Implementation and Performance Evaluation of a Safe Runtime System in Cyclone

Matthew Fluet
Cornell University
Department of Computer Science
4133 Upson Hall
Ithaca, NY 14853
fluet@cs.cornell.edu

Daniel Wang*
Princeton University
Department of Computer Science
35 Olden Street
Princeton, NJ 08544
danwang@cs.princeton.edu

ABSTRACT

In this paper we outline the implementation of a simple Scheme interpreter and a copying garbage collector that manages the memory allocated by the interpreter. The entire system including the garbage collector is implemented in Cyclone [11], a safe dialect of C, which supports safe and explicit memory management. We describe the high-level design of the system, report preliminary benchmarks, and compare our approach to other Scheme systems. Our preliminary benchmarks demonstrate that one can build a system with reasonable performance when compared to other approaches that guarantee safety. More importantly we can significantly reduce the amount of unsafe code needed to implement the system. Our benchmarks also identify some key algorithmic bottlenecks related to our approach that we hope to address in the future.

Although most of the theoretical ideas in this work are not by themselves novel, there are a few interesting variations and combinations of the existing literature we will discuss. However, the primary motivation and goal is to build a realistic working system that puts all the existing theory to test against existing systems which rely on larger amounts of unsafe code in their implementation.

1. INTRODUCTION

Network servers written in unsafe languages, such as C, are a significant source of known security exploits [19]. Fortunately, many new web based applications are written in high-level, safe languages such as C#, Java, Perl, PHP, Python, or Tcl. Web based applications written in safe languages are immune to common buffer overflow and other related security problems found in applications written in C. In addition to ensuring safety, these high-level languages are more convenient and preferable to low-level system languages for building web based applications. One major convenience, that also improves security, is automatic memory management.

Although the majority of web based applications are written in safe languages, the applications are typically hosted on servers or web application platforms that are implemented with unsafe languages such as C. For example, Perl based web applications are hosted on the Apache web server by dynamically loading a native code module that implements the Perl interpreter. Since the interpreter is written in C, we should be concerned that a bug in the module may introduce a security hole. In fact since Apache itself is written in C, we must be concerned about its immunity to buffer overruns.

Some web application platforms such as Jetty [10] are written completely in Java. Even in this situation, we must trust the implementation of the Java Virtual Machine to be free from bugs. The implementation of a JVM contains a significant amount of code written in unsafe languages.

Reducing or eliminating code written in unsafe languages from the implementation of a web application server increases our confidence that the server is immune to common security bugs. There already has been a great deal of work demonstrating how to build dynamically extensible web servers in safe programming languages [8]. What has not been addressed is how to implement an interpreter that executes the actual web application itself.

Implementing the core of an interpreter in a safe language is not in itself a significant challenge. What is challenging is implementing the runtime system for the interpreter that provides automatic memory management. Currently, all systems rely on a trusted runtime system to provide some sort of storage management. Significant advances in type and proof systems have made it possible to implement a runtime system that provides garbage-collection services using programming languages that guarantee safety.

In this paper we outline the implementation of a simple Scheme interpreter and a copying garbage collector that manages the memory allocated by the interpreter. The entire system including the garbage collector is implemented in Cyclone [11], a safe dialect of C, which supports the safe and explicit management of memory. In Section 2 we motivate and describe the high-level design of the system. In

*This research was supported in part by ARDA contract NBCHC030106; this work does not necessarily reflect the opinion or policy of the federal government and no official endorsement should be inferred.

SPACE2004 2004 Venice, Italy

Section 3 we report some preliminary benchmarks and compare our approach to other Scheme systems. In Section 4 we compare the amount of unsafe code needed to implement our system. Our preliminary benchmarks demonstrate that one can build a system with reasonable performance when compared to other approaches that guarantee safety. More importantly we can significantly reduce the amount of unsafe code needed to implement the system. Our benchmarks also identify some key algorithmic bottlenecks related to our approach that we hope to address in the future in order to build a realistic production system.

2. WRITING A SCHEME INTERPRETER IN CYCLONE

2.1 Why Scheme

We have chosen Scheme as our test bed language, primarily because of the relative ease of implementation[1] and the existence of well known benchmarks. Our implementation is small, because we rely on an existing Scheme front-end [4] to expand the majority of the full Scheme language into a core Scheme subset. The existing front-end handles parsing and macro expansion of full Scheme. We take the resulting output and emit Cyclone source code that builds an abstract syntax tree, which we compile and link with our interpreter and runtime system.

Not only is Scheme a well studied and compact language, but it also has many features that make it desirable for web programming. In particular first class continuations allows for a much more natural structuring of web applications [17]. Scheme seems to be developing a large and useful set of tools for manipulating XML. Given that IEEE Scheme was the original processing language for SGML from which XML and HTML are derived, we see a great future for Scheme as a web programming language.

2.2 Key Features of Cyclone

Cyclone [5] is a safe dialect of C. Cyclone attempts to give programmers significant control over data representation, memory management, and performance (like C), while preventing buffer overflows, format-string attacks, and dangling-pointer dereferences (unlike C). Cyclone ensures the safety of programs through a combination of compile-time and run-time checks. Cyclone's combination of performance, control, and safety make it a good language for writing low-level software, like runtime systems.

In this section, we briefly introduce the key features of Cyclone that are used in the implementation of our Scheme interpreter described in Sections 2.3 and 2.4.

Pointers

As with C programs, Cyclone programs make extensive use of pointers. However, improper use of pointers can lead to core dumps and buffer overflows. To prevent these errors, Cyclone introduces different kinds of pointers, which make various tradeoffs between expressiveness and run-time checks.

The type of a (traditional) nullable pointer is written `t*`, as in C. For the most part, such pointers have the same behavior as their counterparts in C. However, there are some restrictions on the use of nullable pointers in Cyclone. First, one cannot cast an integer to a pointer. (Cyclone accepts `0` as a legal pointer value, but using `NULL` (a Cyclone keyword) is preferred.) Second, one cannot perform arbitrary pointer arithmetic on a `*` pointer, which would also allow the programmer to overwrite arbitrary memory locations. Finally, Cyclone inserts a null check whenever the program dereferences a `*` pointer. While these may appear to be drastic restrictions, there are many patterns of pointer usage that are unaffected by them. For example, in our interpreter, Scheme values are naturally represented by nullable pointers to data-structures, where `NULL` corresponds to the Scheme value `nil`.

A variation on a nullable pointer is a non-null pointer, written `t*@notnull`. Such a pointer is guaranteed to not be `NULL`, which eliminates the need for null checks at dereferences. In addition to this performance benefit, non-null pointers capture a useful programming invariant which can be statically checked. For example, in our interpreter, the abstract machine state is represented as a non-null pointer to a data-structure, which is side-effected by each transition of the abstract machine.

A third kind of pointer available in Cyclone is a fat pointer, written `t*@fat`. A fat pointer comes with bounds information (thus, is "fatter" than a traditional pointer). Each time a fat pointer is dereferenced or its contents are assigned to, Cyclone inserts both a null check and a bounds check.[2] The bounds information and these run-time checks ensure the safety of pointer arithmetic and array indexing. In addition, a fat pointer allows its size to be queried at run time. In our interpreter, a Scheme vector is represented by a fat pointer to an array of values.

Regions

In the description of pointers above, we have demonstrated the ways in which Cyclone prevents programs from accessing arbitrary memory. Another violation is to dereference a *dangling pointer*: a pointer to storage that has been deallocated. To prevent such violations, Cyclone adopts a type system for region-based memory management, based on the work of Tofte and Talpin [18], and described in detail in previous work [6]. Here, we discuss only the aspects most crucial to our implementation.

Cyclone pointer types have the form `t*‘r`, where `t` is the type of the pointed to object and `‘r` is a *region name* describing the object's lifetime. The Cyclone type system tracks the set of live regions at each program point; dereferencing a pointer of type `t*‘r` requires that the region `‘r` is in the set. (If `‘r` is not in the set, a compile-time error signals a possible dangling pointer dereference.) Region polymorphism lets functions and data-structures abstract over the region of their arguments and fields. Region parameters in Cyclone are indicated by annotations of the form `<‘r::R>` on functions and types, where `R` distinguishes region parameters from other kinds of parameters.

