Dartmouth College

Dartmouth Digital Commons

Dartmouth College Undergraduate Theses

Theses and Dissertations

5-26-2003

Using Low Level Linear Memory Management for Type-Preserving Mark-Sweep Garbage Collector

Edward Wei
Dartmouth College

Follow this and additional works at: https://digitalcommons.dartmouth.edu/senior_theses

Part of the Computer Sciences Commons

Recommended Citation

Using Low Level Linear Memory Management
for Type-Preserving Mark-Sweep Garbage
Collector

Edward Wei

Advisor: Chris Hawblitzel

Dartmouth College

Department of Computer Science

05/26/2003

Abstract

Efficient low-level systems such as garbage collectors need more control over memory than safe high-level languages usually provide. Due to this constraint, garbage collectors are typically written in unsafe languages such as C. A collector of this form usually resides as a trusted primitive runtime service outside the model of the programming language. The type safety of these languages depends on the assumption that the garbage collector will not violate any typing invariants. However, no realistic systems provide proof of this assumption.

A garbage collector written in a strongly typed language can guarantee not only the safety of the garbage collector and the program being garbage collected (mutator), but also the interaction between the collector and the mutator. Removing the garbage collector from the trusted computing base has many additional benefits: Untrusted code could be given more control over memory management without sacrificing security. Low-level code such as device drivers could interface in a safe way with a garbage collector. For these and the growing prevalence of garbage collectors in the typical programming system necessitate a safe solution.

Previous research by Wang et al introduced a safe copying collector based on regions, where the live graph structure of the heap is copied from an old region to a newer region. This paper seeks to improve the efficiency of type-preserving garbage collection with the introduction of a type-preserving mark and sweep garbage collector.

1   Introduction

More of today's applications rely on trusted low-level infrastructure. These systems perform critical tasks that can tolerate few failures. One would expect these important systems to be especially sound. Efforts have been made in

2

the community to establish clear standards and guidelines for creating bug free code. Due to the high probability of human error, these can only be met with limited success. Other approaches have involved active run time testing. This is also useful only to a certain extent as run time tests cannot examine every possibility of execution. Any bugs missed by human or mechanical intervention could be a potential for undesirable behavior. Just a few lines of incorrect low level code could make the difference between a secure system and an insecure system, successful commerce and failure in down time. Ideally, we want systems for which we can assert certain correctness properties.

However, the usage of such safe high level languages in systems development is rare. Programmers lean towards using unsafe languages like assembly or C because they need more efficient low-level control of memory. For example, Java includes the length of an array in a header word for array bounds checking. Not only is this an inflexible form for an array, there is extra run time overhead to check for buffer overflow. The goal of this paper is to give type-safe code more control over memory in order to implement a mark-sweep collector. We utilize static type-checking techniques in order to reduce overhead.

To demonstrate the practicality of our ideas, we have implemented our type-preserving mark-sweep garbage collector in a safe C-like language called *Clay*. The *Clay* compiler typechecks the garbage collector code to verify its safety, and then translates the code into ordinary C++.

2   Background

In order to discuss the data structures of the garbage collector, we must take a detour into language theory. We will return to the garbage collector in section 4. The language Clay used in this paper is statically checked. A set of properties encoded into a Clay program can be checked at compile-time. These

3

set of properties can be useful for early detection of program bugs and also provide a soundness guarantee that your program will behave accordingly. The language tools we describe below allow us to be expressive with our types to encode such properties.

2.1   Linear Types vs. Nonlinear Types

In a program with complex data structures, memory management can be difficult. To be safe, an explicit "free" operation must guarantee that there is only one reference to the memory being freed. If multiple references exist, dangling pointers would leave a potential hazard the next time those pointers are used.

Linear types can solve this problem. Linear types [4] are types whose values may be used exactly once. Linear types have only one reference that is consumed with the use of the linear value. Linear types can neither be duplicated nor discarded. Nonlinear types are types whose values can be used in the traditional sense. Nonlinear values can be referenced by any number of principals, and can be duplicated or discarded. In Clay, we denote linear variables with *@type*, and nonlinear variables with *type*.
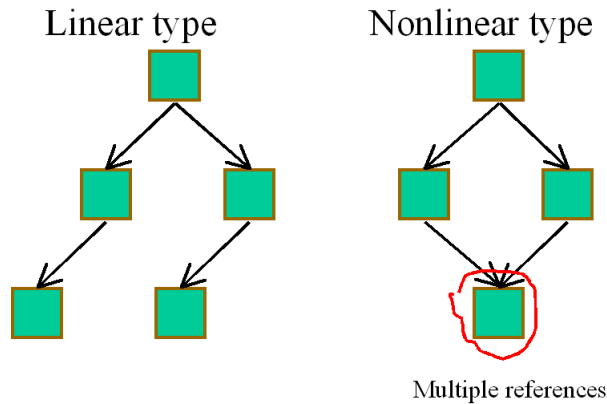
With linear types, values may be used exactly once:

```
void discard(@type T);¹

...

discard(car);

// Error! car is no longer in scope.

drive(car);

...

// This function may not be written.

Pair[T,T] duplicate(@type T);
```

Since linear variables can only have one reference, no garbage collection is needed. Nonlinear types suffer from the multiple aliasing problem and thus require garbage collection. If the world was linear, garbage collection would be a trivial task and this paper would not be needed. However, natively, linear types can only implement trees! (and not all data structures are trees). Furthermore, linear data types can be difficult in practice to use. Especially for low level typed assembly language and compiler generated proof carrying code, nonlinear data types must be supported at the low level in order to express the nonlinear data types in the high level language.

