing pointers with null or prematurely collecting memory still in use. Violations of some of these invariants, such as preserving return addresses, are relatively easy to detect. A more subtle safety property to guarantee when writing a collector is that it must preserve the types of the program it is collecting: it must not replace a pointer to an instance of a T_1 with a pointer to an instance of a T_2. This kind of bug in a collector would likely be very difficult to track down; debugging tools make the assumption that the runtime is correct, so a bug at this level would not be visible through, for example, a Java debugger, because such tools are designed to hide the internal representations used by the garbage collector.

Most collector implementations primarily use C in order to allow arbitrary memory access. This code is unsafe, and a mistake there can easily remain unnoticed until a runtime collection when a collector bug may crash the program or corrupt data. Not only is the collector code itself usually unsafe, but it can be one of the most significant sources of unsafe code in a language runtime; the Singularity Project at Microsoft [11] mentions that garbage collector code accounts for roughly half of all unsafe code in the Singularity source base.

High level languages, such as members of the ML family, offer a great deal of support for reasoning about the correctness, particularly the type safety, of user programs. It would be useful to extend this reasoning capability to garbage collectors by making them explicit ML functions, and allowing greater reasoning power than we have for the typical C implementation of a collector.

Previous work [8, 17] established basic type-safe approaches to collecting the heap, but did so by side-stepping root set collection through methods that turn the stack into a heap-allocated data structure. This transforms the user program into what is essentially a literal implementation of an abstract machine, rendering many existing stack-based debugging tools useless. Rather than removing the machine stack from consideration, what we would really like is a type-safe way to walk the native machine stack. We propose compiler support for a type-safe stack traversal that allows traditional debugging tools to continue working.

```
let gc[$\rho_{src}$] roots[$\rho_{src}$] =
  letregion $\rho_{dest}$ in
    let newroots = copy[$\rho_{src}$, $\rho_{dest}$](roots) in
      only $\rho_{dest}$ in
        continue_with newroots
```

Figure 1: Rough code for a basic collector in Wang and Appel's system

The paper proceeds as follows. Section 2 describes the use of regions for safe memory management, as well as prior work leveraging region-based memory management for safe garbage collection and limitations to these approaches. Section 3 describes an alternative way to safely walk the stack by lifting the native machine stack to the level of an explicit structure in a safe language, and Section 4 describes one implementation of this. Sections 5 and 6 discuss some problem areas of the current design with possible solutions, and some added benefits which fall out of the current implementation of lifted stack frames. Finally, Sections 7 and 8 discuss related work and future directions for this research.

2. SAFE GARBAGE COLLECTION WITH REGIONS

Tofte and Talpin [15] describe a safe translation from the call-by-value lambda calculus to a runtime system based on a stack of regions (areas of machine memory from which allocations are made). Tofte and Birkedal [13] propose an inference algorithm that determines a safe and reasonably efficient set of regions for a source program and prove it sound with respect to inference rules in the earlier region work.

Central to the region-based runtime's intermediate language is the notion that all pointers are typed with not only the standard data type (e.g., `int list`) but also with the region of memory in which they are allocated: their region type. Regions are allocated using the `letregion` ρ `in` e construct, which allocates a region ρ whose lifetime is lexically scoped as the execution of e. Allocations are annotated with the region in which the new value should reside. Functions may be parameterized over one or more regions into which allocations for results should be made or where arguments are allocated. The region identifiers are also lexically scoped, so there is no way to refer to a value allocated in region ρ outside the body of e (assuming that region aliasing is disallowed). Wang and Appel [17] point towards several specific region type systems.

Wang and Appel [17] build upon an extension of the region-based lambda calculus to implement a simple copying garbage collector in an extension of Standard ML. The idea is to extend the region calculus with an `only` ρ `in` e construct for early deallocation (before the conclusion of a `letregion`'s body). This not only deallocates regions other than ρ prior to the execution of e, but also hides the region type for any deallocated region when type-checking e. Mutator programs are implicitly parameterized over a single region, and the garbage collector performs a pivot between semi-spaces and resumes execution.

Figure 1 shows nearly-accurate code for a collector in their system, written in their extension of Standard ML. The code intentionally leaves the `roots` and `continue_with` vague, but is otherwise correct. The function begins by receiving a representation of a root set allocated in the current semispace, ρ_{src}, then allocates a new region, ρ_{dest}, to act as the the destination semi-space (this also creates a new region type, of things allocated in ρ_{dest}). It then copies the root set into a new root set representation, allocated in the destination semispace, using the `copy` function, which works as expected. Finally, it deallocates the source region (and the corresponding region type) using the `only` construct, and continues the program's execution with the new roots.

This alone achieves data type safety for the collector because we have the root set available as a Standard ML data structure, but the region type system also allows other errors to be detected statically. For example if the new root set contains a reference to anything still allocated in the source region, a type error will appear during type checking. The error would be detected because the new root set's region type would contain a reference to the region type of ρ_{src}, which is undefined inside the body of the `only` construct in this case.

It is also important to note that while ρ_{dest} is lexically scoped to exist only in the body of the `letregion` construct, the tail call out of the `letregion` skips the generated code which would perform the deallocation. Therefore ρ_{dest} continues to exist after the execution of this call. Another important point is that the `only` construct is not merely a deallocation, but also an assertion that no dangling pointers exist. If this construct is omitted here, `newroots` may contain references to values in ρ_{src}. This omission would also permit ρ_{src} to continue existing, which means that no collection actually occurs (though some amount of data would be duplicated).

The description of the root set structure and the method for continuing the program's execution after collection were left under-specified above for simplicity. This is because the issue of collecting a root set and updating the root set in a language runtime is generally a difficult portion of code to write, implemented in (unsafe) C. To sidestep this issue and gain type-safe access to the root set, Wang and Appel CPS and closure-convert the mutator, turning the program into a single, tail-recursive dispatch function on a heap-allocated structure representing the current continuation at the time collection began. The root set passed to the `gc` function is this structure, and the `continue_with newroots` in Figure 1 is actually a tail-call to the dispatch function.

The stack is made explicit for the collector, so no unsafe code is required to gather the root set. The result, however, leaves existing debugging tools useless; there is no machine stack for such tools to examine.

Wang and Fluet [8] build a similar system in Cyclone [12], a safe dialect of C. Their collector works only for a Scheme interpreter they write, which is a literal implementation of an abstract machine for Scheme. Thus the stack they need to traverse is already a Cyclone data structure, and their system also avoids walking the machine stack.

3. ANOTHER APPROACH TO SAFE STACK TRAVERSAL

Every garbage collector needs to begin collection by finding the root set — the live pointers on the runtime stack — in order to later find all reachable heap-allocated objects. Typical collector implementations need to traverse the native machine stack one word at a time, and follow somewhat

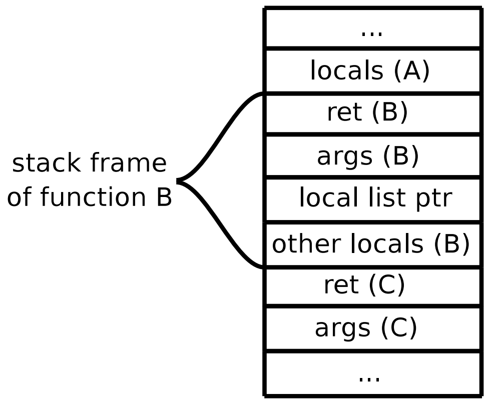

Figure 2: The stack of a typical ML Kit program where function A calls function B which calls function C. The stack grows downwards, as on the x86 architecture.

complicated reasoning schemes to determine whether a word is a heap pointer, and if so to determine the type of the referenced object. Because the words on the stack are untyped, the only practical way to work with them for finding the root set in the past has been to use an unsafe language permitting arbitrary memory access, usually C. These inferences require a lot of very particular reasoning about very specific invariants, and their implementation is error-prone.