Cyclone provides a number of different kinds of regions, suitable for different allocation and deallocation patterns. *Stack regions* correspond to local-declaration blocks: entering a block creates a region and allocates objects, while exiting the block deallocates the region's objects. Hence, a stack region has lexical scope, but the number and sizes of

[1] Our core interpreter loop is only ~500 lines of code.

[2] In many cases, Cyclone's flow analysis determines that the checks are superfluous and removes them.

stack allocated objects is fixed at compile time. *Lexical regions* are also created and deallocated according to program scoping, but a *region handle* allows objects to be allocated into the region throughout the region's lifetime. Hence, a lexical region has lexical scope, but the number and sizes of region allocated objects are not fixed at compile time.

The Cyclone heap is a special region with the name `‘H`. All data allocated in the heap is managed by the Boehm-Demers-Weiser (BDW) conservative garbage collector [3]. Conceptually, the Cyclone heap is just a lexical region with global scope, and the global variable `heap_region` is its handle. In order to understand the cost of a safe garbage collector, we use the Cyclone heap to implement a version of our Scheme interpreter in which all interpreter data is managed by the (trusted) conservative garbage collector.

The lifetimes of stack and lexical regions follow the block structure of the program, beginning and ending in a last-in-first-out (LIFO) discipline. Clearly, such a discipline can be too restrictive, as it can not accommodate objects with overlapping, non-nested lifetimes.

Recent work has added unique pointers and dynamic regions to Cyclone [9]. Unique pointers are based on linear type systems and provide fine-grained memory management for individual objects. In particular, a unique pointer's object can be deallocated at any program point. On the other hand, unique pointers cannot be freely copied and there are further restrictions on their use in Cyclone programs. A unique pointer is written `t*‘U`; conceptually, a pointer into a special unique region with the name `‘U`. While there is much more to be said concerning Cyclone's unique pointers, for our purposes it suffices to reiterate that unique pointers are treated as linear objects, where the type system and a conventional flow analysis ensures that, at every program point, there is at most one usable copy of a value assigned a unique-pointer type. We make use of unique pointers both to manage the garbage collector's frontier data-structures and to implement our dynamic-region sequences.

A dynamic region resembles a lexical region in many ways; the crucial difference is that a dynamic region can be created and freed at (almost) any point within a program. However, before accessing or allocating data within a dynamic region, the region must be *opened*. Opening a dynamic region adds the region to the set of live regions and prevents the region from being freed while it is open. The interface for creating and freeing dynamic regions is given by the following:

```
typedef struct DynamicRegion<‘r>*@notnull‘U
     uregion_key_t<‘r::R>;
struct NewDynamicRegion { <‘r::R>
  uregion_key_t<‘r> key;
};
struct NewDynamicRegion new_ukey();
void free_ukey(uregion_key_t<‘r> k);
```

A dynamic region is represented as a unique non-null pointer to an abstract `struct DynamicRegion<‘r>` (which is parameterized by the region `‘r` and internally contains the handle to the region). This unique pointer is called the *key*, which serves as a run-time capability granting access to the region.

The `new_ukey` function creates a fresh dynamic region and returns the unique key for the region. (The `<‘r::R>` annotation in the `struct NewDynamicRegion` type indicates that the region variable is existentially bound. Unpacking this existential type yields a region variable which does not conflict with any other region name. This is precisely the behavior we require for a function that creates a fresh region.) The `free_ukey` function reclaims the key's region and the storage for the key. Since the key is unique, it must be used in a linear manner; the `free_ukey` function consumes the key.

To access or allocate data within a dynamic region, a key is presented to a special `open` primitive: `region r = open(k)`. Within the remainder of the current scope, the region handle `r` can be used to access `k`'s region; furthermore, `k` is temporarily consumed throughout the scope (preventing deallocation) and becomes accessible again when control leaves the scope.

Symbols:
$q \in \mathcal{Q}$
Booleans:
$b \in \mathcal{B} = \{\mathsf{true}, \mathsf{false}\}$
Numbers:
$x \in \mathcal{N} \supseteq \mathbb{C} \supseteq \mathbb{R} \supseteq \mathbb{Q} \supseteq \mathbb{Z}$
Characters:
$c \in \mathcal{C}$
Constants:
$K \in \mathit{Const}$
$K ::= q \mid b \mid x \mid c \mid \mathsf{nil} \mid \mathsf{undefined} \mid \mathsf{unspecified}$
deBruijn Indices:
$i_j \in \mathbb{N} \times \mathbb{N}$
Expressions:
$e \in \mathit{Exp}$
$e ::= K \mid i_j \mid (e\ e_1\ \ldots\ e_n) \mid (\mathbf{lambda}^n\ e) \mid (\mathbf{valambda}^m\ e) \mid (\mathbf{if}\ e_1\ e_2\ e_3) \mid (\mathbf{set!}\ i_j\ e)$

Figure 1: Core Scheme Syntax (Expressions)

2.3 The Core Scheme Interpreter

In this section, we introduce the Core Scheme language, its semantics, and its implementation in Cyclone. While each of these components is relatively straightforward, it is useful to have a concrete implementation in preparation for the description of the garbage collector.

Formalism

The reader familiar with Scheme interpreters may feel free to proceed directly to the next section, referring back to this reference formalism as necessary. Figure 1 describes the syntax of Core Scheme expressions. Literal constants for various types are provided. Variables are given as deBruijn indices. Hence, procedures elide variable names and are instead annotated with the number of arguments required for application. In the case of a procedure with a variable number of arguments, the annotation indicates the minimum number of required arguments. Likewise, an assignment indicates the location to be updated by a deBruijn index. Finally, procedure calls and conditionals are standard.

The Core Scheme expression language corresponds closely to the primitive expression forms that appear in the Scheme standard [12]. As described in the standard, they are suffi-

Locations:

$l \in Loc$

Values:

$v \in Val$

$v ::= (\textbf{const}\ K) \mid (\textbf{prim}\ p) \mid (\textbf{throw}\ S\ \rho) \mid (\textbf{closure}^n\ \rho\ e) \mid (\textbf{vaclosure}^m\ \rho\ e) \mid (\textbf{pair}\ l_1\ l_2) \mid (\textbf{vector}\ l^*)$

Primitives:

$p \in (Heap \times Stack \times Env \times Loc^*) \rightharpoonup State$

Stacks:

$S \in Stack = (Env \times Frame)^*$

$S ::= \cdot \mid \langle \rho, F \rangle :: S$

Frames:

$F \in Frame$

$F ::= (l_1\ \ldots\ l_{i-1}\ \square\ e_{i+1}\ e_n) \mid (\textbf{if}\ \square\ e_2\ e_3) \mid (\textbf{set!}\ i_j\ \square)$

Environments:

$\rho \in Env = Loc^*$

$\rho ::= \cdot \mid l :: \rho$

Results:

$r \in Res$

$r ::= e \mid l$

Heaps:

$H \in Heap = Loc \rightharpoonup Val$

States:

$\sigma \in State = Heap \times Stack \times Env \times Res$

$\sigma ::= \langle H, S, \rho, r \rangle$

Figure 2: Core Scheme Syntax (Runtime objects)

ciently expressive to encode the various derived expression forms that appear in realistic Scheme programs. Hence, we expect an external Scheme program to first have derived expression forms expanded into primitive expression forms and symbolic variables converted into deBruijn indices, yielding a Core Scheme expression, before being executed. As explained in Section 2.1, we use an external Scheme front-end to do so, producing a Cyclone function that builds an abstract syntax tree.

Figure 2 describes the syntax of run-time objects that appear during the evaluation of a Core Scheme expression and Figure 3 describes the operational semantics of the language. The operational semantics define a small-step rewriting relation from states to states. A state has the form $\langle H, S, \rho, r \rangle$, where H is the heap, S is the stack, ρ is the environment, and r is the result (expression or value) in the current active position.

A heap is a partial map from an infinite set of locations (*Loc*) to values. A stack is a finite sequence of environments and stack frames. A stack frame is a partially evaluated expression with a hole, corresponding to Wright and Felleisen-style semantics [21]. Conditional and assignments have exactly one hole, while applications enforce a left-to-right evaluation order by requiring locations (corresponding to heap allocated values) to the left of the hole and expressions to the right of the hole.