2.2   Regions and Type-Preserving Collector

Linear types are simply too inflexible. Nonlinear types aren't safe when you need to "free" to deallocate the type. One synthesis of these two types is regions [6]. Regions are linear types that contain other possibly nonlinear types. More specifically, a region is an unbounded area of memory where objects/types can be allocated. Regions serve to group objects with similar lifetimes. When a region is deallocated, all the objects it contains are deallocated with it. We will

---

[1]Clay-like pseudo code

see how to implement regions later in our discussion of linear memory. For now, we will use pseudocode to demonstrate the concept of regions.

```
let r = newregion()     returns a new region
alloc(τ at r)           allocates type τ in region r, returns type "τ at r"
store i in x            stores value i in variable x
freeregion(r)           frees region r
```

Objects may be allocated into different regions.

```
let r = newregion();
let r1 = newregion();
...
let x = alloc(int at r);
let f = alloc(float at r);
let t = alloc(int at r1);
```

The type system forbids using an object whose region not in scope (ie. has been deallocated)

```
store 5 in t;    // ok
freeregion(r1);
store 3 in t;       // error region r1 is not available
```

Wang et al [7] introduced the idea of using regions to implement a type-safe garbage collector. Upon garbage collection, the collector creates a fresh alloca-

tion region and performs a deep copy of all the data in the old region reachable from a set of root nodes. Once this is done, all the data reachable from the roots should live in the to-space. The collector can safely deallocate the from-space and resume with the to-space as the new from-space.

3   Advanced types

We now describe the promised tools for the encoding and static checking of invariants in a program.

3.1   Polymorphism

When creating data structures, oftentimes you find that you need that to contain a different type. In C++, we often see usage of the standard template library to generate the version required for a job. Polymorphism allows us to create types that can contain any other type. A common structure that often requires polymorphism is an Array. We would like to define the array type so that it can be used for ints, floats, strings, etc. In Clay we could define such a type as follows.

```
Array polymorphic over type T
Array[type T]
Function polymorphic over type T
T get_i[type T](Array[T] arr, int index)
```

With this we could easily use Array[int], Array[float], etc.

```
Array[int] arr;
get_i[T=int](arr, index);²
```

---

²This explicit declaration of T=int oftentimes can be inferred by the type-checker and

3.2 Type Arithmetic/Constraints via Dependent Types

Dependent types allow for types to be indexed by terms. This is useful for more precisely expressing a type. For example, the length of lists can be expressed by List[length], where *length* is the index object specifying that the length of List is exactly the value *length*. Dependent types allow the programmer to define what invariants she expects to hold, and can detect violations of these invariants during type checking, rather than run time. Following the approach of Xi et al[8], in Clay, Int[I] is a singleton integer representing a one-word runtime representation of the integer type I. For example, Int[7] refers to a run-time word with exactly the value 7. This actually becomes useful when we introduce polmorphism over integers. ie. Int[4*I] refers to an integer that is a multiple of 4. In order to check that a value fits this constraint, we must introduce constraint checking into the type system.

The type system needs to deal with a subset of integer arithmetic that can be solved easily by a standard constraint solver [3]: addition, subtraction, comparison, and multiplication by constants. This avoids having to implement full blown dependent types, which is in undecidable and unpractical. This example in Clay shows use of dependent types to implement bounds check during dependent type checking in contrast to Java's run time bounds check.

```
// With dependent types:
// Polymorphic Array parameterized with length: Array[type T, int Length]
// Singleton value representing the value I:  Int[I]
// Invariant index<length
T get_i[type T, int L, int I; 0<=I && I<L]
    (Array[T,L] array, Int[I] index)
```

omitted.

8

Dependent types may also be used to create data structures that obey a set of constraints. This example in Clay shows a simple data structure supplemented with dependent types. NonNullPtr is polymorphic type that is an integer that cannot be 0. This constraint is introduced with the "exists" construct which packages type variables and constraints.

```
typedef NonNullPtr =
    exists[int Ptr; Ptr!=0] Int[Ptr]
```

In this example, we use exists to package the length of an array in a tuple of (length, array). Linear tuples are represented by @[t1,t2,t3], and nonlinear by .[t1,t2,t3]

```
exists[int L] @[Int[L], Array[A,L]]
```

