Toward a Feasible Functional Logic Compiler Employing Fair Search

A Dissertation Submitted to the Faculty of
Portland State University in
Partial Fulfillment of the Requirements for an
Honors Baccalaureate Degree in Computer Science

Marius Nita

June 10, 2005

Abstract

Traditionally, implementations for functional logic languages (FLP) have used depth-first search to implement narrowing, a well-known search strategy. While this approach yields good performance, it is unnecessarily restrictive in the sense that some programs lead to non-termination (the search computes forever in one part of the search space, failing to consider the rest) when clearly they should yield results. As a result, a number of recent FLP research efforts have focused on designing efficient FLP systems based on breadth-first search strategies, which hold an important *fairness* property: if a solution is known to exist, it will (eventually) be found.

The work presented in this paper addresses issues of efficiency in the context of FLC, a FLP compiler that employs fair search and a novel application of first-class stores in its implementation of narrowing. We show that in order to be tractable, FLC needs an efficient data representation for first-class stores and an efficient garbage collector. We discuss and compare several data representations for first-class stores, and show that a hashing-based data structure exhibits desirable performance. We show that proper collection of an efficient first-class store data representation in FLC involves additional work in the collector, to collect some reachable values which are known to be garbage. We describe the implementation of a complete generational copy collector that correctly handles collection of first-class store data structures. Finally, we evaluate the performance of the complete system, showing favorable results.

Contents

List of Figures

1 Introduction

Functional logic programming (FLP) is a paradigm which integrates features of both functional and logic programming into a unified model. It provides first-class functions and pattern matching, as well as logic variables and built-in search. Several functional logic languages exist, e.g. Curry [9], Escher [12], and Toy [13], of which Curry, the result of an effort to build a standard for FLP, is the most prominent. Current practical Curry implementations, e.g. Pakcs [8] and the Münster Curry Compiler (MCC) [14], use backtracking as a search strategy, which, as in PROLOG, has the undesired effect of causing non-termination in some situations in which a solution is known to exist.

To obviate this shortcoming, recent efforts, e.g. the FLVM [1] and the FLP interpreter by Tolmach, *et al.* (hereafter referred to as FLI) [19], have attempted to employ *fair* search strategies using breadth-first search in a concurrent setting. In a nutshell, each search alternative runs in a private thread, in parallel with all other alternatives. Consequently, each one is given fair and equal treatment, guaranteeing that no alternative is run indefinitely while others might lead to answers.

FLI not only implements a fair search strategy; it also features a novel application of *first-class stores* (Section 2.3) in its implementation of narrowing (Section 2.1). The FLI work left open questions with respect to whether the presented model can lead to an implementation that is as efficient as popular backtracking-based systems. In particular, it was unclear whether first-class stores would introduce an unsurmountable performance problem into low level implementations (compilers) following the FLI model.

The work presented in this dissertation begins where FLI left off and pursues these issues in the context of a *compiler*, named FLC, which follows the general FLI approach to fairness. Although this work does not attempt to give a definite answer to the question of whether this novel approach is tractable, it has yielded interesting results and we believe it is a step in the right direction. This paper presents the design, implementation, and evaluation of (a) an efficient low level representation for first-class stores, and (b) a complete garbage collector which is novel in its collection of first-class store data structures. As we will learn, it is necessary to adjust the runtime system's notion of liveness; in order to respect a set of desirable properties concerning space usage, we will explicitly kill off some data which is reachable but known to be garbage.

This document is structured as follows. Section 1.1 identifies the author's contributions to the FLC project and to the work presented here. Section 2 will introduce the reader to the research setting. Section 3 sets the stage for our low-level development by giving a more detailed technical explanation of the FLC runtime system. Section 4 discusses data representation issues; design constraints, approaches, and results. Section 5 describes the design and implementation of the garbage collector. Finally, we evaluate our completed system in Section 6 and conclude.

1.1 Contributions

The work presented in this paper was done in collaboration with, and under the close supervision of, my research advisor, Prof. Andrew Tolmach. He implemented the initial FLI system, the FLC compiler, and a large portion of the FLC runtime. The "references holding data" approach to first-class store representation (Section 4.2) and the space-related problems (Section 5.3) were identified by Andrew and presented in the original FLI work [19].

The author's contributions to the project have consisted of researching various data representations for first-class stores and identifying hashing-based data structures (Section 4.2.2) as a good speed-for-space tradeoff; implementing a complete garbage collector for FLC and augmenting it with code that solves the space problems identified in the FLI work; implementing other support structures in the runtime; evaluating the resulting system; and writing this document.

2 Background

2.1 Fair search strategies

Narrowing [7] is an evaluation strategy which provides a suitable foundation for implementing FLP systems which employ fair search. It combines term reduction (as in functional programming) with logic variable instantiation (as in logic programming). To understand how narrowing is applied, consider the following Curry program:

```
data State = Full | Half | Stopped

rate :: State -> Float
rate Full    = 1.0
rate Half    = 0.5
rate Stopped = 0.0
```

Suppose that we want to encode a function, named `anyRate`, which computes a rate that corresponds to a state, but does not care which state it corresponds to. In a conventional language, we might "fix" the state:

```
anyRate :: Float
anyRate = 0.5    -- Half
```

which would be "good enough." However, it does not match the specification, and as encoded above, and arguably, it is easy to write a buggy version, e.g.:

```
anyRate = 0.6
```

Curry allows us to encode `anyRate` such that it exhibits neither problem:

```
anyRate = rate x where x free
```

which can be read as "call `rate` with an argument which is valid, but whose value we do not care about." When evaluating this program, we say that it "narrows on `x`," since the value of `x` is needed despited the fact that `x` has no value (it is uninstantiated). The net effect of narrowing is to invoke `rate` with every possible value for `x` in an unspecified order. Every invocation, in this case, will independently contribute an answer to the final result of the program. Since order is insignificant, the final result of `anyRate` is the multiset {`0.0,0.5,1.0`}.

In general, the evaluation of a program may be visualized as a tree whose nodes are narrowing points and whose edges are linear evaluation sequences as in traditional programming. Successful evaluation of a program, then, boils down to performing a full search of this tree and collecting all possible results. One simple approach to doing this would be to search depth-first. In our example, we would first instantiate `x` to `Full`, say, and then evaluate `(rate Full)` all the way down to an answer. When done, we would return to the narrowing point, instantiate `x` to `Half`, evaluate `(rate Half)` all the way down to an answer, and so on. This approach has the obvious shortcoming that if we "get stuck" somewhere in the search tree, we won't have a chance to consider the rest. For example, if `(rate Full)` failed to terminate, we would not have a chance to consider `(rate Half)`, which *may* terminate and lead to a final answer.

A faithful implementation of narrowing will compute the possible search alternatives in a breadth-first manner, making sure that each alternative is given fair consideration without starving its siblings. Thus, even if `(rate Full)` might fail to terminate, we would pause its evaluation after its time slice had expired, and move on to evaluating `(rate Half)` for some period of time. Eventually, `(rate Half)` would lead to and report an answer. The idea is that even if one or more search alternatives computed indefinitely, we would still see all the answers resulting from the search alternatives that are known to terminate.

Currently, there are at least two known approaches to implementing narrowing using a fair search strategy. To illustrate them, consider the following algebraic type declaration:

```
data $T$ = $C_1(T_{11},\ldots,T_{1k_1})$
         | $C_1(T_{21},\ldots,T_{2k_2})$
         | ...
         | $C_n(T_{n1},\ldots,T_{nk_n})$
```

Now suppose that while evaluating a term t, the value of an uninstantiated logic variable x of type T is needed. Then the two approaches are as follows:

Reduction with term copying Separate terms t_1, t_2, …, t_n are created, where

$$t_i \;\equiv\; t[(C_i(x_{i1},\ldots,x_{ik_i}) \texttt{ where } x_{i1},\ldots,x_{ik_i} \texttt{ free})/x]$$

and the substitution is assumed capture-free. Then, n concurrent threads are spawned, where the ith thread proceeds with reducing the term t_i. The FLVM [1] takes this approach.