An environment is a sequence of locations; each location is expected to corresponds to a heap allocated vector. The deBruijn index i_j denotes the j^{th} element of the vector pointed to by the i^{th} location in the environment. We further assume an initial heap H_0 and an initial environment ρ_0 in which all primitive operations are bound.

```
struct Value<`r::R>;
typedef struct Value<`r>*`r value_t<`r::R>;
datatype ValueD<`r> {
  Const_v(const_t<`r> K);
  Primop_v(primop_t p);
  Throw_v(stack_t<`r> S, env_t<`r> rho);
  Closure_v(unsigned int n,
            env_t<`r> rho, exp_t<`r> e);
  VarArgsClosure_v(unsigned int m,
                   env_t<`r> rho, exp_t<`r> e);
  Pair_v(value_t<`r> l1, value_t<`r> l2);
  Vector_v(value_t<`r>*@fat`r ls);
};
struct Value<`r::R> {
  datatype ValueD<`r> value;
};
```

Figure 4: Cyclone Implementation (Values)

A primitive operation is a partial map from a heap, stack, environment, and location sequence to a state. The location sequence corresponds to the arguments to the primitive operation. The other components of the state are provided for primitive operations that perform heap allocation or require access to the stack and environment.

Values correspond to heap allocated data. In addition to primitive operations, there are three other procedure forms. The **throw** carries a stack and environment; this is the form of the "escape procedure" passed as an argument to the argument of the primitive operation `call-with-current-continuation`. The **closure** and **vaclosure** forms correspond to the evaluation of **lambda** and **valambda** expression-types; the closure forms capture the current environment to be extended at the point of application.

The final two value forms, **pair** and **vector**, combine locations into larger data-structures.

Figure 3 describes the operational semantics of the language. The main judgment $\sigma \rightarrow \sigma'$ defines a small-step rewriting relation from states to states. Rules in which the result is an expression either immediately return a location (possibly having performed an allocation) or push a frame onto the stack and start evaluating a sub-expression. Rules in which the result is a location pop a frame, fill the hole, and continue as appropriate. The auxiliary judgments $\langle H, i_j\ @\ \rho \rangle \Downarrow_{\text{get}} l$ and $\langle H, i_j\ @\ \rho \leftarrow l \rangle \Downarrow_{\text{set}} H'$ are concerned with getting and setting deBruijn indexed variables, while the judgment $\langle H, \langle l_1, \ldots, l_n \rangle \rangle \Downarrow_{\text{listify}} \langle H', l' \rangle$ allocates arguments in a Scheme list.

Implementation

We turn now to the implementation of the Core Scheme interpreter in Cyclone. As Cyclone is a C-like language, values are represented using allocated data-structures and pointers (see Figure 4).

Recall from Section 2.2 that region polymorphism lets data-structures abstract over the region of their fields. Hence, in Figure 4, `struct Value<`r::R>` is a forward declaration of a structure type, which is polymorphic in a region `` `r ``. The `typedef` defines `value_t<`r>` to be a (nullable) pointer to a `struct Value<`r>` object in region `` `r ``; essentially, `value_t<`r>` corresponds to locations in the op-

$\boxed{\langle H, \rho \mathbin{@} i_j\rangle \Downarrow_{\text{get}} l}$

$$\frac{H(l) = (\textbf{vector } \langle l_1, \ldots, l_n\rangle)}{\langle H, l :: \rho \mathbin{@} 0_j\rangle \Downarrow_{\text{get}} l_j}(1 \le j \le n) \qquad \frac{\langle H, \rho \mathbin{@} i_j\rangle \Downarrow_{\text{get}} l'}{\langle H, l :: \rho \mathbin{@} (i+1)_j\rangle \Downarrow l'}$$

$\boxed{\langle H, \rho \mathbin{@} i_j \leftarrow l\rangle \Downarrow_{\text{set}} H'}$

$$\frac{H(l) = (\textbf{vector } \langle l_1, \ldots, l_n\rangle)}{\langle H, l :: \rho \mathbin{@} 0_j \leftarrow l'\rangle \Downarrow_{\text{set}} H[l \mapsto (\textbf{vector } \langle l_1, \ldots, l_{j-1}, l', l_{j+1}, \ldots, l_n\rangle)]}(1 \le j \le n) \qquad \frac{\langle H, \rho \mathbin{@} i_j \leftarrow l'\rangle \Downarrow_{\text{set}} H'}{\langle H, l :: \rho \mathbin{@} (i+1)_j \leftarrow l'\rangle \Downarrow_{\text{set}} H'}$$

$\boxed{\langle H, \langle l_1, \ldots, l_n\rangle\rangle \Downarrow_{\text{listify}} \langle H', l'\rangle}$

$$\frac{l' \notin dom(H)}{\langle H, \langle\rangle\rangle \Downarrow_{\text{listify}} \langle H[l' \mapsto (\textbf{const } \mathsf{nil})], l'\rangle} \qquad \frac{\langle H, \langle l_2, \ldots, l_n\rangle\rangle \Downarrow_{\text{listify}} \langle H', l'\rangle \quad l'' \notin dom(H')}{\langle H, \langle l_1, \ldots, l_n\rangle\rangle \Downarrow_{\text{listify}} \langle H'[l'' \mapsto (\textbf{pair } l_1\ l')], l''\rangle}$$

$\boxed{\sigma \to \sigma'}$

$$\text{CONST}\frac{l \notin dom(H)}{\langle H, S, \rho, K\rangle \to \langle H[l \mapsto (\textbf{const } K)], S, \rho, l\rangle} \qquad \text{VAR}\frac{\langle H, \rho \mathbin{@} i_j\rangle \Downarrow_{\text{get}} l}{\langle H, S, \rho, i_j\rangle \to \langle H, S, \rho, l\rangle}$$

$$\text{APPLY}\frac{}{\langle H, S, \rho, (e\ e_1\ \ldots\ e_n)\rangle \to \langle H, \langle(\Box\ e_1\ \ldots\ e_n), \rho\rangle :: S, \rho, e\rangle}$$

$$\text{LAMBDA}\frac{l \notin dom(H)}{\langle H, S, \rho, (\textbf{lambda}^n\ e)\rangle \to \langle H[l \mapsto (\textbf{closure}^n\ \rho\ e)], S, \rho, l\rangle}$$

$$\text{VALAMBDA}\frac{l \notin dom(H)}{\langle H, S, \rho, (\textbf{valambda}^n\ e)\rangle \to \langle H[l \mapsto (\textbf{vaclosure}^m\ \rho\ e)], S, \rho, l\rangle}$$

$$\text{IF}\frac{}{\langle H, S, \rho, (\textbf{if } e_1\ e_2\ e_3)\rangle \to \langle H, \langle(\textbf{if } \Box\ e_2\ e_3), \rho\rangle :: S, \rho, e_1\rangle} \qquad \text{SET!}\frac{}{\langle H, S, \rho, (\textbf{set!}\ i_j\ e)\rangle \to \langle H, \langle(\textbf{set!}\ i_j\ \Box), \rho\rangle :: S, \rho, e\rangle}$$

$$\text{APPLY-ARG-EVAL}\frac{}{\langle H, \langle(l_1\ \ldots\ l_{i-1}\ \Box\ e_{i+1}\ \ldots\ e_n), \rho'\rangle :: S, \rho, l\rangle \to \langle H, \langle(l_1\ \ldots\ l_{i-1}\ l\ \Box\ e_{i+2}\ \ldots\ e_n), \rho'\rangle :: S, \rho', e_{i+1}\rangle}$$

$$\text{APPLY-PRIM-EVAL}\frac{H(l) = (\textbf{prim } p)}{\langle H, \langle(l\ l_1\ \ldots\ l_{n-1}\ \Box), \rho'\rangle :: S, \rho, l_n\rangle \to p(H, S, \rho, \langle l_1, \ldots, l_n\rangle)}$$