The prior attempts at safe collectors just discussed brush this issue aside with a somewhat undesirable solution in order to focus on other important issues. The general solution chosen by Wang and Appel [17] and by Fluet and Wang [8] is to represent the program stack as an explicit heap-allocated structure in the language used to write the collector. This solution has the advantage of providing explicit structures in a safe language, which gives the collector writer many static guarantees that the C stack traversal lacks. However because they no longer use the normal machine stack, extra overhead is incurred for function calls (heap allocation is generally slower than stack allocation), and stack-based tools such as standard debuggers cease to be of any use.

The core of the problem is that while traditional C-based collector implementations tend to treat the stack as nothing more than a series of unstructured, untyped machine words, the stack actually has a structure — it just lacks an explicit definition. The calling conventions used by the compiler for a language define the structures that might arise on the stack. To make them explicit, all that is needed is a set of definitions for the structures — against which other code can be written — and slight changes to the compiler to generate these definitions and massage the stack frames to match the normal representation for those data structure definitions.

Consider the stack shown in Figure 2, which is typical of a program compiled by the ML Kit compiler[1]. Note that it does not use the standard x86 frame layout supported by the `call-enter-leave-ret` idiom on the x86. It has a rough repeating structure, where each stack frame consists of the address to which the call should return, the arguments for

[1]See the `FUNCALL` case of the `GC_ls` function in `src/Compiler/Backend/X86/CodeGenX86.sml` of the ML Kit source.

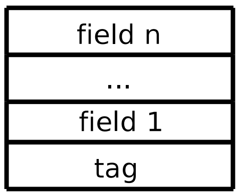

Figure 3: A generic tagged representation of a structure. The tag is at the lowest address in memory, and is generally the word whose address is stored in pointers to the structure.

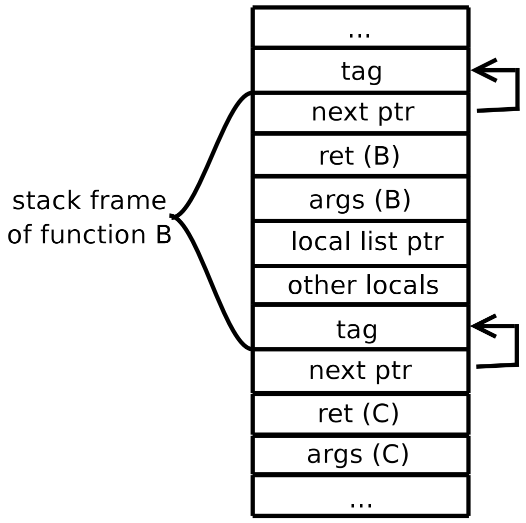

Figure 4: A rough depiction of how lifted stack frames are set up out on the stack.

that call, and some set of local variables, pushed onto the stack in that order. The list pointer shown in the figure is part of the local temporary storage, and will be discussed in an example shortly.

Compare this rough frame structure to the generic depiction of a tagged data structure shown in Figure 3. The words of memory for the return address, arguments, and local storage are very similar to the members of a user-defined structure. The only things missing are the tag, and a guarantee that the way data is stored on the stack otherwise exactly matches the way some definable data structures are laid out in memory.

Figure 4 shows the idea behind the layout of lifted stack frames, where the frames themselves have had their structure slightly modified in a relatively unobtrusive way to match the layout of user-definable data types. Each frame has been augmented with a tag and a pointer to the next stack frame. At the top of the stack (not shown) is a frame tagged as a final frame. The placement of the next-pointer at the "end" or top of the frame is largely because it is easy to place there by simply pushing the stack pointer onto the stack, though Section 4.2 gives an additional reason. When some function A calls function B, it will push a tag for the current state of its own frame onto the stack, tagging itself. Next it will push the stack pointer, which will serve

```
fun count 0 = 0
  | count n = 1 + count (n - 1)
```

Figure 5: A simple function that has simple stack frames for recursive calls

```
datatype stackframe =
  ...
  | frame_count2count0 of int * word * stackframe
  ...
  | lastframe
```

Figure 6: Example of a stack frame data type

as the next-pointer for B's frame. Finally it will push the return address and arguments for B, and actually call B. B then manages its own local storage, and when it calls another function, C, it will tag its own current frame state and do the same basic setup as before for the tail of what will become C's stack frame.

We must add a way to acquire a pointer to the most recent frame, generate an explicit type for each frame whose in-memory layout would match the layout of that stack frame, and make sure we tag each frame correctly. Traversal can then occur safely by looking at pieces of each frame's structure, and moving safely to the next frame by following a normal pointer stored there. The stack is now a chain of slightly different, but related, structures, strongly resembling a recursive ML variant data type in which each variant refers to another instance of the same overall type. This is how we will define the stack frame structures.

Generation of equivalent user types is straightforward, though it requires carrying some explicit type information down as far as the platform-specific code generator, while in the past this information might have been removed from the intermediate representation much earlier in compilation.

We will assume that the type being generated is called `stackframe`, and it is being constructed as a normal ML algebraic data type, of the form shown in Figure 6. Words that represent integers, etc., will be shown as such in the types. For opaque data, such as return addresses or unused local storage, it should suffice to insert a `word` into the type. For the pointer to the next frame, a member of type `stackframe` in the data type is sufficient.

Consider the code for the function shown in Figure 5, a simple function that recurs `n` times, eventually returning `n`. Assuming that all arguments are passed on the stack, the stack frames for calls to `count` should be fairly simple; they will contain the argument, a return address, and a pointer to the next stack frame. The type that might be generated for frames of this function is shown in Figure 6 (the name is explained in Section 5). This is for the simplest case, when no local storage and no registers are saved across function calls.

```
fun B x =
    let val lst = primes_up_to x
        val len = List.length lst
    in
        (lst,len)
    end
```

Figure 7: Code for B, which has local storage without a valid value at one time, and with a valid value later

More complex stack frames may require multiple distinct frame types at different times. When there is local storage that must be preserved, a new frame variant must be generated, and the compiler must generate code to tag it appropriately at runtime. If portions of local storage become live or dead between calls made from the same frame, each different set of live variables on the stack requires a new type. For example, function B's frame in Figure 4 has a list pointer in the set of local variables, as might occur if B made one call to get this list, and then passed it into another call later. Suppose B is the function shown in Figure 7. B first makes one call to get the primes up to x, then another to get the number of primes in that list, and finally returns the pair. Upon initial entry to B, the space for storing the `int list` returned from the first call is allocated on the stack, but the value there is garbage — just whatever happened to be in that word of memory. The `stackframe` variant that will be used for this frame when making the first call must not allow this garbage value to be used, and should use a member of type `word` for that space on the stack. The variant for B's frame after the first call, which will be used and tagged accordingly when making the second call (to get the list length), should then use the correct actual type, `int list`, because the value at that stack location is now valid.