Fixed code with store copying This approach considers a functional logic computation to be the pair (t, s), where t is our term, represented as a pointer to *immutable code*, and s is a store: a mapping from variables in t to values. Separate computations (t, s_1), (t, s_2), …, (t, s_n) are created, where

$$s_i \equiv s[x_{i1} \mapsto \bullet]\ldots[x_{ik_i} \mapsto \bullet][x \mapsto C_i(x_{i1},\ldots,x_{ik_i})]$$

where a variable which is mapped to $\bullet$ is uninstantiated. The FLI interpreter follows this model.

The former approach has the advantage of being straightforward to implement and enjoys the property that reduction of sub-terms which do not contain logic variables can be shared across search alternatives. However, a significant amount of term copying still occurs at narrowing points and being reduction-based, it is naturally less efficient than compiled systems.

The latter approach avoids the overhead incurred by reduction, compiling each function into an immutable piece of code. It also moves the copying problem away from the term and into the store, thus opening the question of whether store copying can lead to overall more efficient implementations than term copying. It remains to be shown whether one approach is strictly superior to each other.

The work presented in this paper is concerned with ways to make store copying tractable in the FLC system.

2.2 The FLC system

A result of the ongoing FLP research effort at Portland State University, the FLC compiler is a natural extension of the FLI interpreter. The goal of the

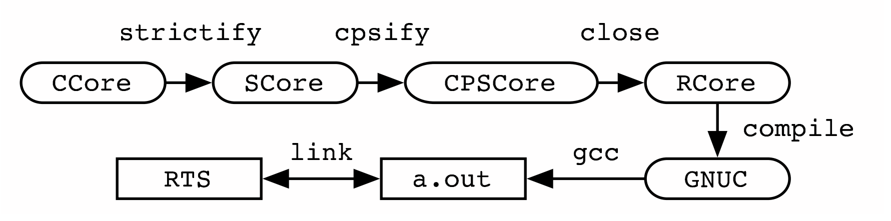

Figure 1: FLC: compilation pipeline

FLC project is to provide a tractable, efficient, and low-level implementation of functional logic programming. Unlike more widely accepted FLP language processors (e.g. MCC [14] or Pakcs [8]), FLC implements a fair search strategy. And unlike systems with a similar intent as its own (e.g. the FLVM [1]), FLC takes the *fixed code with store copying* (see Section 2.1) approach to implementing narrowing.

Figure 1 depicts the FLC compilation stages. Like the interpreter, its input is `CCore`, a high level, lazy FLP language. `CCore` is translated into `SCore`, a strict language with explicit operators for building and activating thunks. Following the compilation model described by Appel [3], `SCore` is translated into continuation passing style (`CPSCore`), closure converted (`RCore`), and compiled into a version of C which makes use of GNU extensions for reasons of practicality. Finally, the resulting code is compiled and linked against the runtime system, yielding a self-executable binary.

The FLC runtime system is composed of two main subsystems: memory and thread management. The memory management system consists of a memory allocator whose heap is presided over by a generational copy collector. Threads are needed to implement fair search, with each search alternative running in a separate thread. Threads are not truly parallel; the execution of threads is explicitly interleaved by a scheduling algorithm which resides in the FLC runtime, with each thread being given a fixed amount of CPU usage before it is preempted. Finally, as discussed in Section 2.1, each search alternative (corresponding to a thread in the runtime) is associated with its own variable store. FLC uses a formulation of first-class stores for this purpose.

2.3 First-class stores (FCS)

Generally speaking, the term *first-class store* (FCS) describes a language construct which allows treating stores as first-class citizens. The term *store* in this context typically refers to a *reference store*, as in Standard ML [4] and O'Caml [11]; but more abstractly, a store is simply a mapping from references to values. FCS systems typically allow several stores to coexist, and any given

```
signature FCS =
sig
  type 'a store
  type 'a sref
  exception BadIndex
  val new        : 'a -> 'a store
  val checkpoint : 'a store -> 'a store
  val update     : 'a store -> 'a sref -> 'a -> unit
  val allocate   : 'a store -> 'a -> 'a sref
  val deref      : 'a store -> 'a sref -> 'a
end
```

Figure 2: A Standard ML signature for first-class stores

```
structure Fcs : FCS =
struct
  type 'a sref = int
  type 'a st = ('a Array.array) * ('a sref ref)
  exception BadIndex

  fun new v = (Array.array(SIZE,v),ref 0)
  fun checkpoint (s,nextRef) =
    let val (ret as (newStore,nr)) = new (Array.sub (s,0))
    in  Array.copy {di = 0, dst = newStore, src = s};
        nr := !nextRef;
        ret
    end
  fun allocate (s,nextRef) v =
    let val next = !nextRef
    in  nextRef := !nextRef + 1;
        Array.update(s,next,v);
        next
    end
  fun update (s,_) r v =
    Array.update(s,r,v) handle Subscript => raise BadIndex
  fun deref (s,_) r =
    Array.sub(s,r) handle Subscript => raise BadIndex
end
```

Figure 3: A Standard ML implementation for first-class stores

reference can be successfully dereferenced in one or more stores.

Figure 2 presents a possible Standard ML signature for FCS and Figure 3 gives a very simple implementation which should capture all of the important invariants that implementations of first-class stores should respect.

The `new` operation generates a fresh store, unrelated to any other store in the program. The `allocate`, `deref`, and `update` operations correspond to the usual operations on store references. They correspond, for example, to the O'Caml `ref`, `(!)`, and `(:=)` operators, respectively. The main difference is that each FCS operator is also parameterized by the store in which to perform the respective operation, in addition to the reference itself. The `checkpoint` operation takes a store and yields a complete copy of that store. Subsequent operations in one store (either the copy or the original) are not to affect the other in any way.

The `checkpoint` operation is fundamentally important to most uses of FCS, and a prime motivator for the invention of FCS systems. Since stores are used to maintain state for some part of a program, copying a store effectively "freezes" program state, allowing it to later be restored. This pattern allows attractive solutions to problems such as undo/redo [6], replay debugging [20], demonic memories [22], nested transaction systems [15], and thread local storage.

Other formulations of FCS can be found in the GL programming language by Johnson and Duggan [10], and Morrisett's FCS extension for Standard ML of New Jersey [15].

2.4 Challenges

To give a better feel for how first-class stores are used in the runtime system, Figure 4 sketches a Standard ML implementation of the core narrowing routine, using an implementation for the FCS signature presented in Figure 2 that respects the first-class store properties we established in Section 2.3.

The implementation sketch follows the *fixed code with store copying* model that we established in Section 2.1. A type is represented as a sequence of constructors of fixed arity. Among the terms in our language we have `Undef`, denoting the *uninstantiated* state in a logic variable; `Var`, denoting a logic variable; and `Constr`, denoting a constructor. The `threadQueue` object represents the global computation queue, containing all the pending search alternatives (which are computations) at any given point.

The function `narrow` takes a computation (which as discussed in Section 2.1, is represented as a pair of a term and a store), a reference into the store, representing the variable that is being "narrowed on," and the type of that variable, and creates several concurrent computations, one for each constructor in the type. The `spawn` function spawns one computation given a corresponding constructor. First, a *copy* of the original store is made by invoking `checkpoint` (line

```
type name  = ...
type arity = int
type typ   = (name * arity) list     (* type *)
datatype term =
    ...
  | Undef                            (* uninstantiated *)
  | Var of (term Fcs.sref)           (* logic variable *)
  | Constr of name * term list       (* constructor *)

type comp = term * term Fcs.st       (* computation *)

(* global computation queue *)
val threadQueue : comp Queue.queue = Queue.mkQueue()

fun narrow (term,store) refn dType =
  let fun spawn (cName,cArity) =
    let
      val newStore = Fcs.checkpoint store
      val cArgs = List.tabulate
                   (cArity,
                    fn _ => Var(Fcs.allocate newStore Undef))
    in
      Fcs.update newStore refn (Constr(cName,cArgs));
      Queue.enqueue(threadQueue,(term,newStore))
    end
  in
    List.app spawn dType
  end
```

Figure 4: A Standard ML narrowing implementation

18). Then, new uninstantiated variables, corresponding to the the constructor's arguments, are allocated in the new store (lines 19-21). A new constructor value is created and used to update the original variable (that is being narrowed on) in the new store (line 23). Finally, the new computation is added to the global computation queue (line 24).