$$\text{APPLY-THROW-EVAL}\frac{H(l) = (\textbf{throw } S''\ \rho'')}{\langle H, \langle(l\ \Box), \rho'\rangle :: S, \rho, l'\rangle \to \langle H, S'', \rho'', l'\rangle}$$

$$\text{APPLY-CLOSURE-EVAL}\frac{H(l) = (\textbf{closure}^n\ \rho''\ e'') \quad l'' \notin dom(H)}{\langle H, \langle(l\ l_1\ \ldots\ l_{n-1}\ \Box), \rho'\rangle :: S, \rho, l_n\rangle \to \langle H[l'' \mapsto (\textbf{vector } \langle l_1, \ldots, l_n\rangle)], S, l'' :: \rho'', e''\rangle}$$

$$\text{APPLY-VACLOSURE-EVAL}\frac{H(l) = (\textbf{vaclosure}^m\ \rho''\ e'') \quad \langle H, \langle l_{m+1}, \ldots, l_n\rangle\rangle \Downarrow_{\text{listify}} \langle H', l'\rangle \quad l'' \notin dom(H')}{\langle H, \langle(l\ l_1\ \ldots\ l_{n-1}\ \Box), \rho'\rangle :: S, \rho, l_n\rangle \to \langle H'[l'' \mapsto (\textbf{vector } \langle l_1, \ldots, l_m, l'\rangle)], S, l'' :: \rho'', e''\rangle}(m \le n)$$

$$\text{IF-TRUE-EVAL}\frac{H(l) \neq (\textbf{const } \mathsf{false})}{\langle H, \langle(\textbf{if } \Box\ e_1\ e_2), \rho'\rangle :: S, \rho, l\rangle \to \langle H, S, \rho', e_1\rangle} \qquad \text{IF-FALSE-EVAL}\frac{H(l) = (\textbf{const } \mathsf{false})}{\langle H, \langle(\textbf{if } \Box\ e_1\ e_2), \rho'\rangle :: S, \rho, l\rangle \to \langle H, S, \rho', e_2\rangle}$$

$$\text{SET!-EVAL}\frac{\langle H, \rho' \mathbin{@} i_j \leftarrow l\rangle \Downarrow_{\text{set}} H' \quad l' \notin dom(H')}{\langle H, \langle(\textbf{set!}\ i_j\ \Box), \rho'\rangle :: S, \rho, l\rangle \to \langle H'[l' \mapsto (\textbf{const } \mathsf{unspecified})], S, \rho', l'\rangle}$$

Figure 3: Core Scheme Operational Semantics

```
void scheme(exp_t<`r> prog<`r>(region_t<`r>)) {
  // load the program into the Cyclone heap
  exp_t<`H> e = prog(heap_region);
  // load the initial environment
  env_t<`H> env = initial_env(heap_region);
  // construct the initial state
  //  - an empty stack,
  //  - the initial environment
  //  - the expression to the evaluated
  state_t<`H> state = State{NULL,env,{.expr = e}};
  // take an unbounded number of steps
  bool done = stepi(-1,heap_region,&state);
}
```

Figure 5: Heap Allocated Interpreter

erational semantics (and `NULL` corresponds to (**const** nil)).

The concrete representation of an expression is given by the `datatype ValueD<`r>` declaration. Cyclone provides *datatypes* as a safe alternative to unions for supporting heterogenous values. Furthermore, while unions require space proportional to the largest member, a datatype only requires space for the member being used and is thus more efficient in its use of memory. Note the use of a fat pointer in the `Vector_v` variant of the `ValueD<`r>` datatype, which ensures the saftey of array accesses and supports querying the vector's length.

At the present time, the `struct Value` declaration serves little purpose, as it simply embeds a `datatype ValueD<`r>` as the only member of a structure. In Section 2.4, we will see how the garbage collector extends this structure with a forwarding pointer.

Expressions are immutable – they do not change during the execution of the program. However, as Cyclone is a C-like language, an expression is represented using allocated data-structures and pointers, in a manner similar to values. Stacks and environments are represented as linked list structures, where `NULL` corresponds to an empty stack or an empty environment.

We conclude this section by examining a simple Core Scheme interpreter (see Figure 5), in which all data is allocated in the Cyclone heap and managed by a conservative garbage collector. The function `scheme` takes one argument, pointer to a function that takes a region handle argument and returns an expression allocated in that region. (Note that the `<`r>` annotation universally quantifies the region variable.) The remainder of the code is straightforward. First, the function pointer is executed to yield the initial expression. Next, the initial environment is allocated. The initial expression and environment are combined with an empty stack to create the initial state. Finally, the interpreter repeatedly makes transitions corresponding to $\sigma \rightarrow \sigma'$, until termination. Note that all region parameters in expression, environment, and state types are instantiated with the heap region (`H`) and all functions performing allocations take the heap-region handle (`heap_region`) as a parameter. (Unlike the operational semantics given above, the state is implicitly side-effected and no state is returned by the `stepi` function.)

2.4 The Garbage Collector

In this section, we describe a safe, copying garbage collector for the Core Scheme language described above. Before proceeding, we point out some novelties in comparison to other type-preserving garbage collectors [15, 20]. As demonstrated in the previous section, we choose to write our interpreter in a trampoline style, rather than in a continuation passing style. This gives rise to a simpler implementation, one more in keeping with a byte-code interpreter. Previous work requires a continuation passing style, with each continuation polymorphic in the heap region, complicating the garbage collector interface. Also, as will be explained, the use of dynamic-region sequences and the `next_rgn` type constructor gives rise to a more natural typing of forwarding pointers.

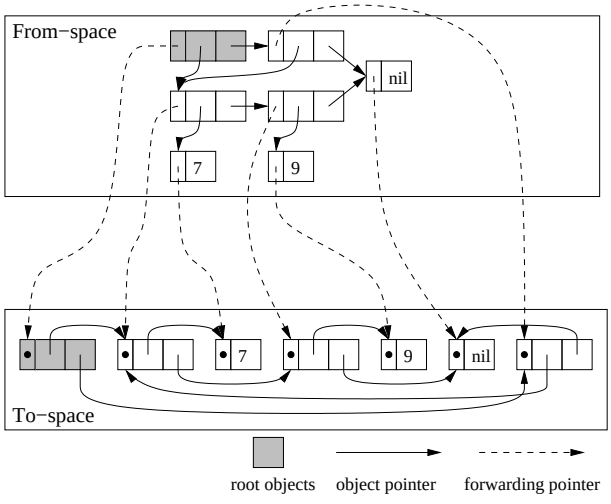

Figure 6: Copying GC Example

The remainder of this section proceeds in the following manner. First, we briefly review the copying algorithm. Next, we consider an intuitive implementation in Cyclone. Finally, we refine our implementation using our *dynamic-region sequence* abstraction to achieve a complete implementation in Cyclone.

Figure 6 illustrates a simple stop-and-copy collector. The collector stops the user program and begins with (live and dead) cells in the *From-space* and an empty *To-space*. The collector traverses the data structure in From-space, copying each live cell into To-space when the cell is first visited. The collector preserves sharing by leaving a forwarding pointer in each From-space cell when it is copied. The forwarding pointer points to the copied cell in To-space. Whenever a cell in From-space is visited, the copying function first examines the forwarding pointer. If it is non-NULL, then the copy function returns the forwarding address. Otherwise, space for the cell is reserved in To-space, the forwarding pointer is set to the address of the reserved space, and the fields of the cell are copied. After all live cells in From-space have been copied (depicted in Figure 6), the collector can free all memory in From-space and the program can continue execution, allocating new cells in To-space.