It is also important to note that for a garbage collector to function properly, it must be able to determine the exact types of all roots, as an `int list` is handled very differently by the collector than an `int list list`. This means that the generated `stackframe` variants must be entirely monomorphic. Support for typing and tagging frames of polymorphic functions should be compatible with this approach to stack traversal, and is discussed extensively in Section 4.3.

4. IMPLEMENTATION OF LIFTED STACK FRAMES

This section describes one particular implementation of lifted stack frames, done inside the ML Kit with Regions [14] compiler, for a first-order subset of Standard ML. Implementation details would vary for other compilers, but most of the details shown here should be generally applicable or should at least provide insight applicable to other compilers.

4.1 Finding the First Frame Pointer

This approach to making stack frames explicit sounds promising so far, but there is no existing way to get the stack pointer in ML, nor is it necessarily the case that the current stack pointer indicates the start of a valid `stackframe`. To remedy this, we must provide a simple primitive to hand off a valid `stackframe` to ML code. An implementation of this is shown in Figures 8 and 9. Figure 8 shows a natural way to write the C code to locate what should be the first valid `stackframe`, and Figure 9 is the foreign function interface declaration to call it from ML. This required small changes to the ML Kit's foreign function interface, to allow the return of `stackframe`s. Previously only simple types and lists

```
extern int DLabGC54GC;
int getframe(int x) {
    int *p = &x;
    while( (int*)(*p) != (p+1) ||
           (int*)p[2] != (p+3) ||
           p[1] != DLabGC54GC) {
        p++;
    }
    p++;
    assert(p[0] == DLabGC54GC);
    return (int*)p;
}
```

Figure 8: C code to find first valid `stackframe`

```
fun getframe () : stackframe = prim ("getframe", ())
```

Figure 9: ML Kit foreign function declaration to get stack frames

could be returned, largely because the ML Kit's region inference system lacked support for reasoning about where other structures might be allocated. This is discussed further in Section 4.5.2.

The ML Kit represents boxed data types in two pieces [14]. The first is a two-word piece whose first word contains the tag for the appropriate variant of the data type (used for pattern-matching) while the second word is a pointer to the second piece. The second piece is simply the fields of the data type variant, in order, one word at a time. Members that point to other boxed data structures are pointers to the tag word of the first piece of that structure's representation.

The C code in Figure 8 relies on this layout to find the first valid stack frame. It walks up the stack one word at a time until it finds a two pointers, one word apart, each of which points to the next word on the stack, where the word between them (the tag) has the same value as a global variable `DLabGC54GC`. The current implementation places the two pieces of the boxed structure adjacent in memory, as shown in Figure 10, and as compared to the types the collector will see in Figure 11. This results in the particular pointer pattern for which the code searches. The compiler inserts an extra instruction at any call to the garbage collector that places the tag for that calling frame into the global variable, so we return the stack pointer for the first valid frame above the collector's frame; we do not want frames inside the collector itself because any roots found there are not necessarily roots for the mutator. If this approach to stack traversal were used outside the realm of garbage collectors, either this variable or an additional global variable might need to refer to the most recent frame of any kind.

This is not safe code, and therefore not ideal, but is fairly simple for a programmer to inspect for bugs assuming the programmer understands the in-memory layout for frame structures placed on the stack.

The C code presented for finding valid `stackframe`s detects one of the few remaining invariants the compiler must enforce, which is the result of the specific representation of boxed data types in the ML Kit and the way we arrange them on the stack. Together these effectively provide a dis-

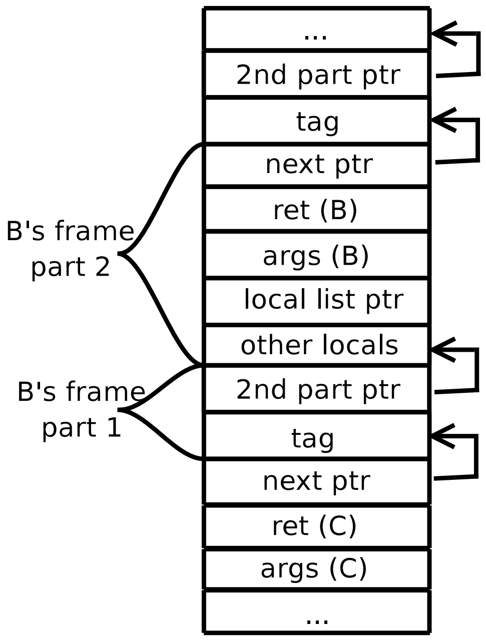

Figure 10: Stack frame layout in the ML Kit with lifted stack frames implemented

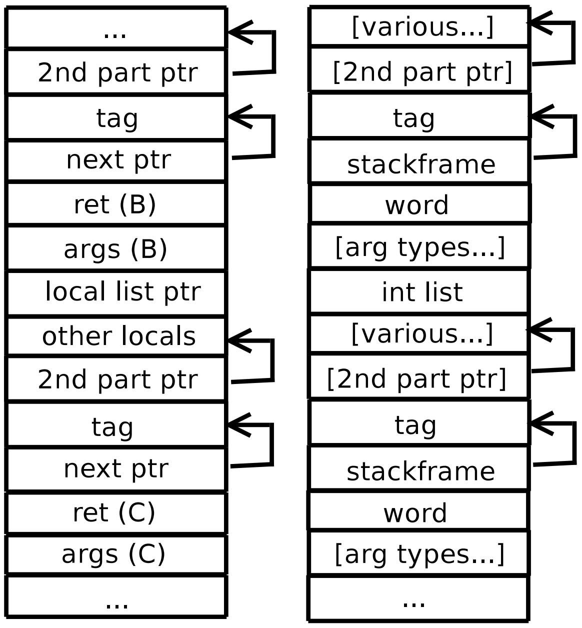

Figure 11: Physical stack layout compared to in-memory representation of algebraic data type

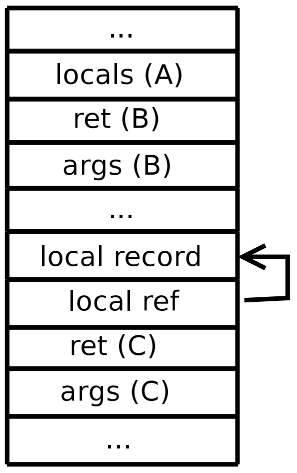

Figure 12: A stack frame with one local pointing to another

```
extern int* DLabGC54GC;
int* getframe(int x) {
    return DLabGC54GC;
}
```

Figure 13: Final version of the `getframe` primitive

tinctive three-word tag (the next-pointer, actual tag, and second-part pointer). There are a couple weaknesses in looking for this invariant. First, this invariant does not generalize: other compilers whose data type tagging does not result in such a peculiar structure would not be able to use similar code to find valid frames. The general tagged frame layout in Figure 4 shows a representation that might be identical to the literal layout of data structures in another compiler. In this case, a stack frame such as that shown in Figure 12 would appear to be an extra frame. The second problem is actually an extension of the first: a doubling of the first case could still register incorrectly as a stack frame in the current system. We have not placed any restrictions on the frame yet that make this a safe invariant for which to search: a pointer to a locally-allocated single-member record with the same value as the collector-caller frame tag followed by a similar structure would appear to the code in Figure 8 as the desired frame. Thus it seems prudent to disallow allocation of objects on the stack, in order to prevent pointers on the stack into the stack from causing false positives when searching for frames.