Whenever the value of an uninstantiated logic variable is demanded in a certain computation, this function is executed, causing the store at that computation to be `checkpoint`ed several times. Since this process is quite common in FLP, it is necessary that the `checkpoint` operation is implemented very efficiently. Moreover, as we will see, benchmarking in FLC has shown that the `deref` function is executed often and should be as efficient as possible. The problem is further exacerbated by the fact that choosing a data representation solely for efficiency purposes solves only part of the problem. Tolmach, *et al.* [19] observed that several garbage collection-related issues arise when using first-class stores to implement narrowing as explained in this section. To see why this is, briefly consider our first-class store representation in Figure 3. Suppose that, using this implementation, we ran a program that `allocate`d many objects in one store never needed to `checkpoint` that store. If running with a "normal" garbage collector, all the `allocate`d objects would remain live until the end of the program, despite the fact that the store references by which they could be accessed from the user program might have become garbage.

2.5 Summary

The implementation challenge boils down to finding an FCS data representation which will allow near constant-time operations on stores while remaining straightforward to garbage collect. Fortunately, there are ways to design a semantics-preserving representation for FCS that improve the situation significantly. The work of Driscoll, *et al.* [5] offer a good start toward inspecting efficient implementations in terms of *versioned data structures.*

3 Technical Details

The purpose of this section is to introduce the reader to the FLC system, in order to facilitate understanding the details of later sections. We begin by quickly introducing the compiler, and then give an overview of the runtime system's internal architecture.

3.1 The compiler

As mentioned in Section 2.2, FLC compiles a high level FLP core language (`CCore`) to GNU C. GCC is then invoked on the resulting code to produce an object file, which is then linked against the runtime system to obtain a standalone program. Reasons for choosing GNU C as our final target language included GCC's ubiquitousness and portability, but most importantly, we chose GCC due to a crucial non-standard extension: *first-class labels* [17]. First-class labels allow assigning labels (jump points) into variables, pass them into and out of functions, and generally treat them as any other program value. This allowed us to build a simple compiler from closure-converted CPS to C with the following general properties:

- The resulting program consists of only one C routine: `main()`.
- Each function is compiled into a sequence of C code preceded by a label.
- Arguments to functions are local variables in `main()`, named `_arg1` through `_arg10`.
- Calling a function with N arguments consists of (a) setting `_arg1` through `_arg`N to the appropriate argument values, and (b) jumping to the label corresponding to the function.
- Indirect calls (e.g. calling a function that is stored in a variable) are handled by the use of first-class labels.

This approach to compilation yields programs which use constant C stack (namely one frame for `main()`) and no extra work has to be done to enforce this property.

3.2 The thread model

As briefly mentioned in Section 2.2, FLC implements a simple scheduler which explicitly interleaves the execution of search alternatives, which we also refer to as "threads." Figure 5 shows the basic thread structure in FLC and some important global objects pertaining to the thread system. A thread, as discussed in previous section, is composed of executable code (the `codeptr` field) and a first-class store mapping variables in that code to values. The `codeptr` field is a suspended closure: a closure which is stored together with the argument it is to be applied to. The details of code representation are beyond the scope of this paper, however, and will not be considered any further.

```
typedef struct Thread {
  Fcs store;
  Code codeptr;
} * Thread;

Thread thread_current;      /* currently executing thread */
Queue  thread_queue;        /* pending threads */
```

Figure 5: FLC: thread model

The thread_current global contains the currently executing computation and the thread_queue contains all pending threads, i.e., all threads in the program except for the currently executing thread. To schedule the next thread, the scheduler enqueues thread_current onto thread_queue, sets thread_current to the first element on thread_queue, and begins executing it. When a thread fails or is done computing, it is dropped (it will not be enqueued again); the next thread is then dequeued and executed.

3.3 First-class store interface

FLC uses first-class stores in its implementation of narrowing, as discussed in section 2.2. The first-class store implementation in FCS is hidden behind an interface which and the runtime system and mutator guarantee not to violate, meaning that the runtime system and mutator are unaware of the underlying data representation. (In fact, the mutator is disallowed, by design, to hold a direct pointer to a first-class store object.) Figure 6 shows the basic C-level interface for first-class stores, which corresponds directly to the SML signature in Figure 2. Since, in FCS, there is no reason for a dereference operation to fail legitimately, there is no error return from fcs_deref; failure halts the program, as it is proof of a bug.

FLC maintains two important FCS-related invariants, which we will be exploiting later. First, the FCS implementation is completely shielded by the interface; no part of the system accesses the internal data representation, except, of course, for the garbage collector. And second, the running program is not allowed, by design, to hold a direct pointer to a first-class store object. First-class store objects are pointed to strictly by Thread objects, which are all maintained by the runtime system, and to which the running program is agnostic. Moreover, the only store object that can be "acted upon" at any point, whether by the running program or RTS itself, is the store in the current thread. This is enforced by exposing only a specialized interface to the program, which fixes the

```
typedef Fcs ...
typedef SRef ...

Fcs   fcs_new(void);
Fcs   fcs_checkpoint(Fcs);
SRef  fcs_allocate(Fcs, Value);
void  fcs_update(Fcs, SRef, Value);
Value fcs_deref(Fcs, SRef);
```

Figure 6: FLC: first-class store interface

store argument to the store belonging to the current thread. E.g.,

```
#define DEREF(r) fcs_deref(thread_current->store,(r))
#define UPDATE(r,v) fcs_update(thread_current->store,(r),(v))
...
```

4 Data Representation

In this section, we explore issues related to discovering and implementing a good data representation for first-class stores in FLC. A key point to note right from the start is that what might be a tractable implementation elsewhere, or a good *general* implementation, could prove to be completely inadequate in FLC. To understand why this is, we will state and motivate two major design constraints to guide the rest of our exploration. The data structure we design must admit

1. a very efficient implementation of `deref`

2. easy, efficient, and correct garbage collection

While the latter constraint appears quite superfluous at first glance, we will see that, with all tradeoffs considered, it becomes of central concern; in some cases we will need to "enhance" the system's notion of liveness to gain acceptable runtime behavior.

To shed some light on the first constraint, consider the data in Figure 7. Using a straightforward FCS implementation[1], we recorded runtime data for four programs (see Appendix A) and determined that `deref`, `update`, and `allocate` are the three FCS operations in highest demand. The table shows percentage of calls to FCS operations over the runs of the four programs. The first four columns show demand of each operation, as a percentage of the total number of

[1]The implementation we used in this test is discussed in Section 4.2.1

	`deref`	`update`	`allocate`	`checkpoint`	FCS Overall
`ndtest 7`	79.07	9.36	11.22	0.03	12.38
`perms 7`	75.35	13.29	11.29	0.03	12.86
`nat 400 400`	70.78	17.04	12.17	—	10.68
`fibo 27`	77.74	13.00	9.24	0.00	11.46

Figure 7: FLC: first-class store usage (percentages)

calls to the FCS interface. The last column shows percentage of calls for *all* FCS operations combined, with respect to the total number of calls in the runtime. A value of "—" denotes a number that was too low to measure accurately. The data points out two major important factors to consider when designing a FCS data representation. First, the `deref` operation is in substantially higher demand than its peers, indicating that we should give it special consideration; perhaps, at the expense of slowing down other FCS operations. Second, the FCS implementation is in high demand in the runtime in general, meaning that changes in FCS performance have a large impact on general FLC performance. We should therefore look to make our first-class store representation very efficient.

To fully capture the essence of the second constraint, we will be formulating a number of properties and definitions which will guide our approach to system design and implementation in later sections.