In the copying algorithm, the separation of managed memory into a From-space and a To-space suggests a natural correspondence with Cyclone's regions. Clearly, the LIFO discipline of Cyclone's lexical regions is insufficient for our copying garbage collector, as we would like the lifetime of From-space to end after the beginning but before the end of To-space's lifetime. Hence, we turn our attention to

Cyclone's dynamic regions, where it appears that we have sufficient expressiveness to write a simple copying garbage collector:

```
...
// create the to-space's key
let NewDynamicRegion {<`to> to_key} = new_ukey();
state_t<`to> to_state;
// open the from-space's key
{ region from_r = open(from_key);
  // open the to-space's key
  { region to_r = open(to_key);
    to_state = copy_state(to_r, from_state);
  }
}
// free the from-space
free_ukey(from_key);
...
```

While this captures the spirit of the garbage collector, a number of details remain. As we described in Section 2.3, the `struct Value<`r::R>` declaration is intended to be extended with a forwarding pointer. The question is: "what is the type of the forwarding pointer?" The answer is (something along the lines of): "a pointer to a `struct Value` in To-space, whose forwarding pointer is a pointer to a `struct Value` in To-space's To-space, whose forwarding pointer ...." What we need is a name for all of the unwindings of the infinite sequence of pointers. We provide such a name in the form of a type constructor, which maps region names to region names and generates an infinite supply of region names:

```
typedef _::R next_rgn<`r::R>
```

Thus, we give each garbage-collected value a forwarding pointer in the following manner:

```
typedef struct Value<`r>*`r value_t<`r::R>;
struct Value<`r::R> {
  value_t<next_rgn<`r>> forward;
  datatype ValueD value;
};
```

Note that although the region names `` `r `` and `next_rgn<`r>` are related, the lifetimes of their corresponding regions are not. In a similar manner, `value_t<next_rgn<`r>>` is a well-formed type anywhere in the scope of `` `r `` – even if the region handle corresponding to `next_rgn<`r>` has not been created. The original inspiration for the `next_rgn` type constructor comes from Hawblitzel et. al. [7], where *type sequences* (mappings from integers to types) are used to index regions by a region number ("epoch"), yielding the connection between successive regions in a copying collector.

Operationally, we expect dynamic region sequences to behave as dynamic regions, with access mediated by a key. In particular, we expect operations to create, open, and free dynamic-region sequence keys. In addition, we need an operation to produce the key for `next_rgn<`r>`. While this operation yields an infinite supply of dynamic region sequence keys, it is not quite enough. We need to ensure that the supply is linear; i.e., there is exactly one way to generate a key for `next_rgn<`r>` for any `` `r ``. Cyclone's unique pointers provide exactly this linearity. Thus, we declare

```
struct DynamicRegionGen<`r::R>;
typedef struct DynamicRegionGen<`r>*@notnull`U
     uregion_gen_t<`r>;
```

where `struct DynamicRegionGen<`r::R>` is abstract. Because `uregion_gen_t<`r>` is a unique pointer, it serves as a capability; in particular, it will serve as a capability to produce the key for `<next_rgn<`r>>` and the next generator. Taken together, a key and a generator form a dynamic-region sequence:

```
struct DynamicRegionSeq<`r> {
  uregion_key_t<`r> key;
  uregion_gen_t<`r> gen;
};
typedef struct DynamicRegionSeq<`r> drseq_t<`r>;
```

Finally, the interface for dynamic-region sequences is given by the following:

```
struct NewDynamicRegionSeq { <`r::R>
  struct DynamicRegionSeq<`r> drseq;
};
struct NewDynamicRegionSeq new_drseq();
struct DynamicRegionSeq<next_rgn<`r>>
     next_drseq(uregion_gen_t<`r> gen);
```

The `new_drseq` function creates a fresh dynamic-region sequence and returns the key to the first region and a generator for the next element in the sequence. (Note that the `new_drseq` function returns a structure that existentially quantifies the region variable. Unpacking this existential yields a region variable which does not conflict with any other region name.) The `next_drseq` function returns the key for `next_rgn<`r>` and a new generator. Because the generator is unique and the `next_drseq` function consumes it, it follows that we can only create one key for `next_rgn<`r>`; hence, the sequence of regions is linear. Furthermore, the key (and region handle) for `next_rgn<`r>` need only be created when `next_drseq` is called; no keys or handles need be pre-allocated. Opening and freeing a key is accomplished through the dynamic region operations: the `open` primitive and the `free_ukey` function.

Figures 7 and 8 present the Cyclone code implementing the central functions in the Core Scheme garbage collector and interpreter. The `struct GCState` existentially binds the current region being used as the Core Scheme heap; each time the interpreter or garbage collector needs to access or allocate data in this region, the state is unpacked and the region opened. The `doGC` function directly implements the example code given above, modified to pack and unpack the state and to generate the To-space from the dynamic-region sequence. The `scheme` function is very similar to the one given in Figure 5, modified to create the dynamic-region sequence and allocate the initial program and environment in the initial dynamic region. The termination check uses a `goto` in order to transfer control from a scope in which the current region is opened to a scope where the current region can be freed. This ensures that the final heap is reclaimed before terminating the Core Scheme interpreter.

```
// the gc state encapsulates
//  - the current region (existentially bound)
//  - a dynamic region sequence (containing a key and generator)
//  - the state (containing roots)
typedef struct GCState { <`r>
  drseq_t<`r> key_gen;
  state_t<`r> state;
} gcstate_t;

gcstate_t doGC(gcstate_t gcs) {
  // unpack the gc state, naming the existentially bound region (`r)
  let GCState{<`r> DynamicRegionSeq {from_key, from_gen}, from_state} = gcs;
  // generate the to-space (next_rgn<`r>)
  let DynamicRegionSeq{to_key, to_gen} = next_drseq(from_gen);
  state_t<next_rgn<`r>> to_state;
  // open the from-space's key
  { region from_r = open(from_key);
    // open the to-space's key
    { region to_r = open(to_key);
      // copy the state and reachable data
      to_state = copy_state(to_r, from_state);
    }
    // pack the new gc state
    gcs = GCState{DynamicRegionSeq{to_key, to_gen}, to_state};
  }
  // free the from-space and return the new gc state
  free_ukey(from_key);
  return gcs;
}
```

Figure 7: Garbage Collector

```
void scheme(exp_t<`r> prog<`r>(region_t<`r>)) {
  // construct the initial heap
  let NewDynamicRegionSeq {<`r> DynamicRegionSeq{key, gen}} = new_drseq();
  // load the program and initial environment into the initial heap
  exp_t<`r> e;
  env_t<`r> env;
  { region r = open(key);
    e = prog(r);
    env = initial_env(r);
  }
  // construct the initial state
  state_t<`r> state = State{NULL,env,{.expr = e}};
  // construct the initial gc state with
  //  - the dynamic region sequence
  //  - the initial state
  struct GCState gcs = GCState{{key,gen},state};
  while (true) {
    // unpack the current gc state, naming the existentially bound region (`s)
    let GCState{<`s> DynamicRegionSeq{key, gen}, state} = gcs;
    { // open the current heap
      region r = open(key);
      // take a fixed number of steps
      bool done = stepi(MAX_STEPS,r,&state);
      // check for termination
      if (done) { goto Finished; }
    }
    // pack the new gc state and allow a GC
    gcs = GCState{DynamicRegionSeq{key, gen}, state};
    gcs = maybeGC(gcs);
  }
 Finished:
  // unpack the final gc state and free the final region
  let GCState{<`r> DynamicRegionSeq{key, gen}, state} = gcs;
  free_ukey(key);
}
```

Figure 8: Dynamic Region Sequence Allocated Interpreter

```
static $(frontier_t<`r>, value_t<next_rgn<`r>>)
  copy_value(frontier_t<`r> frontier,
             region_t<next_rgn<`r>> to_r,
             value_t<`r> from_obj) {
  switch (from_obj) {
  case NULL: return $(frontier, NULL);
  case &Value{*f,*from_obj_}:
    // if a forwarding pointer is installed,
    //  return it and an unmodified frontier.
    if (*f != NULL) return $(frontier,*f);
    switch (from_obj_) {
    case &Closure_v(n,from_env,from_exp):
      // allocate a new Closure in to-space,
      //  extracting the addresses of child pointers
      let to_obj as
        &Value{_,Closure_v(_,*to_envp,*to_expp)} =
        rnew(to_r) Value{NULL,Closure_v(n,NULL,NULL)};
      // install the forwarding pointer
      *f = to_obj;
      // add children to the frontier
      frontier = add_front(frontier,copy_env,
                           from_env,to_envp);
      frontier = add_front(frontier,copy_exp,
                           from_exp,to_expp);
      // return new frontier and pointer
      return $(frontier, to_obj);
    case &Pair_v(from_obj1,from_obj2):
      // allocate a new Pair in to-space,
      //  extracting the addresses of child pointers
      let to_obj as
        &Value{_,Pair_v(*to_obj1p,*to_obj2p)} =
        rnew(to_r) Value{NULL,Pair_v(NULL,NULL)};
      // install the forwarding pointer
      *f = to_obj;
      // add children to the frontier
      frontier = add_front(frontier,copy_value,
                           from_obj1,to_obj1p);
      frontier = add_front(frontier,copy_value,
                           from_obj2,to_obj2p);
      // return new frontier and pointer
      return $(frontier,to_obj);
    ...
    }
  }
}
```