But we would like to avoid this restriction if possible; even without needing to handle invariants which are different for various `stackframe`s, searching for this layout invariant makes the code fragile and echoes the problematic invariant-checking of old collectors. Figure 8 shows a natural approach to finding the first stack frame based on a traditional stack traversal, but it is not robust. Is there a better solution? Yes: remember that we already cache the relevant frame tag in a global variable which only needs to be updated when the collector is called, because we do not need (or want) any frames inside the collector. Instead of storing the relevant tag when the collector is invoked, we will instead store a pointer to that frame. Because the `getframe` primitive is only needed inside the collector, this pointer will then be up to date whenever we look for the stack, and never needs to be modified when we return from the collector. Figure 13 shows the final implementation for the stack retrieval primitive, which no longer relies on the peculiar tagging structure our tagging implementation produces.

As it turns out, for another reason explained in Section 4.2, we must still disallow pointers from the stack into the stack (most easily done by allocating no actual structures on the stack). But the code for retrieving stack frames is now very short and extremely simple.

4.2 Updating the Stack

While safely traversing the stack and gathering information about it is useful in its own right, this work is in the context of a basic copying garbage collector, which requires the ability to replace pointers on the stack when it moves objects.

Figure 14 shows the code to update the stack from a fully-constructed, heap-allocated copy of the stack. It first ensures that it has sane pointers for both stacks. Next, it ensures that both pointers refer to the same type of stack frame. The next check determines whether we have reached the end of the real stack (a check explained momentarily). In this case, we return the ML value for true to signal a complete collection. Otherwise, we check that we indeed have a valid "second-part" pointer on the machine stack, find the next-structure pointer on the stack, and copy everything but the next-pointer from the heap copy to the real stack. Then we repeat the process.

To mark the end of the machine stack, it is set up so that the tag for the `lastframe` variant appears twice in a row at the absolute top of the stack. This check would have a low chance of a false positive if the tag for `lastframe` could possibly be the same as the pointer to the second part of a structure. Because of the way the ML Kit generates tags, this cannot occur. The tags generated for structures will never have the two least significant bits both zeroed, whereas the pointer to the second part of a structure will always be word-aligned and therefore have zeros for the two least significant bits[2]. Every variant for each type will therefore have a different tag. If we detect that we are at the end of the machine stack, we are also at the end of the copied stack (remember, we already ensured we have the same variant for each stack, and we now know this variant was `lastframe`).

The dependence on finding a valid next-frame pointer in order to determine how much memory to copy is a definite weakness. Because of this, we still cannot allow stack allocations of the sort shown in Figure 12 (a local pointer referring to the word immediately above it on the stack) to arise. The simplest way to do this is to ban allocation of objects on the stack, and force them to be placed on the heap.

Another limitation of the current code for updating the stack is that it returns false if an error occurs during the update. This is easy to ignore, and if our larger goal is to make garbage collector bugs as easy to catch as possible then a proper implementation should not allow these errors to be discarded. The current implementation does not have working exception support, but a complete implementation

[2]See the code for `tag_con0` and `tag_con1` in `src/Compiler/Backend/BackendInfo.sml` in the ML Kit source.

```
int installframe(int *newp) {
    int words = 0;
    int *sp, *np;
    int *stackp = getframe(0);
    while (1) {
        /* Some sanity checking... */
        if (stackp == NULL ||
            newp == NULL ||
            stackp == newp) {
            goto error;
        }
        /* If these aren't the same variants, that's bad. */
        if (*stackp != *newp)
            goto error;
        /* If we have the same variant, and the next stack
         * word is the same as the tag, we're at the end of
         * the stack! */
        if (stackp[0] == stackp[1])
            return mlTRUE;
        /* If the word after the tag doesn't point to the
         * following word for the stack pointer, that's
         * bad. */
        if (stackp[1] != (int)(stackp+2))
            goto error;
        /* Follow to second part of structure, which will
         * be replaced */
        stackp = (int*)(stackp[1]);
        newp = (int*)(newp[1]);
        for (sp = stackp;
             *sp!=(int)(sp+1);
             sp++) {
            words++;
        }
        memcpy(stackp, newp, 4*words);
        stackp = sp+1;
        newp = (int*)newp[words];
        words = 0;
    }

error:
    return mlFALSE;
}
```

Figure 14: Code to update the stack from an updated copy

should raise different exceptions for each of these error conditions. This way if one of these problems does appear at runtime, some useful information about the failure can be conveyed, but not so easily ignored. Alternatively we could simply print a message to the terminal and exit the program directly from the C code, as these are likely not to be errors from which we can recover.

Another criticism of this update solution is that it requires a separate copy of the stack. In a language that supported destructive updates to members of algebraic data types, this update code could be done safely as well, in the actual collector language. One intuition for a solution to this might be to make traced members of the variants references (SML `ref`s) to the appropriate type. Unfortunately references produce a pointer to a mutable pointer (allowing shared references), rather than a direct pointer. It is possible that some adjustment to the compilation of references or management of local pointers to heap objects might make this possible in the ML Kit. In any case, mutating the frame structure's members individually rather than replacing them entirely would likely complicate the region type system if one were used in conjunction with this approach. The members' region types would change mid-function, and this would need to be tracked. Some of the future work in Section 8 discusses this.

Another subtlety of this code is that it makes certain assumptions about the copy function. Specifically it assumes that any opaque data (e.g., `words`) is copied simply from place to place, so the copy-and-update is functionally equivalent to not having modified those locations. This includes unused temporary space that might not need to be changed, which is a weakness of the current implementation.

This precise copying also includes — for region-polymorphic functions in the current implementation — region pointers. When we copy data to update the stack we rearrange data underneath the region inference system present in the ML Kit, without its explicit knowledge: specifically we change the region in which a pointer's target data is allocated without updating the region pointers. This is a shortcut in the current implementation, and Section 4.5.2 explains why this is acceptable.

4.3 Handling Polymorphism

One issue left unaddressed thus far is that of tagging polymorphic functions' stack frames. There are no concrete types for some of these functions' arguments and intermediate values, but to be able to traverse the stack properly, the garbage collector must know the exact types of everything in the activation record for a polymorphic function, at every invocation. Fortunately, the site of any call to a polymorphic function determines the types for that invocation.

Because the tagging for frames (the actual placing of tags into the stack) occurs at runtime, and we would like to only have one instance of the code, we need a way to determine the tag corresponding to the correct types of a polymorphic frame at program run time. We know that the concrete types of the calling function uniquely determine the concrete types for the called polymorphic function, and that the calling function's frame is already tagged with a monomorphic type. Therefore to tag frames for polymorphic functions, we can use the calling function's tag as a representation of the types of arguments passed in, which allows us to determine the concrete types for this new polymorphic function's frame. We should note that this is not currently implemented, but suggested as one compatible extension to the frame type generation to support polymorphism.

Let us again consider Figure 4. If B is a polymorphic function that calls some other function C, it must tag its own frame when this call is made. If its own calling function A is tagged with a `stackframe` variant for a call site where A calls B, passing only `int`s, then the exact types for all polymorphic variables in B become known, and B can tag itself with the appropriate frame type, which was generated at compile time. This means that frame type generation must create monomorphic frame types for every call site of a polymorphic function for every way it is invoked in the source program. This is similar to the implementation strategy of flattening all polymorphic code and calls into monomorphic code, but here it occurs only with generated types, and need not occur with actual mutator code present after compilation. The collector code size still increases because it must handle all of the types, regardless of how much code is responsible for them.