3.3   Linear Memory

Linear memory types in this paper are based on a simplified version of alias types[5]. Linear memory types map integer word addresses to the types of individual memory words. A linear memory type that contains type τ at word location 400 is expressed by the *size zero* linear type Has[400,τ] . Using each memory word as an individual linear value, we can construct types with multiple memory words. For example, @[Has[400,int], Has[401,int]] describes a pair of integers at word locations 400 and 401. Since Has is a linear type, it is consumed upon its first usage. To use the type, store and load are defined accordingly:

9

```
@[Has[I,A],A] load[int I, type A]
                    (Int[4*I] ptr, Has[I,A] state);
Has[I,A2]      store[int I, type A1, type A2]
                    (Int[4*I] ptr, A2 data,  Has[I,A1] state);
```

Consistent with linear type usage, calling the load or store function consumes the linear type and produces a new linear type.

```
//Load returns a Has type stating that the location still contains
//the same type, and the value of the word at the location.
Has[50,Int[7]] x1 = ...;
let (x1, y1) = load(50*4, x1);  // x1: Has[50,Int[7]]
                                // y1: Int[7]
//Store returns a Has type stating that the location now contains
//the new type stored into the location.
let x2 = store(50*4, 3, x1);    // x2: Has[50,Int[3]]
                                // x1: not in scope
//x1 as a linear type is consumed.  The new Has contains the up to date
//state of the memory word at the location.
```

Note that the type of τ in Has[50,τ] is nonlinear as it may be discarded or copied.

```
let (x2,y1) = load(50*4, x2);
let y2 = y1; // duplication of y1 nonlinear value
```

10

3.3.1 Lists and Stacks

Simple data types such as lists and stacks can be created using linear memory. We will use types of the form If[B,τ], which contains τ if B is true.

```
If[B,τ]
    B == true => τ
```

The following type defines a singly linked list terminated by 0 containing items of type τ. *I*, the address of the cell in the list can be null (0) or some valid address. If *I* is 0, the list terminates with an empty tuple. Otherwise, it contains two *Has* linear memory words, one containing the cell data of type τ and the other containing the address of the next cell in the list. This data-type is recursively defined for for the remainder of the list *List[Next]*.

```
List[int I] =
  If[I!=0,
         exists[int Next]
             @[Has[I,Int[Next]], Has[I+1,τ], List[Next]]]
```

Similarly we can define a stack type in a contiguous sequence of memory locations I1 .. I2.

```
Stack[int I1, int I2] =
  If[I1!=I2,
       @[Has[I1,τ], Stack[I1+1,I2]]]
```

More complex data-types such as trees and free lists can also be built with linear memory and simple integer arithmetic.

3.4 Coercion Functions

In the previous section, the stack type illustrates a problem with recursively defined data types. It is not trivial to conceive how one could implement constant-time random access into the middle of the stack. The program must run a loop for n iterations. In fact, this operation should not impose any run time cost. We describe coercion functions as a method for combining coercion operations into a single constant time operation. A coercion function must satisfy the following constraints:

1. It has no effect on memory (no stores to memory)

2. Returns no values of size greater than 0

3. It is guaranteed to terminate by evaluating to a value in a finite time.

Here is such a function for constant access into the middle of a stack:[3]

```
@[Stack[I1,Ii], Has[Ii,τ], Stack[Ii+1,I2]]
        stack_i[int I1, int I2, int Ii; I1<=Ii && Ii<I2]
                (Stack[I1,I2] stack, Int[Ii] index) limited[Ii-I1] {
    // unpack the stack
    let (has, next_stack) = if_true(stack);
    if[I1==Ii] {
        // base case.  if we are at the right location, return the has.
        return @(if_make_false(), has, next_stack);
    } else {
        // recursive case, run function on smaller stack.
        let (left_stack, has_i, right_stack) = stack_i(next_stack,index);
```

---

[3]Intensionally written as pseudo clay code

12

```
        // package the left_stack

        let left_stack = if_make_true(@(has, left_stack));

        return @(left_stack, has_i, right_stack);

    }

}
```

Since the stack type is of size 0, and the *Has* linear memory word is simply a size zero fact stating that the memory word should contain a certain type, conditions 1 & 2 are easily met. If we could guarantee that the function evaluates to a value in a finite amount of time (terminates), then we could have the compiler eliminate this call at run-time. We will do so by annotating coercion functions with a nonnegative integer limit. The type system only allows coercion functions to call functions with lower limits than their own limit. We see that if we annotate the stack_i function with "limit *Ii-Il*", we can show that this function will terminate as the recursive call is to a function of limit *Ii-Il-1*.

Viewed from another perspective, with the Curry-Howard isomorphism we argue that our coercive function is no more than a strongly normalizing proof that takes a proof as an argument and returns three proofs as a result.

3.5   Nonlinear Type Sequences

Although linear types can be used to implement some common data structures efficiently, linear data structures are too difficult to use for other applications. Compiler generated proof-carrying code and typed assembly language must support nonlinear data types at a low level in order to support nonlinear types in the high level language. In our case, a garbage collector must be able

to support the nonlinear data types of the language being garbage collected. We introduce type sequences to support nonlinear types.