Property (Stores One-to-one Correspondence): *There is a one-to-one correspondence between threads and stores.* There are two points in the RTS which yield new threads: (a) system initialization, when the program is prepared to be run, and (b) narrowing in an existing thread. In (a), the new thread is given a store which is acquired via the `new` operation and so the thread and store have a one-to-one correspondence. In (b), the store is acquired via `checkpoint`, which we know to produce an independent copy of a current store, so the thread and store are in one-to-one correspondence again. □

An important consequence of the *One-to-one Correspondence* property is the fact that when a thread becomes unreachable, so does its corresponding store. We can further elaborate this fact into another important property:

Property (Stores Correct Collection): *Once a thread becomes garbage, its corresponding store and all the values that are reachable <u>only</u> through its corresponding store will immediately become garbage.* □

One may think that the latter property is superfluous; that when a store becomes unreachable, those values that are reachable only from that store should necessarily follow. However, we intend to be very clever in data structure design; for example, it may be desirable to avoid keeping values in the store altogether, and keep them in other, related, data structures.

Now that we have established some properties for correct store collection, we turn to the more complex issue of store reference collection. When a reference becomes unreachable, we should not have to wait for its corresponding store to become garbage before the value "indexed" by that reference becomes garbage. In fact, the issue is much more complicated: there may be *several* stores, perhaps a very large number of them, in which the reference can be looked up. We arrive to the following property:

Property (Reference Garbage): *When a store reference becomes garbage, the data that it points to (in all the stores that it can be dereferenced in), provided it is not reachable from elsewhere, also becomes garbage.* □

Earlier, we have established that `checkpoint` is directly associated with narrowing, and stores are in a one-to-one relationship with threads. The thread that narrowing occurs in is replicated a number of times, and its corresponding store is `checkpoint`ed accordingly. After this process, however, the thread is *discarded*, and so is its corresponding store! We can then state the following property:

Property (Store Liveness): *Upon narrowing, the thread that narrowing occurs in immediately becomes garbage, and so does its corresponding store. Therefore, subsequent operations in that store are illegal.* □

The FLC runtime respects this property, but it is not explicitly enforced in the store interface or implementations. We state one final set of invariants which will play an important role in the rest of the paper:

Property (Store Reachability): *The mutator is disallowed by design to hold references to stores. All the stores in the program can be found in (a) the current thread, and (b) threads on the global queue.*

Definition (Checkpoint Path): *We say that there is a checkpoint path between two stores H_c and H_p if either (a) H_c was* `checkpoint`*ed from H_p, or (b) H_c was* `checkpoint`*ed from H'_p and there is a checkpoint path between H'_p and H_p.* □

Property (Reference One-to-many Correspondence): *A reference p can be dereferenced in any number of stores $H_1, \ldots, H_n$ as long as there is a checkpoint path between H_i and H, where H is the store in which p was initially allocated.* □

In the rest of this section, we will consider various approaches to data representation for first-class stores. In Section 4.1, we will consider a "traditional" approach to stores, where stores are data structures holding data and references are indices into these structures. Section 4.2 reverses the roles of stores and references; data is stored in references and stores are indices into references. Throughout the design process, we will make sure that every approach strictly respects the properties established above.

4.1 Stores holding data

Traditionally, stores have been implemented as straightforward mappings from "keys" to "values" (or references to data). A trivial implementation might represent the store as an array of data and references as integer offsets into the array. Fancier implementation might represent stores as self-balancing trees or fine-tuned hash tables. In this section, we explore two representations for first-class stores, where the data is held in the store data structure. First, we briefly consider a hypothetical red-black tree implementation and then an array-based implementation.

4.1.1 Red-black trees

It is conceivable that we could represent stores as red-black trees (or any functional data structure for dictionaries), and references as any suitable key type. This approach seems promising with respect to our first constraint: searching a store is order $O(\log n)$. Although we are yet unsure whether this is sufficient performance for our goals, there are more important issues to consider. For example, `checkpoint` might become unduly expensive if stores are sufficiently large on average.

Leveraging ideas from the world of purely functional data structures [16], we can attempt to devise a tractable `checkpoint`ing mechanism. Instead of building a complete duplicate of a store, we instead do no work at all and return a pointer to the tree. Upon `update`, we copy the tree nodes on the spine leading to the node containing the value in question, but maintain the rest of the tree as-is. We do the same thing upon `allocate`. If `allocate` induces a rotation, we also have to copy all nodes which change their position relative to the root. In a worst-case scenario, a rotation near the top of the tree might cause the entire tree to be copied... Note that while it may seem that we are violating the *One-to-one Correspondence* property, this is actually not the case. Two threads can point to the same store as long as they are ready to copy it upon mutation.

Overall, since we make copies of everything that we "mutate" and never assign to existing data structures, we maintain a non-interference property between any two stores related by a fork path. This approach, however, has one irreconcilable complication with respect to our garbage collection-related constraints. If the program being run contains no non-determinism, and hence no narrowing, the entire execution will occur in the context of a single store. Presumably data will repeatedly be allocated in this store, but will it ever be freed? Without additional work in the collector, *every* allocated value will remain live until the end of the program; a clear violation of the *Reference Garbage* property. This is certainly unacceptable, but is there something we can do to fix it?

We could "notice" when references become garbage and try to manually force their corresponding values to garbage during collection. This would involve a large amount of bookkeeping; we would have to actively maintain a mapping from references to lists of stores, updated during allocation. During collection, we would identify the garbage references, look them up in our map, track down their corresponding stores, and remove their corresponding values from each store.

This approach is infeasible, however, due to the fact that identifying all the garbage references at collection time *before* collecting all the stores is impossible. Since references are first-class citizens, they can certainly be stored as values in stores, meaning that identifying all the live references involves scanning all the live data. Due to its several complications, this approach will not be considered further.

4.1.2 Contiguous memory regions

Having given up on our fancy red-black tree data structure, we attempt a more "bare-metal" approach: we represent stores as arrays and references as integer offsets. An encoding in FLC might look like the following[2]:

```
typedef unsigned int SRef;
typedef struct {
  size_t size;
  size_t cursor;
  Value data[1];
} * Fcs;
```

The purpose of `size` is to keep track of the overall size of the number of *possible* elements in the `data` array, and `cursor` holds the next free index. When `(size==cursor)` becomes true, the store is resized. To accommodate for the fact that the allocation routine may return a new (resized) store, we need a slight change to the store interface:

```
SRef fcs_allocate(Fcs*, Value);
```

The advantage of this scheme is extremely good (constant-time) `deref` and `update` performance. These operations can be implemented as trivial macros:

```
#define fcs_deref(h,i) ((h)->data[i])
#define fcs_update(h,i,v) {(h)->data[i] = (v);}
```

[2]`Value` is used as a typedef for `void*`

Checkpointing, however, is slightly more complex. Unlike in the red-black tree case, we do not have the luxury of partial copying; we must copy the entire store. Since `checkpoint`ing is quite infrequent and copying contiguous memory regions is a very straightforward, fast operation (one allocation plus one `memcpy()`), so speed is not an issue. We should, however, be concerned with potential space usage issues. Brief testing showed that stores make up more than half of the live data at any point; a very worrisome figure.

One may think that a sort of copy-on-write policy can be deployed here, where stores are not copied immediately upon `checkpoint`, and the copying could be delayed until the next `update` or `allocate` operation. This could certainly be done, but it turns out it is a quite useless optimization, since the general pattern of narrowing entails a `checkpoint` immediately followed by an `update` in the new store.

Contiguous memory regions look quite good performance-wise, but they behave just as badly as red-black trees when it comes to collection. They suffer from the same reference liveness detection problem: all the live data must be scanned before we can determine exactly which references are garbage. Furthermore, explicitly deleting garbage values results in sparse stores, which can cause large space leaks unless we deploy a more complex allocation algorithm which can efficiently reuse indices. Finally, due to its several complications, we decide not consider this approach any further.

4.1.3 Summary

The general patterns that have surfaced in our exploration of the "stores holding data" approach are that the operations that we are concerned with can be made very fast and we get the *Stores Correct Collection* property for free, but significant complications are encountered when factoring in garbage collection. Namely the *Reference Garbage* property does not follow naturally, and significant (intractable) machinery must be exercised in order to correct this shortcoming.