Goldberg [10] suggests an entirely dynamic approach to this type reconstruction, determining which functions are further up the call chain, locating the frame of a monomorphic function, and reconstructing types in later calls from that information. This has some disadvantages discussed in Section 7. We prefer a static lookup table embedded in the object file (or rather, a lookup table for each call site within a polymorphic function) with a short, simple lookup whenever a polymorphic function calls another function. This table must map from parent-function tags and current-function call sites to current-function tags. Each parent-function tag determines the types for the current polymorphic function, but the current call site within that function must still be considered to handle such things as changing sets of live variables in the frame between outgoing calls. In practice, because we already generate a new `stackframe` variant for every call site, the call site is not an extra concern, and the table is simply a direct tag to tag mapping.

This approach imposes an additional restriction on the generation of `stackframe` variants. Whereas previously if two function call sites had identical `stackframe` variants the two could be merged, with this approach to supporting polymorphism this tempting optimization is no longer always possible.

The polymorphic tagging implementation now adds an additional meaning to the variants, which was intuitive but not strictly necessary before. These otherwise-identical variants now identify two distinct call sites in the program, which may pass different types to any polymorphic function(s) called. The most recent stack frame may look the same and have the same types at two different call points, but those points may still pass arguments of different types to the same polymorphic function. This unique identification of call sites is not represented explicitly in the variant's definition, only in the fact that it is declared and defined separately from other variants. Therefore an additional consideration this implementation requires when generating `stackframe` variants is that the types of the arguments passed at a call site must also be considered when generating the set of variants for calls to polymorphic functions, in order to properly perform what is essentially a flattening of the polymorphic types into sets of monomorphic types.

```
fun f args* = body
```

Figure 15: User code before GC safe-point added.

```
fun f args* = (maybegc (); body)
```

Figure 16: User code after GC safe-point added.

4.4 Compilation Changes

The current implementation requires two rounds of compilation to use the frame types. The first round compiles the user program with dummy versions of the `stackframe` type and any functions that use it. It generates all of the stack frame types, and emits them in a form the collector's implementor can use. The second round uses the actual stack frame types and related code, the latter of which must currently be written by hand. It also places accurate tagging information in call sites in the resulting object code, as the types were not present in the normal sense during the first compilation, and therefore proper tags could not be generated. This design is not ideal, and improvements are discussed in Section 5.

4.5 GC with Lifted Stack Frames

This section provides an example of how lifted stack frames might be used when implementing a collector. We describe a collector we have implemented using lifted stack frames and careful manipulation of the region inference support present in the ML Kit.

4.5.1 GC Safe Points

As with many collectors, the only safe time for the garbage collector to run is at function entry, which ensures that no new roots have been added other than the function arguments (already on the stack) and that there are no roots present only in registers (for example, the return value from an allocation call).

We assume the source program has been transformed into a form where each function of the form seen in Figure 15 has its body prefixed with a call that might garbage collect, with the resulting form seen in Figure 16. This is actually only necessary when entering calls that will allocate additional memory, but we will assume all mutator functions are transformed like this for simplicity. This is similar to the safe point approach taken by Wang and Appel [17].

4.5.2 Building a Collector Over Region Inference

The current system does not catch dangling pointers in copied state as Wang and Appel do [17]. This would require implementing extensions to Standard ML as described in that work, and time constraints prevented this for the current publication.

A working collector is implemented without that static check, but using the mostly-type-safe stack traversal and update described here. Instead of the `letregion` and `only` language extensions we rely on the ML Kit's underlying region inference system, described by Tofte and Birkedal [13]. We have written a proof-of-concept collector function, which copies the entire state of the system in such a way that the region inference algorithm allocates this in a new region. We then use a modified version of the ML Kit's `forceResetting`

```
local
    val prevframe = ref NONE
in
    fun gc () = let val f = getframe ()
                    val f_copy = copy f
                in
                (forceResetting f_copy;
                    let val f_new = copy f_copy
                    in
                        prevframe := SOME f_new;
                        if installframe f_new
                        then print "Install success!\n"
                        else print "Install ERROR\n"
                    end)
                end
end
```

Figure 17: Code for a collector relying implicitly on region inference.

primitive to empty the original regions, and copy the result back into a persistent global region. This approach is not optimal, and done under time constraints to demonstrate that lifted stack frames can be used as the root set in a working collector.

Figure 17 shows the code for our collector. There is a global pointer to the previous root set, `prevframe`, which is visible only to the garbage collector. This exists to force the region inference system to place new copies of the program state in a global region that will not be deallocated when the collector, or any function higher up the call chain, returns. Any value stored here will be inferred to be in a persistent global region by the region inference algorithm.

The collector itself begins by retrieving a pointer to the current stack above the frame of the garbage collector using the `getframe` primitive. Remember that because it returns the value of a global location that is updated at every invocation of the collector, we will have an accurate cached pointer for the frame that called the collector.

Next we make a copy of the current state of the system (including the stack), which will be inferred to be in a new local region for two reasons. First, the copy function is written such that it places the new copy in a different region from its source. All that is necessary in order to have region inference choose this allocation scheme is to write a copy function that makes a complete and total copy. Leaving any dangling pointers back to the original state will cause the region inference algorithm to infer that the source and copy should be in the same region. The ML Kit provides options for printing region-annotated intermediate forms of the program after region inference. The inferred region type for the `copy` function shows whether it leaves a dangling pointer. If the regions for the input argument and those for the result are different, a complete copy has been made. If they are the same, `copy` leaves a dangling reference into the source state. This does not cause a static error, and does not give any information as to where in the copy function this dangling reference is left, but there is information available to check. The second reason the copy is local will be explained shortly.

At this point we make a call to a modified version of the `forceResetting` primitive provided by the ML Kit. Normally this primitive resets (empties) all regions used by anything accessible from the value provided as its argument (all regions in its region type) regardless of safety. We have modified it to instead reset all regions not in its region type[3], and so it acts to preserve only its argument's regions and clear all others. In either case, all regions continue to exist, but those emptied no longer contain data.

After deallocating the original state, we copy the local version of the state back into the global region. None of the region change is explicit. The only reason this new copy will be allocated in the original global state region is because we update the `prevframe` reference to point to this new copy. If we had not updated this reference, region inference would have allocated this second copy in a local region as well, which would have been deallocated when the collector finished; the region inference algorithm has not been modified to be aware of the special status of the stack. If we had instead updated this pointer to refer to the first copy we made, the first copy would have been allocated in the same global region as the original. But we need them in separate regions to preserve one and throw away the other (along with any data in other regions). This is the second reason the first copy is inferred to be in a local region: because the first copy is created, but never escapes the scope of the collector. Therefore, when the collector finishes executing, the data from the second copy — now held in a global region because of the `prevframe` pointer — will persist, but our temporary copy will be deallocated. This double-copying is unfortunate, and expensive. However it is necessary because we are still relying on region inference to generate actual allocation and deallocation points. It is important to keep in mind that this collector is only a proof of concept, and using lifted stack frames with region-based memory management in a real system should include a more thorough integration with the region system or whatever other memory management primitives are exposed to the collector language.