A type sequence is a mapping from integers to types. The linear memory type, which is a mapping from integer memory locations to types, can be modified to use a type sequence F that maps integers to types:

Has[I$_1$, F[I$_2$]]

The type sequence F is controlled by a size zero linear generator of type Fun-Gen[F,I]. This sequence is growable using the expression define_fun(FunGen[F,I], τ). define_fun(FunGen[F,I], τ) grows the sequence by 1 and adds the type τ to the sequence at index I. It does this by consuming the old generator and returning three size zero values:

- a new generator of type FunGen[F,I+1]

- a nonlinear proof (Eq_T[F[I],τ]) that F[I] = τ

- a nonlinear proof (InDomain[F,I]) that I will always be in the domain of F (since sequences grow but don't shrink)

```
@type0 FunGen[SEQ F, int I]; // linear function generator
// creates a new function generator with no types in sequence
exists[SEQ F] FunGen[F,0] new_fun();
@[FunGen[F,I+1], Eq_T[F[I],τ], InDomain[F,I]]
        define_fun[SEQ F, int I, type T](FunGen[F,I] fungen);
```

For example defining F = [Int[111], Int[333], Int[222]]:

```
let fungen = new_fun();

let (fungen,eq_1,ind1) = define_fun[T=Int[111]](fungen);

let (fungen,eq_2,ind2) = define_fun[T=Int[333]](fungen);

let (fungen,eq_3,ind3) = define_fun[T=Int[222]](fungen);
```

Nonlinear types eq_1,eq_2,eq_3 may be used as the basis of a nonlinear data structure:

```
NonlinearData = struct {
    Eq[F[0],Int[111]] eq_1;
    Eq[F[1],Int[333]] eq_2;
    Eq[F[2],Int[222]] eq_3;
}
```

We can reflect this data structure on the linear side:

```
@(Has[500,F[0]], Has[501,F[1]], Has[502, F[2]])
```

We can write this more generally as:

$\forall 0 \leq I < 3.$`Has[500+I,F[I]]`

In Clay, we have defined a Linear array type LArray[int StartI, int EndI, (int)->type], which is really a just a more polymorphic version of the stack type. (int)->type is a type function that takes an int as a parameter and returns a type. In Clay we show this using the construct fun[int I] T, where T is some type.

```
//The above example in an array
LArray[500,503, G]
// where G[I] = Has[I,F[I]]
// One step further
LArray[0,3, fun[int I] Has[500+I,F[I]]]
```

Now we have the basis for creating regions.

```
Region[SEQ F, int Alloc] = struct {
    FunGen[F,Alloc] fungen;
    LArray[0, Alloc, fun[int I] Has[I, F[I]]] facts;
}
```

By obtaining a *Has[I,F[I]]* from the array of facts, we may load the value with type F[I]. The type of F[I] is recovered through the equality F[I] $= \tau$. By substitution of type F[I] withτ, we may load a value at location I of type τ from the region.

3.6   Garbage Collection

Garbage collection is the automatic reclamation of computer storage. In many systems, programmers have to explicitly reclaim heap memory by using statements such as "free" and "delete". Manual memory management can be error prone. Garbage collection systems free programmers from this burden.

Mark-sweep garbage collection is defined by two phases, namely mark, then sweep.

1. Mark phase: Distinguish the live objects from the garbage. This is done by starting from some root set of nodes, pointers into the heap, and traversing

16

the graph structure formed by inter-object pointers. This is usually done by a depth-first or breadth-first traversal. Objects that are reached are marked as live. Typically this marking is done by setting a header bit in the object, or by recording in a bitmap or table. At the end of the mark phase, any objects left unmarked are garbage as they cannot be reached in any future execution of the program.

2. Sweep phase: Reclaiming the garbage. Once the live objects have been distinguished from the garbage, memory is swept to find all the unmarked objects (garbage) in order to reclaim their space.

4   Type-Preserving Garbage Collection

This section describes a type-preserving mark-sweep garbage collector.

This collector improves on previous type-preserving copying collectors:[2, 7]

- The nature of the mark-sweep algorithm allows for objects to be garbage collected in place. This is in contrast to the two space region copying garbage collectors described previously.

- Potential for extension with additional improvements such as incremental collection. Mark-sweep collectors are easier to make incremental than copying collectors. With the approach of Wang et al, collection could only occur at safe points which needed to be defined.