4.2 References holding data

In previous work [19], we presented an approach to first-class store representation that was very different than the "traditional" view which we discussed above. Instead of storing data in the store data structure, we store it in the reference itself! While in the previous approaches we considered stores to be collections of values, each value "tagged" by a reference, here we reverse the roles of store and reference, and consider references to be collections of values, where each value is tagged by the store it was allocated in. Essentially, references become versioned pointers, or "fat nodes," as discussed by Driscoll *et al.* [5].

This approach leads to an important issue: if all the data is stored in references, (a) how do we represent stores, and (b) how do we implement the `checkpoint` operation? As explained earlier, stores are now simply tags which are used to look up a value in a reference. Therefore, to answer part (a), stores need not be anything more than generated values, such as integers or reference cells. Part (b) is a bit more complicated. An extremely naive approach would be to, when given a store to checkpoint, do something of the form:

```
checkpoint($h$) $\equiv$
  $h'$ = generate new store number
  for every reference $p$
    if ($h$,$v$) exists in $p$ then
      insert into $p$ ($h'$,$v$)
```

where (h, v) denotes a tagged value: v is the value and h is the store tag. If we were to take this approach, the following property should hold:

Property(Correct Dereference): If a reference p can be dereferenced in a store h then a value tagged by h must exist in p. Therefore, dereferencing p in h is a matter of searching p's contents. □

The approach is, however, intractable. Not only would we have to maintain a side list of all the references in the program, but the `checkpoint` operation would be order $O(pq)$, where p is the total number of references in the program and q is the average number of values per reference.

Fortunately, we can take advantage of an important FCS invariant to improve the situation significantly. The *Store Liveness* property states that threads become unreachable after being narrowed in, and so do their corresponding stores. Since `checkpoint`ing occurs exactly at narrowing points, it is always the case that after narrowing, the `checkpoint`ed store becomes immediately unreachable, and so we are guaranteed that no further operations can be performed on it.

We can exploit this property to represent stores as sequences of tags which encode "`checkpoint` histories." In a sequence $h_0, h_1, \ldots, h_n$, h_0 is the tag of the store in question, h_1 is the tag of the store that h_0 was `checkpoint`ed from, and so on. In any `checkpoint` history, the last element is the tag of a store which was acquired via the `new` operation. The general idea is that by representing stores as `checkpoint` histories, for any given store we can very quickly access its "ancestors." This allows us to have a light `checkpoint` operation, which does not modify the contents of any store references. Then, when looking up a reference in a store that was the result of a `checkpoint`, we can look up that reference in each of the "ancestors" until we succeed. We need not worry that "ancestors" might be modified in another part of the program and so we might end up looking up bogus values, because, as established above, any store that is an "ancestor" (i.e, `checkpoint` has been invoked on it) is unreachable from the program.

The first obvious consequence of this approach is that the `checkpoint` operation is constant time:

```
checkpoint($hseq$) $\equiv$
  $h'$ = generate new store number
  return ($h' :: hseq$)
```

where $(a :: b)$ (pronounced "a cons b") is used to represent a sequence whose first element is a and whose "tail" (the rest of the sequence) is the sequence b. The second consequence is that the *Correct Dereference* property, as stated in the context of the previous approach, no longer holds. We have to restate it:

Property(Correct Dereference'): If a reference p can be dereferenced in a store $h :: hs$, then either a value tagged by h exists in p, or p can be dereferenced in hs. □

With this in mind, we can sketch the `deref` operation as follows:

```
deref($store$,$p$) $\equiv$
  if $store$ is the empty sequence then error
  else
    $store \equiv h :: hseq$
    if ($h$,$v$) exists in $p$ then return $v$
    else deref($hseq$,$p$)
```

`deref` is now order $O(n+k)$, where n is the maximum length of a store sequence and $k = \sum_1^n k_i$, k_i being the cost of determining whether a reference contains a value tagged by a given store. As we will see, k_i can be easily reduced to the number of values in a reference, or even amortized to a constant.

The key advantage that the *references holding data* approach has over the previously considered *stores holding data* approach (Section 4.1) is the fact that it is significantly easier to respect the properties we established at the outset of this section. The *Reference Garbage* property now is implicitly respected; we do not need to do additional work to enforce it. On the other hand, the *Stores Correct Collection* property no longer holds naturally, but it is relatively easy to enforce. Notably, no bookkeeping is necessary during mutation, and the overhead added to the collection phase is negligible. We can notice at collection time when stores become garbage; then, when scanning individual values in a reference, we can determine very quickly whether their corresponding stores are garbage, and purposely fail to mark them as live. The details involved in this process will be explained in Section 5, but for now suffice it to say that we believe the *references holding data* approach has numerous benefits, leads to good implementations, and we will favor it over our previous attempts.

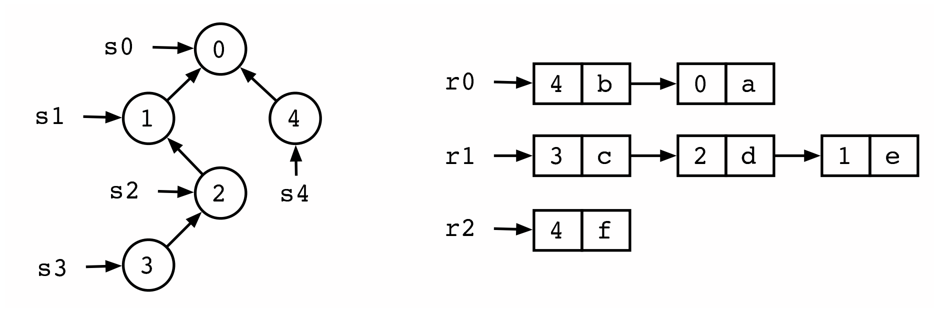

Figure 8: Linked list FCS representation

4.2.1 Linked lists

We begin by considering an obvious approach to implementing the *references holding data* approach: we represent references as linked lists of (store tag, value) pairs, stores as linked lists of store tags, and store tags as integers. Data declarations for this approach in FLC might be as follows:

```
typedef struct SRef *SRef;
struct SRef {
  int store_tag;
  Value value;                      /* tagged value */
  SRef next;
};
typedef struct Fcs *Fcs;
struct Fcs {
  int tag;                          /* store tag */
  Fcs next;                         /* checkpoint history */
};
```

A store and store reference are both linked lists. The former is a sequence of store tags (the checkpoint history) and the latter is a sequence of tagged values. To visualize how this approach works in general, consider Figure 8, which shows the state of five stores and three references after the following sequence of commands:

```
Fcs s0,s1,s2,s4,s4; SRef r0,r1,r2;
s0 = fcs_new();
r0 = fcs_alloc(s0,(Value)'a');
s1 = fcs_checkpoint(s0);
r1 = fcs_alloc(s1,(Value)'e');
s2 = fcs_checkpoint(s1);
fcs_update(s2,r1,(Value)'d');
```

```
s3 = fcs_checkpoint(s2);
fcs_update(s3,r1,(Value)'c');
s4 = fcs_checkpoint(s0);
fcs_update(s4,r0,(Value)'b');
r2 = fcs_alloc(s4,(Value)'f');
```

As can be seen, though each store is a linked lists of tags, the entire set of stores forms a reversed tree in memory. The tree is an accurate account of `checkpoint` activity; it is evident which stores are the descendents of which stores.

We have established that in order for a store representation to be viable, it has to provide good `deref` performance, and it has to admit an efficient garbage collection implementation while respecting the properties stated at the beginning of this section. To start addressing the former, we might try a very straightforward implementation for `deref`, where a linear search is performed in the reference for each tag number:

```
Value fcs_deref(Fcs st, SRef ref) {
  SRef r;
  for (; st; st = st->next)
    for (r = ref; r; r = r->next)
      if (r->store_tag == st->tag)
        return r->value;
}
```

This implementation runs in $O(nk)$ time, where n is the maximum length of a store and k is the length of a store reference. This is quite bad, given our goals. We can quickly improve the situation by simply keeping data in references sorted by store tag, in reverse order. Since stores are also reverse-ordered sequences of store tags, we can iterate the two sequences in a concurrent manner:

```
Value fcs_deref(Fcs st, SRef ref) {
  while (st && ref) {
    if (st->tag == ref->store_tag)
      return ref->value;
    else if (st->tag > ref->store_tag)
      st = st->next;
    else
      ref = ref->next;
  }
}
```

The new implementation runs in $O(n + k)$, a clear improvement over the former. Performance of the other store operations is not affected by our change.