After ensuring the final copy will not be deallocated at the end of collection, we update the stack with the `installframe` primitive described earlier. This replaces now-stale pointers into now-empty regions with valid pointers into what is likely a different region. This means we are technically invalidating the region type of these pointers. While this seems dangerous, in practice it is actually safe because the region types are used mainly to ensure that memory is not deallocated too early, and we are actually keeping it around longer than necessary by anchoring it in a global region. New allocations from a certain scope that were destined for a certain region will still be placed in local regions after collection. These may end up pointing into the longer-lasting global state from the most recent collection.

Eventually the global copy of state from collection time may have a pointer into a new local region if it previously contained a reference. When the region it points into is deallocated, this will be a dead pointer, which must not be dereferenced. Fortunately, this is not a danger as long as the collector does not attempt to directly handle the previous collection's root set — remember, this previous root set is

[3]Actually, we reset all but the earliest couple regions. The compiler's startup code for executables allocates some data in several global regions and then stores pointers to those allocations in global variables that are inaccessible to ML code. A complete implementation would require collecting from these roots as well, but that is outside the focus of the current work.

only clearly accessible to the collector because the `prevframe` pointer makes region inference store it in a global region. While it may appear to be a root, `prevframe` alone does not mean the mutator can access that memory.

Consider the case where a reference exists in the root set installed at the end of a collection, and the reference is later updated to point into a region that will be deallocated upon return from a certain call. The only way this pointer from the global collection state into a local temporary region can develop is if a reference is still reachable from the current (post-collection) stack. If another collection occurs after all frames from which that reference is reachable have returned, then because it is no longer reachable from the current, active stack, it will not be traversed by the copy function. If a collection occurs while that reference is still reachable from some live stack frames, then assuming region inference is implemented correctly, the region the globally-located reference points into would be reachable in a system with region inference alone. This means that the local region would not have been deallocated yet. When the copy function copies the active root set, it will follow a path from a live stack frame to the still-accessible portion of the previous collection's root set containing the reference. Then it will follow the reference to the still-living local region, and copy data out for the temporary version of the program state.

5. SHORTCOMINGS AND SUBTLETIES

The most obvious limitations of this work are that it requires a hand-crafted collector for every program, and that it requires two rounds of compiling to occur (once to generate the frame types, and once to include them). An alternative is to automate the two passes, automatically adding the types, automatically generating the copy code (in the source language, in this case ML), and re-checking and compiling everything together. This automatic generation is similar to what the ML Kit already does for eliminating polymorphic equality tests (generating statically-typed equality checks for every data type whose equality is tested) [7, 14] and somewhat similar to the automatic generation of specific GC code as suggested by Goldberg [9]. Unlike the elimination of polymorphic equality in the ML Kit, this code would need to be generated much later, after most of the compilation has already occurred, requiring a second type-checking phase.

Another obvious negative of this approach is the additional overhead on function calls. For the current implementation, three additional words must be pushed onto the stack for calls, and returns require an operation on the stack pointer to remove these three words. This could be reduced by having each function pre-allocate its tags on the stack just once at function entry, adjusting stack offsets appropriately, and overwriting the tag word when calls are made instead of pushing. Either way, the overhead is likely to be less expensive than the allocation of a continuation structure on the heap as is done by Wang and Appel [17]. Of course, a thorough performance evaluation would be necessary to confirm this.

A related set of concerns stems from the mere presence of these additional words on the stack. One minor concern is the fact that these extra words increase the chances of overflowing the stack for a given stack size, so a larger stack size may be necessary for some programs using this approach. Because of the very fine-grained manipulation of these words on the stack, this approach would also be extremely difficult to apply to a compiler targeting a C backend, as opposed to targeting assembly. While this is possible in C, this direct manipulation would be much more difficult. The resulting C would also no longer be portable, losing one of the main advantages of compiler backends targeting C.

A less obvious, relatively minor, difficulty in implementation is that the way in which the names for the ML `stackframe` variants are generated for the mutator code must not be sensitive to changes in the garbage collector code. So for example, simply naming the variants `"frame1"`, `"frame2"`, etc., when the number depends on the order in which the compiler encounters invocation points, may not work. If the compiler generates code for the garbage collector before other functions, then adding code to the collector will likely change the correct variant names for mutator frames. This could necessitate tedious rounds of edits doing nothing but correcting frame names. The solution is to make frame names for the mutator stable across compiles.

For this work we name each frame using the name of the calling function, the name of the function being called, and a number indicating which call in the body of the calling function the frame referred to. The frame type variant for the recursive call to `count` shown in Figure 5 is `frame_count2count0` because it is for a call inside the body of the `count` function, it is a call to the `count` function, and it is the first call (to any function) in the body of `count`. The problem does not arise again for collector code itself because there is no need to generate frame types for it; the execution of the collector should not trigger the collector again, so those frame types need not even be present. Simply not generating the types for garbage collector calls would be enough to avoid this problem, but this approach aids debugging as well by giving the variants meaningful names.

This approach for garbage collection also requires rethinking support for separate compilation. With standard approaches to garbage collection, it is only necessary to have the type signatures of functions from other object files. Lifted stack frames require knowing all function invocation points in a program, as well as the types of any intermediate values that might be stored on the stack at those points, and having either the data type definitions or existing copy functions for every data type used within pre-compiled code. This is far more information than usual, and conflicts with some of the goals of separate compilation, such as the distribution of libraries without sharing source. Even with this call site information, the binary file would need to already include code for tagging stack frames; these tags and the names for those frame variants must not overlap with those generated for the mutator code that calls the library. With source available this can be handled by simply recompiling all code together, but using binaries with unknown source is another matter which requires further consideration. It is possible that the linker could handle this, by placing external references in the library object code, and having programs compiled against such libraries emit static entries to satisfy these references. The tags could then be resolved when the executable is linked with the library. Polymorphism would of course further complicate matters.

Another concern is limiting use of a stack frame pointer to legitimate uses. For example, we do not want to permit the mutator to have access to stack frames for reading or updating. Placing the GC code in another module where it could be hidden by a module signature could solve this. However that would still allow misuse within the collector code itself (for example, stashing a stack pointer in a module-local variable and re-using the old pointer during subsequent collection is highly unlikely, but not disallowed). Adding linear types to the collector language and specifying that all stack frames and frame elements are linear could detect many such errors — more than one use of a stack frame by any single function would cause a type error. This also helps with the need to correctly copy opaque data for collection, as mentioned in Section 4.2, by requiring that each part of the frame be either copied over into the new frame or deep-copied (swapping of data of the same type would still be allowed by this scheme). Of course, switching to automatic generation of the actual collector code rather than having it hand-written would also reduce the likelihood of such errors. The two approaches together would be desirable.

The implementation described here does not address any issues of pointer sharing between data structures; this includes cyclic data structures. It is our belief that the approach outlined here can be integrated with the "encoding" forwarding pointer implementation described by Wang and Appel [17]. This approach requires adding an explicit forwarding pointer (a reference to an `option` containing a structure of the same type, located in another region) to each data structure. The mutator only receives access to a version of the structure where that member contains a reference to `NONE`. The collector code has access to another version of the structure where the reference may point to other values, and during collection it may perform a safe upcast from the user variant to the collector-only version. New values are constructed using the proper mutator-visible versions.