In order to model the mark phase of the collector, we will borrow Dijkstra's tri-color abstraction [1]. We will divide existing objects into three regions represented by the colors white, gray, and black. Before the mark phase of collection, all objects will start as white. At the end of the mark phase, live objects

are black and garbage objects are white. In order to implement the depth first search, root pointers will be followed to objects in the heap of the white color. These objects will be then marked as gray. The gray objects form a list structure. The head of this list is popped off and pointers examined. The objects these pointers refer to are pushed onto the head of the gray list. The original popped head of the gray list is then colored black. While marking, each object will progress from white to gray to black. This step is repeated until the gray list is empty, in which case all live objects will have been marked. When the mark phase is completed, the sweep phase will scan through the heap and reclaim all white objects that remain as garbage. At the finish of the sweep phase, the black objects will become white objects ready for the next collection. Here is pseudocode of the garbage collection loop, assuming every object contains two pointers (pointer1 and pointer2).

```
make_gray(root_object_pointer)4
enqueue(gray_list,root_object)
while (gray_list.is_not_empty()) {
    let obj = dequeue(gray_list)
    make_gray(obj.pointer1)
    enqueue(gray_list,obj.pointer1)
    make_gray(obj.pointer2)
    enqueue(gray_list,obj.pointer2)
    make_black(obj)
}
```

We represent the three color abstraction of the mark phase by the region number R. Upon the first collection, all objects are *white* with the region number 1.

---

[4]Pseudocode for garbage collection loop.

RMin refers to the region number that currently represents *white*. In this first collection, *gray* objects have the region number 2, and *black* objects, 3. When garbage collection is finished, and the *black* objects must be converted to *white*, RMin is incremented by 2. *White* objects have region number RMin+0, *gray* objects RMin+1, and *black* objects RMin+2.

```
        region0    region1    region2
        (RMin=1)   (RMin=3)   (RMin=5)
  R=1    white
  R=2    gray
  R=3    black       white
  R=4                gray
  R=5                black       white
  ...
```

4.1   Nonlinear Representation

Using the linear memory/nonlinear representation model described earlier, we will now describe the nonlinear representation of the garbage collection space.

4.1.1   Abstract Invariants

We use a two dimensional type sequence M that carries information about each memory location's color and types as a sequence over time. $M[V]$ denotes a row containing all memory objects' color and types. $M[V][i]$ refers to object i at time V. $M[V][i] = R{:}T$ where R is the number whose value refers to the current region number of the memory word, and within context of a specific region (ie. region0, RMin=1), the color *white*, *gray*, or *black*. T is the type of the Object.

```
m[v][i] = R:T
i = memory word
v = version
*      ---> mem object
*        0     1     2      3     4 ...
*   m[0][ 0  ][ 0  ][ 0   ][ 0  ][ 0   ]
* | m[1][ 0  ][ 0  ][ 1:T ][ 0  ][ 0   ]
* | m[2][ 0  ][ 0  ][ 1:T ][ 0  ][ 1:T2']
* V m[3][ 0  ][ 0  ][ 2:T'][ 0  ][ 1:T2']
*    .
* t  .
* i  .
* m  .
* e  .
*   m[v][     ][     ][     ][     ][      ]
```

Using this type sequence of type sequences M array, we can make assertions on
the types of objects through the mark phase:

Type-Preserving: Live objects should retain the same type
before and after garbage collection.

An object should additionally retain the same type in between collection cycles.

```
[RMin]  m[V][i] = R:T
[RMin'] m[V'][i] = R':T'
```

$$R \geq RMin' \Rightarrow T = T'$$

We motivate this invariant by looking at three cases of it:

- R<RMin': Object is not live at time V'. When objects aren't live, the
  memory words they occupy reside in a freelist. These words may be allo-
  cated for use in the program. At this point, the type may change to reflect
  the type of the new allocated object.

20

```
                       0      1      2      3      4 ...
(RMin=1)  m[2][ 0   ][ 0   ][ 0:T ][ 0   ][ 0   ]
(RMin=1)  m[3][ 0   ][ 0   ][ 1:T'][ 0   ][ 0   ]
(RMin=1)  m[4][ 0   ][ 0   ][ 2:T'][ 0   ][ 0   ]
```

- *R>=RMin' and RMin>=RMin':* With two rows belonging in the same collection cycle, any valid object either white, gray or black (RMin+0, RMin+1, or RMin+2) must be greater than equal to RMin. Thus the type of that object is forced to remain the same within the same collection cycle.

```
                       0      1      2      3      4 ...
(RMin=1)  m[2][ 0   ][ 0   ][ 2:T ][ 0   ][ 2:T2]
(RMin=1)  m[3][ 0   ][ 0   ][ 3:T ][ 0   ][ 2:T2]
Here valid objects at 2 and 4 are both gray in row 2.
Objects at row 3 become black and gray respectively.
Types are preserved with this step of the collection.
```

- R>=RMin' and RMin'>RMin: In between collection cycles, RMin becomes RMin'. If an object was valid in the old region RMin and if the object is still valid in the new region RMin', the type of the object will remain the same.

```
                       0      1      2      3      4 ...
(RMin=1)  m[2][ 0   ][ 0   ][ 3:T ][ 0   ][ 2:T2 ]
(RMin=3)  m[3][ 0   ][ 0   ][ 3:T ][ 0   ][ 2:T2']
Here valid objects 2 and 4 are black and white respectively in row 2.
With the sweep phase, all objects still white are garbage collected,
and all objects once black are now white as RMin'=RMin+2.
Between collection cycles, T type is preserved.
```

In short, as long as an object is live, it will retain the same type.