Figure 9: Copy Function (Values)

Thus far, we have said little about the actual copying of Core Scheme objects. A careful inspection of the code in Figure 8 reveals that expressions, environments, and stacks are allocated in the Scheme heap, in addition to values. Hence, objects of each kind must be garbage collected. The copying algorithm needs space to manage the traversal of live data.[3] A recursive algorithm that uses the run-time stack to maintain state runs the risk of stack overflow. Therefore, we must maintain an explicit frontier of objects to be forwarded. Our solution is to use unique pointer allocated data-structures which are freed at the end of the garbage collection. We use unique pointers in order to immediately free the data-structure if it must be resized to accommodate a larger frontier. During a garbage collection, three regions are used: the From-space (`r), the To-space (next_rgn<`r>), and the unique region (`U).

Figure 9 presents the Cyclone code for copying a Core Scheme value. The &Value{*f,*from_obj_} "label" makes use of Cyclone's pattern matching facilities, which provide an efficient method of binding parts of large objects to new local variables. In particular, this pattern binds f to the address of the forwarding pointer component of the Value<`r> structure and from_obj_ to the address of the datatype ValueD<`r> component. The comparison *f != NULL checks for an installed forwarding pointer; if one exists, it is returned with an unmodified frontier. If no forwarding pointer has been installed, then the value variant is determined, a new object is allocated in To-space (via rnew(to_r)), the forwarding pointer is installed (*f = to_obj), and child pointers are extracted and added to the frontier (via add_front).

We have also said little about the triggering of a garbage collection. Currently, the Cyclone region interface allows the programmer to query the current size of allocated data in a region. We use a simple live-ratio method to set a heap limit, which is checked after a fixed number of interpreter steps. It is straightforward to extend the Cyclone region interface to establish an upper bound on a region's size, causing an exception to be raised when an allocation would exceed the limit. The only complication is how to resume the computation after the exception has occurred. We must structure parts of our interpreter and primitives to guarantee they can be correctly resumed in the case of such a region exhaustion exception is rasied.

3. PERFORMANCE EVALUATION

Now that we have outlined the design and implementation of our safe Scheme interpreter and runtime system, we wish to evaluate the performance of the system compared to various alternatives. Because we have constrained ourselves to programming in a safe language, some standard implementation techniques that are applicable in unsafe implementations are unavailable to us. We want to understand how much of a performance penalty this causes. For comparison we compare our system with a safe garbage collector against three different other Scheme implementations.

The Scheme Implementations and Benchmark Programs

We summarize the systems we study below.

Cyclone Safe GC - Our implementation of a Core Scheme interpreter with a safe GC also written in Cyclone; essentially, that of Figure 8.

Cyclone BDW GC - The same implementation of our Core Scheme interpreter, but using the Bohem-Demers-Weiser conservative collector; essentially that of Figure 5.

SISC Sun JVM - A freely available high-performance implementation of Scheme written in Java [14]. It supports first-class continuations and is fully tail-recursive. It relies on the native Java VM for it's runtime memory management.

MzScheme BDW GC - Widely used implementation of Scheme written in C [16]. It also uses the Bohem-Demers-Weiser conservative collector for storage management.

[3] A Cheney-style copying collector cannot be accommodated in this framework, as we cannot iterate through the objects allocated in a region.

The table below summarizes the safety guarantees provided by each system.

	Interpreter	Runtime System
Cyclone Safe GC	Safe	Safe
Cyclone BDW GC	Safe	Unsafe
SISC Sun JVM	Safe	Unsafe
MzScheme BDW GC	Unsafe	Unsafe

By comparing the performance of the Cyclone Safe GC and Cyclone BDW GC, we hope to understand the performance cost of a safe garbage collector independent of the core interpreter. Comparing SISC with both our Cyclone versions gives us an idea of how efficient our core interpreter is compared with another realistic safe implementation. Finally, comparing MzScheme with the other systems give us an idea of how much performance can be had if we forgo safety entirely.

We report execution times and memory consumption for all these systems on a subset of the Gabriel benchmarks[4] translated into Scheme.

Important Caveats

Before we begin, we should highlight some important differences between the various systems in-order to properly interpret the meaning of the benchmarks. The first caveat is that not all systems implement the same subset of Scheme. The Cyclone-based systems implement the bare minimum set of primitives in order to properly execute the benchmark applications. The Cyclone-based systems also have a space leak associated with the storage of symbols created via `string->symbol`. This leak occurs because we do not have support for weak pointers to properly collect useless entries from the symbol table. It is straightforward to extend our system to do so. The Cyclone system does not implement the full numeric tower or multiple return values. Doing both is straightforward. We do not expect the lack of these features to significantly skew our results. Like the other implementations our Cyclone implementations are fully tail-recursive and support first-class continuations.

MzScheme stack allocates activations frames and uses `setjmp/longjmp` to implement first-class continuations, while all of the other implementations heap allocate activation frames. This difference is apparent in the one benchmark (`ctak`) which heavily relies on first-class continuations. Our safe GC uses a copying GC, the BDW collector uses a mark-sweep GC. To the best of our knowledge the JVM uses a hybrid two-generation scheme, where the first generation use a copying GC and the second generation uses a mark-sweep algorithm. These algorithmic difference makes fair comparisons tricky to do at best.

In performing the benchmarks we do not artificially limit the heap usage of each program. Since each program uses a different amount of heap space and space can be traded for time, it is not fair to simply examine execution times without taking into consideration memory usage. However, since it is tricky and difficult to limit memory usage to create a "fair" playing field we simply report the time and space metrics for "out of the box" configurations. Even with these caveats we can still come to some sound qualitative conclusions.

[4]Note `nboyer` is a corrected version of the original `boyer` benchmark that fixes correctness bugs in the original benchmark.

Total Execution Time

Our first experiment simply measures the execution time for each benchmark. The execution times are measured in terms of absolute wall clock time and *do not* include start up overheads for any preprocessing or compilation. Table 1 summarizes the absolute execution times for the various systems on the benchmark programs. All times reported are in milliseconds on a lightly loaded 993Mhz dual-processor x86 with 256KB of L2 cache and 2GB of physical memory, running a Linux 2.4 kernel.

The Cost of Safety

With the exception of `ctak`, it is clear that MzScheme is significantly faster than all of the other systems. The performance difference for `ctak` is due to the fact that MzScheme does not bias the implementation in-order to support cheap first-class continuations. Some preliminary experiments with stack allocation of activation frames in our Cyclone implementation suggest that the performance difference between the systems can not be simply be accounted for because of stack allocation. Being unsafe, MzScheme is able to use more compact and efficient data representations such as using the lower order bits of a pointer to encode type tags. The other systems implemented in safe languages are forced to use more heavyweight representations.

Comparison of Safe Interpreters

If we examine the ratio of execution times with respect to the Cyclone Safe GC (presented in parentheses in Table 1), we can see the relative costs of safety. Looking at the normalized execution times it is clear that all three safe systems are roughly comparable. Our Cyclone implementation is faster in many cases when compared to SISC, but is slower in others cases. Given that a great deal of effort has been placed in optimizing SISC as well as the Java VM on which it runs, we are happy with the performance of our core interpreter. We hope to make the gap between the Cyclone based interpreter meet or exceed the performance of SISC in all the benchmarks.

Doing so will probably require changing our rather straightforward implementation of the Scheme operational semantics into more optimized code. We have applied many of the key algorithmic optimizations [14] used by SISC into our interpreter, but still believe there is room for optimization of the core implementation as well as library procedures. We also believe having the flexible memory management primitives that Cyclone offers may also provide a significant boost in the performance of web application servers. We discuss this in more detail in Section 5.1.