One larger point which may be raised about this work is that while it purports to reduce the complexity of the collector, a number of extensions to support writing these safe collectors have been proposed (`letregion` and linear types to name a couple, in addition to the general compiler changes proposed by this work). Each of these requires additional code to implement. At a minimum we are trading unsafe code for code that may be implemented in a safe language. Additionally we are trading a great deal of complexity in one area which benefits one subsystem (garbage collection) for possibly simpler code that enriches the language supported by the compiler with additional useful features. Even the unsafe portions of code for generating polymorphic tag lookup tables and doing the lookup itself are fairly simple. The only code which would be relatively complex is that for generating monomorphic types for all calls to polymorphic functions, which is likely equivalent in implementation complexity to alternatives [9, 10], and is safe code.

With polymorphic tagging support, it is possible that the number of types generated for a program that uses many instantiations of relatively few polymorphic functions and structures may result in a larger garbage collector than mutator. A relatively simple program that makes use of a polymorphic hash table implementation would demonstrate the possible discrepancy between mutator size and collector size. Each instantiation of such a hash table could generate a large number of frame types after flattening the polymorphic instantiations. An analysis of how many variants are generated for realistic programs, compared to the size of the corresponding collector, should be considered future work.

There is also a possibility that our proposal for handling polymorphism ends up being more costly than Goldberg's. While we would pay a small overhead at each polymorphic function call, Goldberg's solution is likely a larger cost each time it runs, but that cost is paid only for each collection.

The update function's design prohibits some allocation optimizations. Specifically, disallowing pointers on the stack into the stack restricts allocations as mentioned earlier. Normally when the ML Kit can determine the maximum size of an inferred region at compile time, it converts that region into a stack allocation. With the current implementation, such an optimization is unsafe, and disallowed.

The last few shortcomings of this work are mostly incidental to the current implementation. One limitation of the current collector is that we are limited to a simple copying collector because of the dependence on regions for memory allocation and deallocation. Other memory management primitives would allow lifted stack frames to be used with other kinds of collectors. We also do not currently handle (off-the-stack) global variables. These are outside the focus of this work (safe stack traversal), but should certainly be handled by any robust implementation.

6. OTHER IMPLEMENTATION BENEFITS OF LIFTED STACK FRAMES

Once using lifted stack frames, many of the complex or subtle invariants that the compiler previously had to maintain for the garbage collector are no longer necessary. In particular, we now assign precise types to every word on the stack that must be traversed by defining and tagging structures that represent the stack. The tagging for this occurs at the frame or structure level, and every important word on the stack is contained within a structure. Therefore tagging individual words on the stack is no longer necessary. This includes pointer tagging, and other tags such as tagged integers. This allows for such things as increasing the range of integers without increasing their size. Some code can also execute slightly faster because there are no longer data types that require additional bit manipulation to work with. Goldberg describes other advantages to removing (or in this case, reducing) the number of tags in memory [9].

7. RELATED WORK

The most directly related work is clearly that of Wang and Appel [17]. As was described earlier, this work inspired the current line of research, as it performed undesirable transformations on the source program to make the root set explicit. Later work by Fluet and Wang [8], also mentioned earlier, takes a similar approach in another system, but does not encounter the same stack issue because the collector is written for a Scheme interpreter whose stack is already an explicit heap-allocated data structure. Both of these also implement forwarding pointers for handling structures pointed to from multiple locations. As mentioned in Section 5, the forwarding-pointer approaches used in these systems should be perfectly compatible with the work done here, and in a complete system the two should be combined. Ideally this integration would also include use of other aspects of these systems, namely the use of region types to disallow dangling pointers. This would require the explicit `letregion` and `only` constructs as well.

Some other directly related work is that of Vanderwaart and Crary [16], which presents a typed assembly language for use as an intermediate representation during compilation.

The typed assembly language makes stack frame types and stack manipulation explicit. Also described are a frame descriptor that gives information about traceability of entries in an activation record, and the construction of a GC table mapping function return addresses to information about active parts of a frame. Their goal was to define an interface against which a garbage collector could be verified, rather than to reduce collector bugs by making garbage collection explicit in a source language. Their frame descriptor format relies on lists indexed by integers, whose interpretation and generation are error-prone. Their system also requires closure-converting source programs in order to translate them to the typed assembly language, which is undesirable for reasons similar to those which make performing CPS conversion on source programs undesirable. The approach described here, while currently only implemented for a first-order subset of ML, should be straightforward to extend to higher-order languages by tagging and emitting explicit types for closure representations as well.

Conservative garbage collectors [2, 3, 4, 5] are designed to maintain safety by interpreting every word of memory as a potential pointer to an allocated portion of memory. By also explicitly tracking the allocated pointers and sizes of allocations, conservative collectors can determine whether any word, if interpreted as a pointer, might point into an allocated portion of memory. They avoid the need to know about internal structure of user data types by treating the appropriate number of words referenced by a valid allocation pointer as potential pointers to traverse. Because the collector cannot actually distinguish between an integer and a pointer without some form of tagging, conservative collectors are restricted to mark and sweep algorithms, and cannot move heap objects. This is in effect the reverse limitation of the work presented here and by Wang, Appel, and Fluet [8, 17], which rely on the copying to change the region-types from what are effectively "pre-collection" to "post-collection" types. Because of this and the lack of any form of explicit type information, the notion of safety for a conservative collector is very different than in a traditional collector. Conservative collectors also take the opposite approach to safety from this work; they attempt to preserve safety through extremely conservative action while remaining invisible to higher levels, while this work attempts to lift the primitives for garbage collection to explicit representation at higher levels in order to statically detect errors.

Much of this current line of work is focused on tagging stack frames when function calls are made. An alternative approach might be suggested by the work of Appel [1] and of Goldberg and Gloger [9, 10] on how to garbage collect without explicitly tagging structures. The general approach is to generate program-specific collection code, embed code pointers for specific GC routines into carefully-chosen places in the generated object code, such that when the collector is invoked from within an allocation function, the code pointer for the appropriate GC routine is located and invoked. Polymorphic functions are handled by traversing the stack to find an earlier frame where all types are known statically, and the types for later frames are then reconstructed from that at runtime. This is then used to call a frame-specific GC routine, which was generated at compiled time. There are a couple significant disadvantages to such an approach: the type reconstruction is potentially expensive and occurs at runtime, but more importantly it still relies on the ability to carefully use unsafe code to traverse the stack. Removing the unsafe code would make this tag-free approach roughly equivalent to this work — a safe stack traversal used to invoke frame-specific collector code.

Goldberg's runtime type inference has the potential for introducing significant overhead inconsistently for programs that make extensive use of polymorphism. The overhead for the approach described in Section 4.3 is near-constant-time per function call while Goldberg's is a variable-cost operation performed only during collection. Ours is likely better-suited for situations when unpredictable collection pauses are unacceptable, but as mentioned in Section 5, there is not a clear winner in the general case. Future work should include a thorough investigation of this tradeoff.

Goldberg's reconstruction algorithm is also complex to implement, which conflicts with the goals of this work: to reduce the overall amount of unsafe code in the compiler and runtime, to make the remaining unsafe code as simple as possible, and to make any new and possibly complex safe code as generally useful as possible. The implementation of lifted stack frames described here relies heavily on re-using existing pattern-matching code from the compiler, which must already be correct for Standard ML, in handling stack frames. The new code introduced is all either relatively simple (`getframe`'s C backing in Figure 13), totally safe, or of potentially broader use than just for the garbage collector (`letregion`, `only`, and if chosen for controlling access to stack frames, linear types).