Progression: Objects can only travel from white to gray to black.

We maintain in the M array that the region number is nondecreasing. That is for:

```
m[V][i] = R:T
m[V'][i] = R':T
```

$$V' \geq V => R' \geq R$$

Non discarding: Live objects should not be discarded as garbage.

Constraints are placed upon the colors of pointers in an object. Pointers from white objects may point to white, gray or black objects. Pointers from gray objects may only point to white, gray or black objects. Pointers from black objects may only point to gray or black objects. Upon sweep phase, black objects may only point to black objects.

white   gray   black
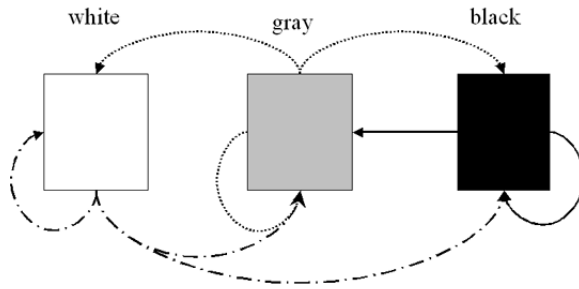
With this, any live object graph structure must be marked, and upon completion of collection, this structure must reside completely in the black region to avoid loss of live objects.

4.1.2 Nonlinear Datatypes

A nonlinear heap object consists of information contained in M and colors of the internal pointers.

GcHas[V,I,RegionColor,Ptr1,Ptr2] contains the relation from the two dimensional array: M[V][I] = RegionColor:Type, where Type is Ptr1,Ptr2. Using this equality of the cell M[V][I] and the MArray, we can assert the invariants mentioned above.

```
// information about M array
GcHas[int V, int I, int Color,
     int Ptr1, int Ptr2] =
   M[V][I]=Color:Ptr1,Ptr2⁵
```

NObject[V,I,RegionColor,ObjPtr1,ObjPtr2] is the nonlinear version of a heap object. RegionColor refers to the current region color of the object: white,

---

[5]This is pseudocode notation for Clay's construct Eq_T for the proof that M[V][I] = τ .

gray, black, or non-live (RegionColor<RMin). ObjPtr1 and ObjPtr2 refer to
the addresses of the two object internal pointers that allow the object to point
to other objects in order to create the live graph structure of the heap. As we
will see later, a heap object is actually four words containing: region number,
gray list pointer, internal pointer #1, and internal pointer #2. Three of these
words are described in the nonlinear portion of the object.

```
// heap object
NObject[int V, int I, int RegionColor,
        int ObjPtr1, int ObjPtr2] =
   If[I>0,
     GcHas[V,I,RegionColor,
           ObjPtr1,ObjPtr2],
     NPtr[V, ObjPtr1, RegionColor-1],
     NPtr[V, ObjPtr2, RegionColor-1]]
```

NPtr[V,Ptr,HadColor] is a statement about the color of the object of an internal
pointer stating that the object at least has color Region-1. For white objects at
RMin+0, this generality requires some extra work to show that no objects exist
at RMin-1. White objects can point to objects with colors greater than equal
to RMin, white, gray, and black. Gray objects at RMin+1 can point to objects
with colors greater than equal to RMin, white, gray, and black. Black objects
at RMin+2 can point to objects greater than equal to RMin+1, gray and black.

```
// object internal pointer
NPtr[int V, int Ptr, int Color] =
 exists[int V0; V0<=V]
    GcHadColor[V0,Ptr,Color]
// object at least with color HadColor
```

```
GcHadColor[int V, int I, int HadColor]
 exists[int Color; Color>=HadColor]
   M[V][I] = Color
```

It is important to note that the nonlinear representation of objects and the type sequence M are simply type checking time constructs that take no run time space or time.

4.2   Linear Representation

We maintain an array of linear objects.

In our linear representation, the heap object consists of two header words and the internal pointers. The first word with R is the region color of the object. The second word with *GrayPtr* is the pointer to the next gray object in a singly linked list of gray objects. The third and fourth words are the data words of the object. In our case, we have described these two words on the nonlinear side as the object's internal pointers. *GrayColor[GrayPtr,R]* places the constraint that if the current object is gray, then its gray pointer must be either null or valid and once have pointed to a gray object. In reality, the gray list could have black elements and this would indicate a bug in the collector. We will discuss how we solve this with a run-time check.

```
LinearObject[int V, int I, int R, int GrayPtr,
      type T1, type T2, int RMin] =
struct {
  M[V][I] = R: T1,T2;
  GrayColor[V,GrayPtr, R] graycolor;
```

```
    @[Has[I+0, Int[R]],
      LIf[R>=RMin,
         Has[I+1, Int[GrayPtr]],
         Has[I+2, T1],
         Has[I+3, T2]]] has;
  }
  GrayColor[int V, int GrayPtr, int R] =
    exists[int V0, int GrayR;
         V0<=V &&
         (R!=RMin+GRAY_COLOR || GrayR==RMin+GRAY_COLOR)]
      If[GrayPtr>0, M[V0][GrayPtr] = GrayR:_,_ ]
```