Comparison of Safe vs. Unsafe GC

Examining the execution times of Cyclone Safe GC and Cyclone BDW GC, we can see that our safe collector does not add any significant run-time penalty. In fact in many cases it is slightly faster. However, this may be a misleading conclusion since the underlying GC algorithms are different. Our safe GC uses a two-space copying GC while the comparable BDW GC is using a mark-sweep algorithm. This difference is obvious when we look at the memory footprint of the various implementations.

We provide two metrics to help understand memory consumption. One is based on the garbage collector's view of

	Cyclone Safe GC		Cyclone BDW GC		SISC Sun JVM		MzScheme BDW GC	
`cpstak`	248	(1.00)	263	(1.06)	279	(1.12)	93	(0.37)
`ctak`	357	(1.00)	381	(1.07)	412	(1.15)	3590	(10.05)
`deriv`	1160	(1.00)	1145	(0.99)	660	(0.57)	190	(0.16)
`destruct`	1027	(1.00)	1009	(0.98)	1027	(1.00)	351	(0.34)
`div-iter`	466	(1.00)	467	(1.00)	360	(0.77)	139	(0.30)
`div-rec`	448	(1.00)	453	(1.01)	363	(0.81)	133	(0.30)
`fft`	234	(1.00)	223	(0.96)	450	(1.92)	36	(0.16)
`nboyer`	4292	(1.00)	5018	(1.17)	3431	(0.80)	867	(0.20)
`puzzle`	55	(1.00)	62	(1.13)	51	(0.93)	11	(0.20)
`tak`	209	(1.00)	226	(1.08)	375	(1.79)	44	(0.21)
`takl`	2134	(1.00)	2281	(1.07)	2010	(0.94)	434	(0.20)
`takr`	241	(1.00)	258	(1.07)	238	(0.99)	46	(0.19)
`traverse`	19060	(1.00)	23594	(1.24)	16632	(0.87)	4168	(0.22)

Table 1: Total Execution Time in Milliseconds and with Ratio of Execution Times Normalized to Cyclone Safe GC in Parentheses

the current heap size. The other is based on the real memory in use as reported by the underlying operating system. Unfortunately, there is no easy portable way to measure the actual heap usage of each system without including the startup costs associated with each system.

The numbers we report include the overhead for start up and compilation/parsing of the benchmark programs. This makes SISC and MzScheme look artificially worse since we are also measuring the overhead of their entire system, including the infrastructure need to support an interactive REPL. We provide these numbers for completeness, but will focus on the comparisons between our two different Cyclone versions. Both these systems have comparably small startup costs.

To collect information about heap size, we enable debugging information printed at each invocation of the GC. For our own Cyclone Safe GC collector we simply instrumented our collector appropriately. For the BDW collector we set the internal variable `GC_print_stats` which causes the BDW collector to report the heap size after each GC. For the JVM we simply enabled the `-verbosegc` option to collect information about heap sizes. Table 2 summarizes the maximum heap size as reported by the various collectors. Note that we do not report some numbers for programs running under MzScheme because these programs do not cause a garbage collection to occur.

If we look at the heap sizes for our two Cyclone systems, we see that for small programs our heap sizes are comparable, however one can see systematic difference in larger programs such as `nboyer` and `traverse`. The heap resizing policy of our safe copying collector resizes the heap based on the amount of reachable data that remains in the heap after a collection and a tunable "live ratio" parameter. In all experiments we set our live ratio to 4.0. A larger live ratio amortizes the cost of the a GC over more of the program execution by consuming more space.

While heap sizes is a useful metric in a system with virtual memory, a more important metric is the amount of physical memory needed to hold the program's working set. Table 3 summarizes the size of the maximum resident set of virtual memory pages as measured by repeated polling of the `/proc/` file system for each benchmark during execution. While the first numbers give us some insight into the algorithmic consumption of space, these numbers allow us to understand how those numbers may translate into real system performance.

It is clear from both measurements that our Cyclone implementation using the Safe GC has a significantly larger footprint when compared to the equivalent implementation that uses the BDW collector. This difference is accounted for by the use of a copying GC compared to a mark-sweep GC.

Figure 10 show the resident set size as a function of time for the various systems during the execution of the `nboyer` benchmark. The graphs clearly show the difference in the algorithms used, and show that the actual working set size at a particular point in time can be significantly smaller than our maximum set. Regardless of this fact, the extra copying will have a negative effect on overall performance if we consider the costs of servicing page faults. In our current experiments, memory is not a scare resource, so most page faults can be serviced without going to disk. In a realistic server environment, all others things being equal, our copying GC will perform worse.

Although a given server typically will have large amounts of memory available, that memory must be shared across many different instances of an application. Each application has a limited amount of memory. Therefore, it is unrealistic to assume that memory is cheap or free. We should explore how to implement other techniques such as mark-sweep or generational-collection algorithms for our approach to be competitive with other unsafe approaches. However, our overall approach has other system level benefits that may recover or ameliorate these costs in a web-application server. We touch on these benefits in Section 5.1.

4. SIZE OF UNSAFE CODE

The primary goal of our approach is not to produce a faster system, but to produce a system which we believe is safer. It is important to compare the amount of unsafe code in the system. Table 4 summarizes the approximate amount of unsafe C code used in the implementation of each system. The line counts ignore comments and whitespace.

The amount of unsafe C code need to implement the ba-

	Cyclone Safe GC		Cyclone BDW GC		SISC Sun JVM		MzScheme BDW GC	
cpstak	276	(1.00)	292	(1.06)	3306	(11.98)	1024	(3.71)
ctak	282	(1.00)	292	(1.04)	3312	(11.74)	3180	(11.28)
deriv	381	(1.00)	392	(1.03)	3319	(8.71)	1024	(2.69)
destruct	327	(1.00)	392	(1.20)	3364	(10.29)	1024	(3.13)
div-iter	360	(1.00)	392	(1.09)	3316	(9.21)	1024	(2.84)
div-rec	390	(1.00)	392	(1.01)	3319	(8.51)	1024	(2.63)
fft	794	(1.00)	700	(0.88)	4065	(5.12)	N/A	
nboyer	8913	(1.00)	2232	(0.25)	5522	(0.62)	2012	(0.23)
puzzle	1554	(1.00)	700	(0.45)	3438	(2.21)	1024	(0.66)
tak	269	(1.00)	292	(1.09)	3291	(12.23)	N/A	
takl	282	(1.00)	392	(1.39)	3301	(11.71)	N/A	
takr	955	(1.00)	524	(0.55)	3510	(3.68)	1024	(1.07)
traverse	4984	(1.00)	1672	(0.34)	4227	(0.85)	2012	(0.40)

Table 2: Maximum Heap Size in Kilobytes Observed at GC Points with Ratio of Sizes Normalized to Cyclone Safe GC in Parentheses.

	Cyclone Safe GC		Cyclone BDW GC		SISC Sun JVM		MzScheme BDW GC	
cpstak	298	(1.00)	241	(0.81)	5072	(17.02)	652	(2.19)
ctak	301	(1.00)	239	(0.79)	5071	(16.85)	1203	(4.00)
deriv	335	(1.00)	276	(0.82)	5076	(15.15)	651	(1.94)
destruct	316	(1.00)	276	(0.87)	5097	(16.13)	652	(2.06)
div-iter	323	(1.00)	271	(0.84)	5067	(15.69)	650	(2.01)
div-rec	301	(1.00)	271	(0.90)	5070	(16.84)	653	(2.17)
fft	428	(1.00)	374	(0.87)	5299	(12.38)	627	(1.46)
nboyer	3092	(1.00)	844	(0.27)	5657	(1.83)	947	(0.31)
puzzle	593	(1.00)	380	(0.64)	5110	(8.62)	628	(1.06)
tak	274	(1.00)	238	(0.87)	5068	(18.50)	589	(2.15)
takl	292	(1.00)	240	(0.82)	5068	(17.36)	588	(2.01)
takr	563	(1.00)	388	(0.69)	5160	(9.17)	628	(1.12)
traverse	1828	(1.00)	628	(0.34)	5338	(2.92)	883	(0.48)

Table 3: Maximum Working Set in Virtual Pages with Ratio of Sizes Normalized to Cyclone Safe GC in Parentheses.