8. FUTURE WORK

This line of work is far from complete. The most immediate next steps would be to implement the polymorphic tagging support described in Section 4.3 along with proper region inference integration, and perform some analyses on the resulting system. Specifically, the complete system should be compared to others, including that of Wang and Appel [17], a standard collector, and a version with a Goldberg-style polymorphic type reconstruction algorithm, to see what the overhead is in terms of executable performance, executable size, and memory consumption. Another useful analysis would be to look at the number of frame variants generated by the polymorphic support, and see what effect this has on the garbage collection code required. For these measurements to be worthwhile, lifted stack frames would also need to be integrated with the forwarding pointer approach described by Wang and Appel [17] to preserve pointer sharing (and avoid looping forever on cyclic structures). Eventually this work could also be extended to include support for closures. Working from an earlier starting point, extension of the memory management approach in Wang and Appel's collector to other sorts of collectors (mark and sweep, generational) should be investigated.

Another desirable next step would be to implement automatic generation of the copy functions and other GC code, such that they are then type-checked with the rest of the program again. This shifts the collector-writer's task from writing the collector to writing the collector-generator. This may seem like we have moved to another unverified batch of code, but because the generated GC code would be type-checked with any mutator program, many errors would be caught statically when one compiled a program aggravating the GC-generator bug, just as if the collector were hand written. In addition to this, most of the code that would need to be generated would be basic structure-copying code, making this shift a reasonable one.

There are also a few more advanced directions to explore. Parallel collectors such as that described by Boehm [2] create GC safe points similar to those used in this work, and with forwarding pointers implemented, this could potentially be fairly straightforward. Real time applications of this may be much harder, but would likely fare better than Goldberg's work [9, 10], whose run time type reconstruction may yield unpredictable and potentially large delays. Finally, while one of the main arguments for this line of work versus that of Wang and Appel was to avoid rendering existing debuggers useless, this work might also be leveraged to enhance debugger functionality; the `stackframe` variants provide a great deal of useful information for debuggers, tagging the exact call site and types for any function invocation.

Lastly, much of this work has been intimately tied to the particular compiler in which we implement our system. In particular, stack frame layout has been particular to the ML Kit's use of the stack on the x86 architecture. As alluded to in Section 3, the ML Kit does not use the architecture-provided instructions for implementing functions. The x86 architecture manual recommends that compiler implementors use instructions that treat the `ebp` register as a "back-pointer" to the previous stack frame, pushing its value and copying the stack pointer into the register at each function call [6]. These recommended calling conventions manage what is essentially a chain of frames, but they ensure that the chain is always a simple series of `ebp` values with implicit locations relative to the rest of the frame. It would be possible to naïvely implement the tagging approach described in this work alongside this functionality (resulting in effectively two independent lists), but it would be desirable to reuse this hardware functionality if possible. Whether this can be done is not immediately obvious.

One last peculiarity of this work, which might be examined, is the stack traversal's dependency on algebraic data types and pattern matching. This is crucial for accessing various parts of the stack safely, as we then reuse the existing structure traversal code already present in the compiler. Translating this idea to an object oriented language such as Java seems promising, as objects have a header that is essentially a tag. Instead of variants of one algebraic data type, each frame could be an object implementing some `Stackframe` interface containing a `collect()` method. Each object's collect method would perform the appropriate copying on each of its fields, and then call the `collect()` method of the next frame through dynamic dispatch. Basically, collection would be performed using a visitor pattern initiated by a call on the most recent stack frame. Of course to understand any limitations with that approach, it must be tried.

An added benefit of doing this is that most object-oriented languages support destructive updates to member fields, which removes the need for unsafe update code like that from Section 4.2. As mentioned there, this would also complicate the region type system if this work were to be integrated with that approach to safe memory management in collectors. Still, investigating this and other ways to reduce or eliminate the unsafe code involved in the stack update would be an important next step.

9. ACKNOWLEDGEMENTS

Many many thanks go to, above all, my advisor Shriram Krishnamurthi, who first lured me in the direction of research with tantalizing promises of verifying correct functionality for traffic lights. I am exceptionally grateful for this (not to mention his guidance of my technical writing and presentation skills, and with this particular project). Thanks also to Aleks Bromfield and Andrew Oates, who embarked with me on the three-month literature-survey "odyssey" that eventually pointed towards this project. I should also thank Martin Elsman, ML Kit maintainer, for giving me a kick start on understanding memory layout of ML Kit data types. Thanks also to Nong Li, Aurojit Panda, and the countless visitors and faculty candidates who sat through my "brief" spiels earlier in the semester, and whose comments and questions during those presentations helped me learn how to get a point across more efficiently and effectively. Further thanks go of course to my reader (and advisor on another project), Maurice Herlihy, and also to the rest of Brown Computer Science's faculty and students for helping me get to this point. This work was supported in large part by a grant from my parents.

10. REFERENCES

[1] Andrew W. Appel. Runtime Tags Aren't Necessary. In *Lisp and Symbolic Computation*, volume 2, pages 153–162, 1989.

[2] Hans-Juergen Boehm. Mostly Parallel Garbage Collection. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 157–164, 1991.

[3] Hans-Juergen Boehm. Space Efficient Conservative Garbage Collection. In *Proceedings of ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 197–206, June 1993.

[4] Hans-Juergen Boehm. Simple Garbage-Collector Safety. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, pages 89–98, 1996.

[5] Hans-Juergen Boehm and Mark Weiser. Garbage Collection in an Uncooperative Environment. 1988.

[6] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*, volume 1. Intel Corporation, June 2005.

[7] Martin Elsman. Polymorphic Equality — No Tags Required. In *Second International Workshop on Types in Compilation*, Kyoto, Japan, March 1998.

[8] Matthew Fluet and Daniel Wang. Implementation and Performance Evaluation of a Safe Runtime System in Cyclone. In *SPACE Workshop*, 2004.

[9] Benjamin Goldberg. Tag-Free Garbage Collection for Strongly Typed Programming Languages. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 165–176, 1991.

[10] Benjamin Goldberg. Polymorphic Type Reconstruction for Garbage Collection without Tags. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pages 53–65, 1992.

[11] Galen C. Hunt and James R. Larus. Singularity: Rethinking the Software Stack. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 37–49, 2007.

[12] Trevor Jim, Greg Morrisett, Dan Grossman, Michael

Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, June 2002.

[13] Mads Tofte and Lars Birkedal. A Region Inference Algorithm. In *ACM Transactions on Programming Languages and Systems*, volume 20,5, pages 724–767, July 1998.

[14] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. Programming with Regions in the MLKit (Revised for Version 4.3.0). Technical report, IT University of Copenhagen, Denmark, January 2006.

[15] Mads Tofte and Jean-Pierre Talpin. Implementation of the Typed Call-by-Value λ–calculus using a Stack of Regions. In *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, pages 188–201. ACM Press, January 1994.

[16] Joseph C. Vanderwaart and Karl Crary. A Typed Interface for Garbage Collection. In *Workshop on Types in Language Design and Implementation*, pages 109–122, 2003.

[17] Daniel C. Wang and Andrew W. Appel. Type-Preserving Garbage Collectors. In *28th Annual ACM SIGPLAN – SIGACT Symposium on Principals of Programming Languages*, pages 166–178, 2001.