4.3 Garbage Collection Space

In order to describe in detail the changes in types as we garbage collect, we will describe a simplified version of the garbage collection data structure.

```
    GcSpace[int V, int RMin, int S, int GrayPtr] =
    struct {
      MArray[TSEQ,RSEQ,V,S,RMin] marray;
      LArray[...,
         fun[int I]
            exists[int R, int GrayPtr, type T1, type T2]
               LinearObject[V,I,R,GrayPtr,T1,T2,RMin] linear_memory;

      // array of ∃Region,Ptr1,Ptr2.NObjects
      LArray[...,
```

```
      fun[int I]

        exists[int R, int Ptr1, int Ptr2]

            NObject[V,I,R,Ptr1,Ptr2] object_list;

    // head of the gray list

    exists[type T1, type T2]

        NObject[V,GrayPtr,RMin+COLOR_GRAY,Ptr1,Ptr2] gray_object;



    FreeList[...] freelist;

    FreeStore[...] freestore;

  }
```

MArray[RSEQ,TSEQ,S,V,RMin] is a datatype containing the abstractions mentioned in the previous section. RSEQ and TSEQ refer to the type sequence two-dimensional arrays for the region color and the types respectively (for technical reasons, these are maintained separately). S is the number of gray objects in the current row. For sake of brevity, this datatype is not included in this paper.

Garbage collection starts with *GcSpace[M_ V,RMin,0,0]*. Of note, S=0 and GrayPtr=0, which means that there are 0 gray objects and the gray_list is empty. All objects are currently colored white (RMin). There are no gray objects at this time, and the *gray_ object* list is empty.

Given a root object NObject[M_V,RootPtr,RMin+GRAY_COLOR,...] root, we add this object to the gray list. Now the GcSpace is *GcSpace[M_ V,RMin,1,RootPtr]* where there is one gray object. Now proceeds the mark phase.

27
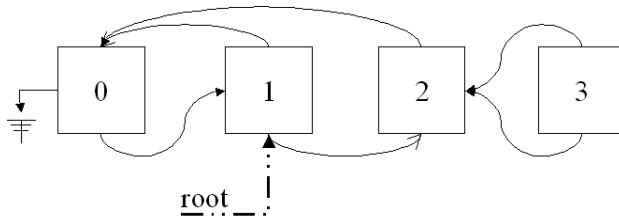
* these are indices 0, 1, 2, 3 into object array.
actual addresses would be Base+OBJECT_SIZE*Index

Mark Phase:

Suppose we have the case in the figure above. RootPtr is Base+OBJECT_SIZE*1, or the object at index 1 in the figure. Currently the gray list consists only of the root pointer at index 1.

1. The Collector removes the head from the gray list. This involves a load of the gray pointer linear memory word *Has[RootPtr+1, GrayPtr]*, followed by extracting the NObject at index 1 from the ObjectArray *object_list*. After the dequeue operation, the gray list is null.
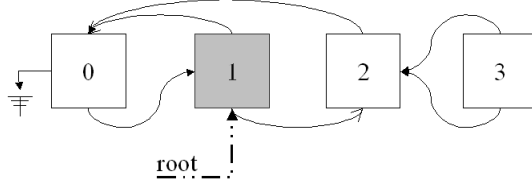

2. Next, the internal pointers of object 1 are examined. The first internal pointer is to the object at index 0. This object is made gray. When an object's color changes, this must be reflected in M. The latest row in MArray is duplicated, with the exception of the cell which changed. Any additional row to the M array must also obey the abstract invariants we discussed in 4.1.1. ie. Region number is nondecreasing.

```
                   0      1      2      3      4 ...
(RMin=1)  m[2][ 1W ][ 2G ][ 1W  ][ 1W ][ 0  ]
(RMin=1)  m[3][ 2G ][ 2G ][ 1W  ][ 1W ][ 0  ]
```

The gray list now consists of object at indice 0. GcSpace is now *Gc-Space[M_V,RMin,2,ObjPtr1]* where ObjPtr1 is a pointer to object at index 0. We repeat the same with object at index 2. GcSpace is now *GcSpace[M_V,RMin,3,ObjPtr2]* where ObjPtr2 is a pointer to object at index 2. The gray list is now 2, 0, null.
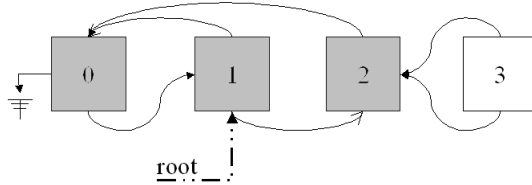
3. The object 1 popped off the head of the gray list is now turned black. Note that one of the condition of a black object is that its children must be at least color gray. This case is fulfilled
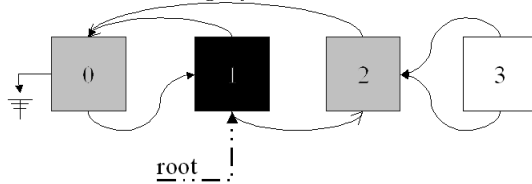
4. Steps #1,2,3 are repeated with object at indice 2 as the new head of the gray list. This cycle is repeated until the gray list is empty, only white and black objects remain in the list.