`checkpoint` and `alloc` remain unchanged; `update` still has to perform a linear search ($O(k)$).

With respect to garbage collection, we established earlier in this section that the *Reference Garbage* property holds trivially. When a reference becomes unreachable, so do all of its nodes. If some values are reachable only from the reference, they will naturally become garbage. The *Stores Correct Collection* does not immediately hold, however. A store (which is a list of tags) can naturally become unreachable when its corresponding thread becomes unreachable, but several live references may hold values tagged by that particular store, values which, without additional work, remain unnecessarily live. We can fix this problem by using a collection algorithm which roughly performs the following steps:

1. Walk the thread queue and mark all stores on the queue as live. (See the *Stores Reachability* property).

2. Perform a normal GC; however, when encountering a store reference node, if its corresponding store has not been marked live, re-link the reference list around it.

The latter may be done with light bookkeeping, by creating a list of all the live stores during step (1). We can, however, avoid all bookkeeping by replacing the store tag in each reference node with a store pointer. This would involve a slight change to our store implementation to do pointer comparison instead of tag comparison, but it would allow us to very quickly identify whether the store associated with a tagged value in a store reference was live.

4.2.2 Hash tables

Another data representation for the "references holding values" approach might involve using a hash table [18] instead of a sorted linked list to represent a store reference. The hash table would give amortized constant time to determine whether a value exists in the reference for a given tag, so `deref` would run in $O(n)$ where n is the length of the store (`checkpoint` history). An added bonus of a hash table representation is that `update` becomes constant-time. With a proper implementation, we should expect to see a big speed increase over the linked list approach, at the expense of space.

During the evaluation of the hash table approach, we discovered that the `deref` operation is indeed an extremely important variable in system performance. So much so that a simple tweak of the hashing function could result in very notable differences in performance. Therefore, we were faced with a set of important tradeoffs. First we had to pick the right hash function; better hash functions are more expensive but lead to better data distribution and fewer collisions, and simpler hash functions run very fast but may lead to poor distribution and a high number of collisions. Second, we had to decide whether we should try to shrink tables when occupancy became very low. Tables whose size is a prime number offer overall good behavior under appropriate hash functions [18], but shrinking such a table is an expense we cannot afford: one would have to rehash every value in the table to its new location. On the other hand, tables whose size is a power of 2 (assuming a chaining implementation) can be shrunk down very efficiently, via an algorithm linear in the table size.

We attempted several hash table implementations, using both open addressing and chaining. In each, the hash function turned out to be a major performance variable. In fact, anything more complex than a *mod* operation would easily make `deref` the most expensive operation in the system. In the end, a power-of-2 hash table with chaining and a *mod* function turned out to be the best performer. In addition to being relatively easy to garbage collect, the power-of-2 table admits a very fast *mod* operation, since the following property holds when *size* is a power of 2:

$$\texttt{mod}(key\texttt{,}size) \equiv key \mathbin{\texttt{\&}} (size\texttt{-1})$$

where `(&)` is the logical *and* operator.

Garbage collection issues for chained hash tables remain relatively the same as those for the linked list approach (Section 4.2.1). Instead of fixing a single linked list per store reference during collection, we now work with an array of linked lists (buckets). For each bucket, we proceed to force dead values to garbage in a way identical to the linked list approach.

4.3 Comparisons

In this section, we compare the relative performance of the representations we have considered so far. We have gathered data for five implementations on four programs. The five implementations are as follows:

- *Hash open addressing*: A store reference is a prime sized table which uses open addressing for collision resolution.
- *Hash chaining (prime size)*: A prime sized table which uses chaining for collision resolution. The table is an array of linked lists, each list holding values which hashed to that same "bucket."
- *Hash chaining (power of 2 size)*: Same as above, but uses power of 2 sizes.
- *Linked list*: A store reference is a linear linked list of store-tagged values.

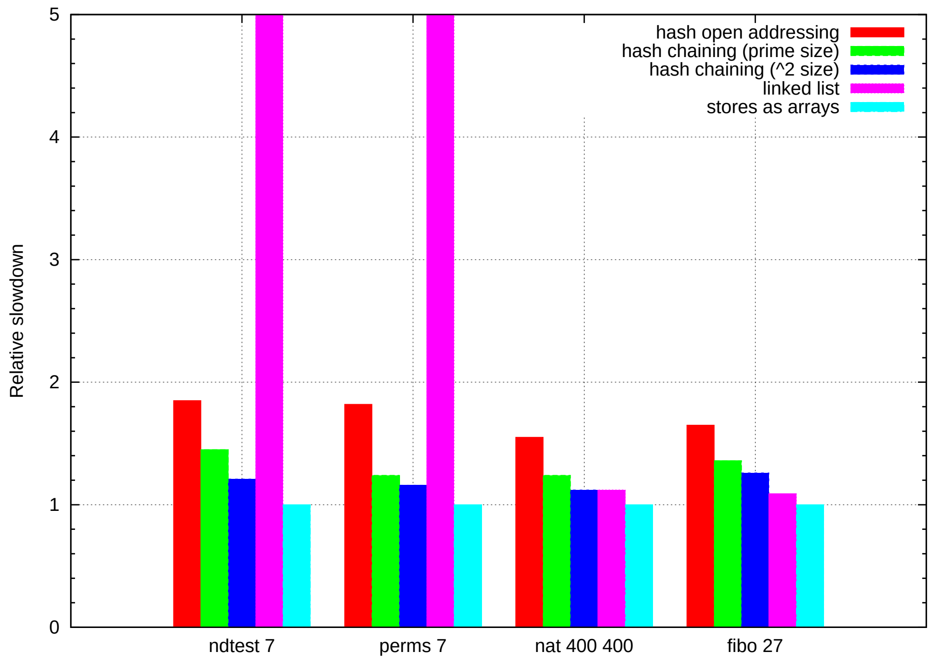

Figure 9: FCS comparison: relative execution time slowdown

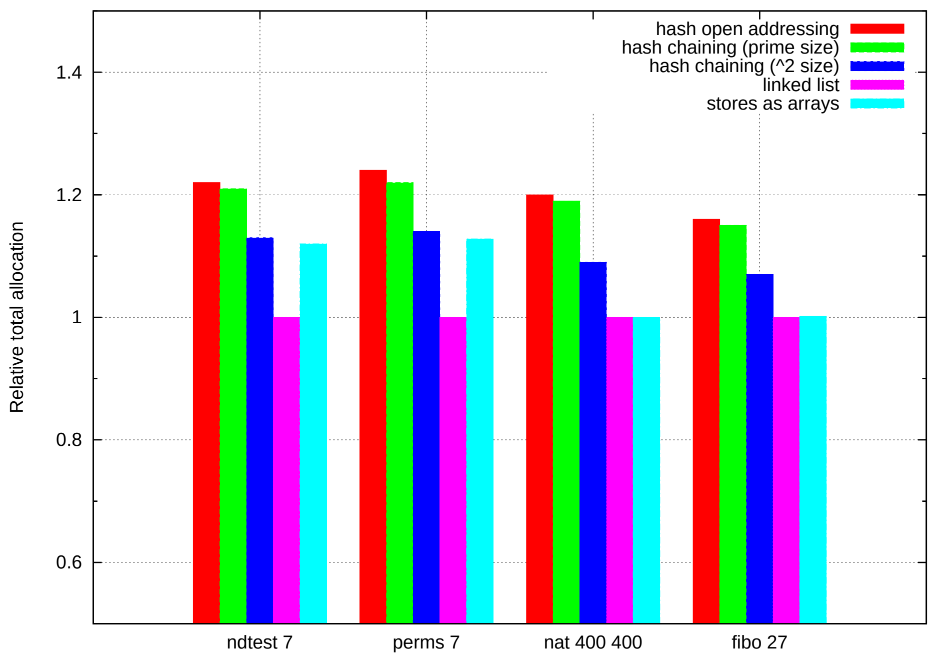

Figure 10: FCS comparison: relative total allocation

- *Stores as arrays*: A "stores hold data" implementation, where stores are arrays and store references are integer offsets, as discussed in Section 4.1.2.