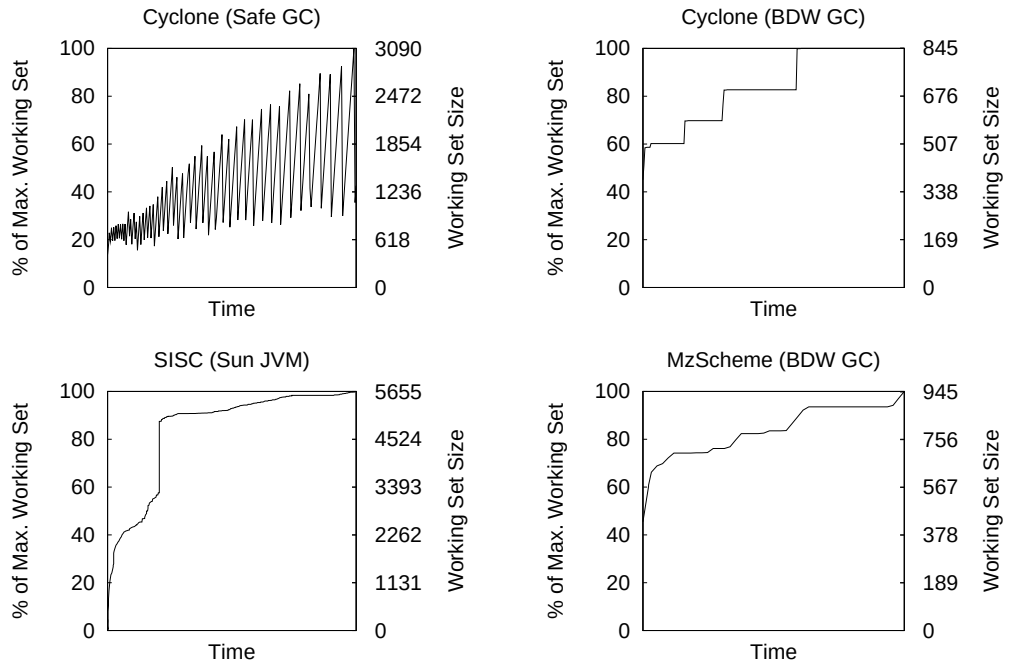

Figure 10: Working Set Over Time for `nboyer`

	Interpreter	Runtime System
Cyclone Safe GC	0	1800
Cyclone BDW GC	0	9000
SISC Sun JVM	0	229,100
MzScheme BDW GC	31,000	9000

Table 4: Approximate Number of Lines of Unsafe C Code.

sic region primitives needed for our safe GC is very small in comparison to other systems. The vast majority of the code (1200 lines) is associated with a fast implementation of `malloc`[13], on which the region primitives are built. In theory one could build the region primitives on a much simpler underlying storage system.

The BDW system is roughly 9000 lines of code. Despite its wide usage and relatively small size there are several known bugs that can cause the collector to fail in an unsafe way because of compiler optimizations or unsafe pointer manipulations [2]. Simple verification of the core 9000 line of code does not guarantee safety of the system so the line count is misleadingly on the low side. Arguably, one has to verify the entire system it is linked with and verify the client does not violate any invariants needed for the safe operation of the collector.

Appel and Wang [1] analyze the size of the safety TCB for several Java VMs. We quote their number used for the Hotspot JIT compiler, which is now the standard Java VM shipped by Sun. Of that 229,100 lines of code, the majority is the actual optimizing compiler. Since in a web application framework there needs to be a facility to dynamically extend the functionality of a running server without halting the system, running Java based servers must include a JIT and bytecode verifier in order to perform dynamic loading. So including the optimizing JIT compiler as part of the unsafe code of the system is reasonable. Other approaches can dynamically load native optimized code in a manner that does not require a heavy weight JIT [8].

MzScheme has a relatively compact implementation, but the total size of the system core is still on the order of 40,000 lines of code.

5. CONCLUSION

Compared to other systems, it is clear that our approach can significantly reduce the amount of unsafe code needed to implement a system. This increases our confidence in the safety and security of the entire system. Unfortunately, our copying-collection technique may incur a performance penalty for this extra degree of safety. Hopefully, in the future we can reduce this performance penalty so that the extra safety comes with little or no cost at all.

5.1 Cyclone vs. Java and Other Safe VMs

We should also emphasize that our approach potentially has some significant system-level performance advantages, when compared to a pure Java web application framework or any other safe virtual machine based approach. Typically, each web application will be allocated a separate thread of control to handle each web request. These threads are created frequently and have short lifetimes. In a pure Java environment there is no way to bound the allocation of a given Java thread or cheaply reclaim all the storage associated with a thread when it dies. We must rely on a system wide GC to efficiently reclaim storage for all our threads. The only other alternative is to spawn a heavy-weight sys-

tem process.

In a pure Cyclone system we can allocate a unique Scheme interpreter and garbage collector with its own private heap on a per thread basis. We can tune the initial and maximum heap sizes as well as the heap resizing policy on a per thread basis taking into account knowledge about the particular application servicing a request. More importantly, we can immediately reclaim all the storage associated with a terminated thread, without requiring a system wide GC. These per thread policies are simply not possible if one relies on a system wide GC. Exploiting these large scale system-level benefits is one area which we would like to explore in the future.

In general, if an application can benefit from a customized runtime system a pure Cyclone approach, allows the developer to customize and deploy a runtime system without compromising the safety of the system. This level of customization is not typically available in systems like Java.

Acknowledgements

We benefited from discussions with Greg Morrisett, who also contributed to the development of the Scheme interpreter and garbage collector.

6. REFERENCES

[1] A. W. Appel and D. C. Wang. JVM TCB: Measurements of the trusted computing base of Java virtual machines. Technical Report CS-TR-647-02, Princeton University, Apr. 2002.

[2] H.-J. Boehm. Simple garbage-collector safety. In *Proceedings of SIGPLAN'96 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 89–98. ACM Press, 1996.

[3] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software – Practice and Experience*, 18(9):807–820, 1988.

[4] *Chicken: A Scheme-to-C Compiler*, 2003. `http://www.call-with-current-continuation.org/chicken.html`.

[5] Cyclone user's manual. Technical Report 2001-1855, Department of Computer Science, Cornell University, Nov. 2001. Current version at `http://www.cs.cornell.edu/projects/cyclone/`.

[6] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 282–293, Berlin, Germany, June 2002.

[7] C. Hawblitzel, H. Huang, E. Krupski, and E. Wei. Low-level linear memory management. `http://www.cs.dartmouth.edu/~hawblitz/publish/linearmem-draft4.ps`, Dec. 2002.

[8] M. Hicks, J. T. Moore, and S. Nettles. Dynamic software updating. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI'01)*, June 2001.

[9] M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Safe and flexible memory management in Cyclone. Technical Report CS-TR-4514, University of Maryland Department of Computer Science, July 2003.

[10] *Jetty Web Server and Servlet Container*, 2003. `http://jetty.mortbay.org/`.

[11] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, Monterey, CA, June 2002.

[12] R. Kelsey, W. Clinger, and J. R. (Eds.). Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1), August 1998.

[13] D. Lea. A memory allocator. `http://gee.cs.oswego.edu/dl/html/malloc.html`.

[14] S. G. Miller. SISC: A complete scheme interpreter in java. `http://sisc.sourceforge.net/sisc.pdf`, 2003.

[15] S. Monnier, B. Saha, and Z. Shao. Principled scavenging. *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 81–91, 2001.

[16] *MzScheme*, 2003. `http://www.plt-scheme.org/software/mzscheme`.

[17] C. Queinnec. The influence of browsers on evaluators or, continuations to program web servers. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 23–33. ACM Press, 2000.

[18] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, Feb. 1997.

[19] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.

[20] D. Wang and A. Appel. Type-preserving garbage collectors. *Conference Record of the Twenty-Eigth Annual ACM Symposium on Principles of Programming Languages*, pages 166–178, 2001.

[21] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.