These steps might seem very standard to a mark-sweep collection, except upon closer examination of the types involved in conjunction with the M array.

After collection, we see that a black object 01 pointed at one time to a gray object 02, but no gray objects are left. Since we maintain a count "S" of

29

gray objects, and no gray objects are left, we can roll forward in the M array to conclude that 02 is black, not collected, and 01->02 is not dangling. The "rolling forward in the M array" operation is actually a simple index into the M array cell. Each M array cell has the constraint that if S == 0, RegionColor != GRAY_COLOR. So once we index into the array, we can be assured that the cell's current color is at least black.

```
// Cell M[V][I] of the M Array
typedef XMArrayCell[SEQ M, int V, int I, int S, int RMin] =
    exists[int RegionColor, type T1, type T2; R<=RMin+2
        && (S>0 || R!=RMin+GRAY_COLOR)
        MArrayCell[M,V,I,RegionColor,T1,T2]
// heap object
NObject[int M_V, int This, int RegionColor,
        int ObjPtr1, int ObjPtr2] =
    If[This>0,
        GcHas[M_V,This,RegionColor,
            ObjPtr1,ObjPtr2],
        NPtr[M_V, ObjPtr1, RegionColor-1],
        NPtr[M_V, ObjPtr2, RegionColor-1]]
```

An object contains facts about itself and the color the internal pointer objects once had. A gray object will always claim that its internal pointer objects were once white - a claim that will always be true. Recall that while linear facts may be invalidated, nonlinear facts persist forever.

4.4 Features, Optimizations, Inefficiencies

One additional implementation detail glossed over was the treatment of the gray list. Recall that the linear type representation of the heap object only maintains that if the current object is gray, then the gray pointer points to an object that once was gray. This could allow an incorrect implementation of the mark-sweep garbage collector to use this datatype and incorrectly create a gray list with black objects during the mark phase. This would be a bug. Fortunately, the type system exposes this problem and forces us to solve it with a runtime check. We see that make_black (in the gc loop pseudocode) must take a gray object as a parameter. Since the best information we have on the color of the object is that it is at least gray, the only way to know that we in fact have a gray object, and not a black object is to make a run time check.

Another difficulty with the gray list resided in the type variable "S". "S" should reflect the number of gray objects currently in the object list. However, this is not connected to the proper construction of a gray list. If there is a bug in the collector, an incorrect implementation could have gray objects (S>0) and an empty gray list. Once again, the type system requires that we deal with this at runtime if this bug occurs.

These run-time checks only slow the collector. They do not slow the program's loads/stores to the object data fields. This is evident in the pseudocode below, that once a constraint has been introduced into the environment (result of being inside if/else branch after run-time check), static checking of the maintanance of that constraint is possible. Lacking a better scheme for encoding the number of gray objects and the length of the gray list, we accept this minor overhead in runtime checks.

```
// code to handle incorrect cases.
```

```
    if (S>0) {

      if (Gray_list == null) abort("Bug! Gray List should not be null");

    } else {

      let object = hd(graylist);

      if (object.color != gray) {

        abort("Bug! Gray List should only contain gray objects");

      } else {

        //make_black_object has as constraint that object must be currently gray

        make_black(object);

      }

    }
```

5   Conclusions and Future Work

This paper demonstrated that linear types with support for simple arithmetic types, coercion functions and type sequences can give type-safe code more control over low-level memory management. Using these we constructed a type-preserving garbage collector with many advancements over existing type-preserving garbage collectors. With static type-checking techniques and little added runtime overhead, it is hoped that this garbage collector can be comparable in performance to existing collectors. Performance evaluations could help to fine tune our methods. Our mark-sweep implementation has the potential for extension. Incremental collecting would be more feasible with a mark-sweep collector than previous type-preserving copying collectors. It should also be feasible to create a general-purpose collector with arbitrary type heap objects. Hopefully in a few years time, we can make use of some of our ideas in the creation of an efficient practical garbage collector.

References

[1] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 1978.

[2] Stefan Monnier, Bratin Saha, and Zhong Shao. Principled scavenging. In *Proceedings of the ACM SIGPLAN'01 conference on Programming language design and implementation*, pages 81–91. ACM Press, 2001.

[3] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13. ACM Press, 1991.

[4] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, 1990.

[5] David Walker and Greg Morrisett. Alias types for recursive data structures. *Lecture Notes in Computer Science*, 2071, 2001.

[6] David Walker and Kevin Watkins. On regions and linear types (extended abstract). In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 181–192. ACM Press, 2001.

[7] Daniel C. Wang and Andrew W. Appel. Type-preserving garbage collectors. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 166–178. ACM Press, 2001.

[8] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN '98 conference on Programming language design and implementation*, pages 249–257. ACM Press, 1998.