The "stores as arrays" implementation was included in the comparison due to its excellent runtime performance; it provides a very good normalization point for speed, especially in `deref`-intensive programs such as `ndtest`.

Figure 9 shows relative execution time slowdown for the five implementations, normalized around *stores as arrays*. The first thing to note is that the linked list implementation is slightly over 16 times slower than the normal, which is not evident in the graph in order to allow discerning between the other implementations. The *power of 2* implementation is impressive, only slightly slower than the normal in all tests, with *prime size* following closely. Note that the linked list implementation, though proving to be inadequate in general, performs well in the latter two tests. This is because they are not very `deref`-intensive, and perform more allocation and checkpointing.

Figure 10 shows relative total memory allocation for the five implementations. The numbers are normalized around the linked list implementation, which is "optimal:" every allocated word is immediately needed. The relative differences between the implementations are much smaller than in the timing test, but once again, we can see that the *power of 2* implementation is a good performer. Our speed-for-space tradeoff is certainly reasonable.

Although the *stores as arrays* approach is very attractive, we will not be considering it further due to severe garbage collection complications, as discussed in Section 4.1. Whether this approach could be made feasible with clever enough bookkeeping is an issue left for future work.

4.4 Conclusion

We have investigated various approaches to data design for first-class stores in FLC. The unique design constraints that FLC imposes on our design process led to a data representation that is quite esoteric: stores are linked lists of store "tags" (unique numbers), representing checkpoint histories, and store references are hash tables of values, keyed on store tag.

We favor the *power of 2* implementation over the others in the rest of the document due to its good overall performance and the fact that we believe it can be garbage collected fairly easily. The next section describes the rest of a complete, properly collected system which uses power-of-2-sized hash tables to represent store references.

5 Garbage Collection

Having decided on a data representation for first-class stores, we now present the design and implementation of a garbage collector that, besides working to provide decent performance, enforces the properties we established in Section 4.

5.1 Design requirements

Like functional programs in a typical functional language processor, functional logic programs in FLC involve a tremendous amount of implicit allocation. Furthermore, FLC's source language is *lazy*, meaning that every function invocation causes its arguments to be suspended in "thunks," which are also allocated in the heap. Finally, FLC programs are concurrent, which involves added heap allocation upon context switches. Therefore, the allocation routine must be very fast.

Another important property of functional and functional logic programs is that "most data die young." A quick survey in FLC shows that 93% of closures are garbage by the time a collection phase should kick in. On average over the four programs considered in the previous section, approximately 78% of all data is garbage by the time a collection should take place. We would like our collector to avoid collecting prematurely; to avoid discovering a large amount of live data right before it becomes unreachable. Collection should be delayed as much as possible, to give data a good opportunity to become garbage before it is considered.

A moving collector (e.g. copying, mark and compact [21]) satisfies the former requirement: memory is allocated in a contiguous free region, so the allocation routine consists of simply incrementing a pointer. The latter requirement is typically satisfied by a generational collector. Instead of considering the entire heap in one monolithic collection phase, we have two types of collections:

- *Minor*: Frequent cycle which collects data in a small part of the heap dedicated to allocation (a.k.a. nursery)
- *Major*: Much less frequent cycle which collects a larger part of the heap. In a two-generation collector, the major collects the entire heap.

The general idea is that the minor cycle "delays" a full heap collection by collecting a little at a time. When the major kicks in, the overwhelming majority of the data is expected to be garbage.

5.2 The basic collector

Given our design constraints, we decided to implement the generational copy collector described by Appel [2]. The collector maintains two generations: *newer* (or nursery) and *older*. The newer generation contains data allocated since the last minor cycle, and it is the memory area that the minor cycle operates on. The minor cycle copies all the live data in *newer* to to the end of *older*. Similarly, the older generation contains data copied into it by the minor cycle since the last major cycle, and it is the memory area that the major cycle operates on.

The collector makes one fundamental guarantee: *as long as the live data is equal to or less than half the size of the heap, the collector will not run out of memory.* Why do we need to make this guarantee? First of all, a copy collector needs auxiliary free space to copy live data into at collection time. In a typical two-space copy collector [21], the heap is divided exactly in two equal regions: the allocation space and the auxiliary space. At collection time, the live data in the allocation space is copied into the auxiliary space, and the roles of the two spaces are reversed. The reason that the auxiliary space is equal in size to the allocation space is that in case the entire allocation space is almost full with live data, we would like the collection to succeed. We would like to make a guarantee in our generational collector that is at least as strong as that of the two-space one. In a nutshell, we maintain that if the size of the live data is less than half of the heap size when a major cycle kicks in, the major will necessarily succeed.

To give a better idea of how the collector works, we show various heap states during different phases of mutation and collection in Figure 11. Initially, in part (a), the heap is divided exactly in two equal regions: *older* and *newer*. As the mutator invokes the allocation routine, memory (gray area) is allocated in *newer* until it fills up, as shown in part (b). Whenever *newer* fills up, a minor cycle collects the *newer* space, moving the live data to the end of *older*, as shown in parts (b) and (c).

After some period of time has elapsed and several minor cycles have occurred, the heap is brought in a state similar to one illustrated in part (d). The older generation is very close to the middle of the heap and the newer generation is full. First, a minor cycle occurs as usual, collecting data in *newer* to the end of older, as shown in (d) and (e). If, during a minor, the older generation becomes larger than half of the heap, a major collection is invoked. A major is made up of two phases. First, all the data in *older*, but *ignoring* the part of *older* that was the result of the immediately preceding minor, is collected into the empty space to the right of *older*, as shown in (e) and (f). Second, all the live data (all shades of gray) is *moved* to the beginning of the heap, as shown in (f) and (g). The *newer* pointer is then reset and mutation resumes.

There are a few subtle points in this process. For example, how does the

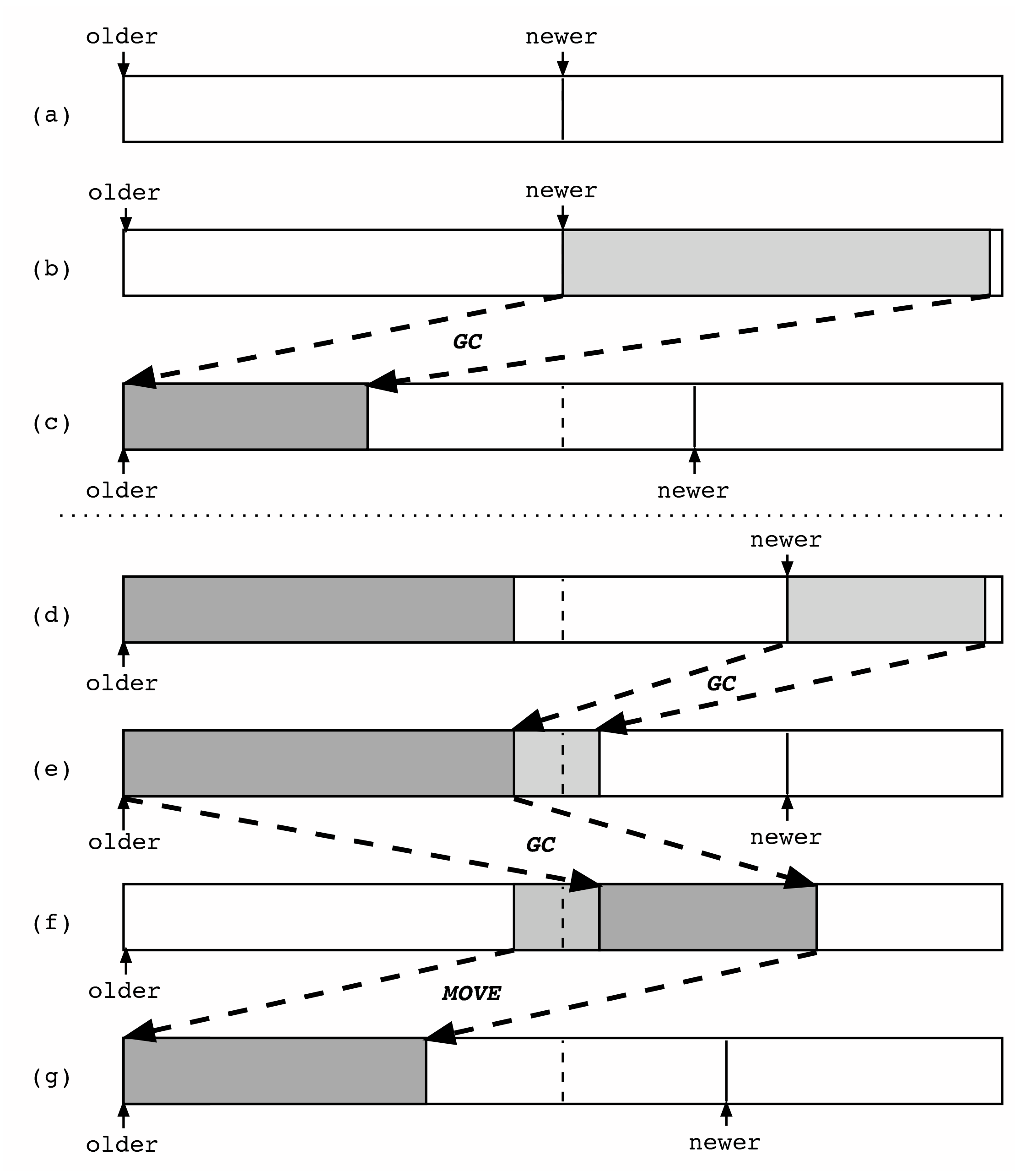

Figure 11: GC: heap states at various points in mutation and collection.

major cycle get away with ignoring the memory collected in the last minor? And how can we block-move an entire memory region without invalidating pointers into that region? The answer to the former is simple: all the data copied into older by the last minor is *guaranteed live*, since that's the criterion by which it's copied. There is no need to collect a region which we know to be 100% live. The answer to the latter is that we cannot simply copy the area as we claimed. We must also re-adjust all memory references which point into the space where our region used to be. Every root and every heap pointer is potentially in need of re-adjustment.

5.2.1 Implementation overview

Before beginning to describe the collector itself, we need to explain some low-level choices for data structures and introduce some C-level terminology. We chose to represent the heap as a contiguous region of bytes, instead of say, words. As such, our fundamental data type is

```
typedef char* GCPtr;
```

and the heap is presided over by a number of pointers:

```
GCPtr heap_begin;         /* also the beginning of older */
GCPtr heap_reserve;       /* end of older */
GCPtr heap_newer;
GCPtr heap_allocpt;       /* end of newer */
GCPtr heap_middle;
GCPtr heap_end;
```

The end of *newer* (`heap_allocpt`) is where new memory is allocated. The allocation routine simply increments `heap_allocpt` and returns its value prior to the increment.

The collector is based around one fundamental routine, inspired by the implementation of Appel's collector in the 1992 version of the Standard ML of New Jersey [4] runtime system:

```
void gc(GCPtr from_begin,     /* beginning of space to collect */
        GCPtr from_limit,     /* end of space to collect */
        GCPtr to_begin,       /* beginning of "to" space */
        GCPtr to_limit,       /* end of "to" space */
        GCPtr start_ptr,      /* end of data already copied */
        GCPtr **roots,        /* roots */
        GCPtr **wb,           /* write barrier */
        GCPtr *from_new);     /* return end of copied data */
```

The `gc()` routine intends to be fully agnostic with respect to the rest of the collector's context. It refers to no global data and makes no special assumptions. It simply copies all the live data between `from_begin` and `from_limit` into the region delimited by `to_begin` and `to_limit`. The notion of liveness, for now, is defined in terms of reachability via a pointer path of any length starting at the `roots` array or the write barrier array (`wb`). It begins copying data into the "to" space starting at `start_ptr`, just in case some data has already been copied there prior to the current invocation of `gc()`. Admittedly this feature exists solely to make major collection easier, but the routine makes no further assumption about whether it is performing a minor or a major cycle.

In addition to `gc()`, the collector is composed of a number of other routines, the most important of which are:

```
/* public interface */
void  gc_init(void);
Value gc_alloc(size_t nbytes)
void  gc_collect(void);
void  gc_barrier_add(Value *v);
/* key private routines */
void  gc_callgc(GCPtr **roots,GCPtr **wb);
GCPtr forward(GCPtr ptr,
              GCPtr from_begin,
              GCPtr from_limit,
              GCPtr to_begin,
              GCPtr to_limit,
              GCPtr *next);
```

The mutator invokes `gc_init()` to initialize the collector: the heap is allocated, and proper initial values are given to all the the heap pointers. As mutation proceeds, memory is allocated via the `gc_alloc()` routine. Whenever an update which may assign a pointer into *newer* into a record in *older*, the mutator is required to invoke `gc_barrier_add()` to place a reference to the pointer in *older* on the write barrier. Finally, when the mutator decides that a collection should occur, `gc_collect()` is invoked.

Behind the public interface, when `gc_collect()` is invoked, all the (pointers to) roots are collected into a local array. The roots array and (the already existing) write barrier array are then passed to `gc_callgc()`. The latter routine invokes `gc()` to perform a minor collection:

```
gc(heap_newer, heap_allocpt, heap_reserve, heap_newer,
   heap_reserve, roots, wb, &new_reserve);
```

i.e., collect from *newer* into the end of *older*, and set `new_reserve` to the end of the copied memory. If in this process the older region straddles the middle of the heap, `gc_collect()` invokes a major:

```
gc(heap_begin, heap_reserve, heap_reserve, heap_end,
   new_reserve, roots, NULL, &major_end);
```

Notice the key trick: the memory between `heap_reserve` and `new_reserve` is ignored, since it is known to be 100% live (copied in the immediately preceding minor). After the major, the area between `heap_reserve` and `new_major` is our live data; it is block-moved to the beginning of the heap and all pointers are readjusted accordingly. Finally, `forward()` is used by `gc()` to forward a single pointer from one space to another. The `next` pointer holds the location where the record should be copied. The routine modifies `next` accordingly, leaves a forwarding pointer in the old record, and returns a pointer to the new copy.

5.2.2 Gathering roots

The following are sources of roots in FLC:

- the execution context
- the global thread queue
- arguments to the currently executing mutator function

The execution context contains the current store and the currently executing code. The store is a root; the code pointer is a root. The global thread queue is processed similarly. Every store and every code pointer associated with each thread are roots.

The arguments to the currently executing function are communicated between the mutator and the collector via an array. The array is set at system initialization time to hold pointers to all the variables in which function arguments are stored (see Section 3.1 for how function arguments are represented). The mutator then communicates the arity of the current function to the collector, so the collector will know exactly how many elements in the roots array to consider.

5.2.3 The write barrier

The write barrier is the set of pointers from the old generation into the newer generation. It is necessary to keep track of these pointers in order to maintain a correct notion of liveness throughout collection. To understand why we must keep track of a write barrier, consider a record in *newer* which is pointed to exclusively from *older*. When running a minor collection cycle, we only consider the typical roots (previous section) to determine liveness. Therefore, our record would be found to be garbage and subsequent uses of it (through the *older* pointers) would be invalid. To ensure that the collector does not leave these kinds of dangling pointers, the write barrier must be actively maintained and considered as a source of roots.

The write barrier is eligible to be extended at any point in the mutator which involves updating a record field. In the case of FLC, updates are strictly limited to the `update` operation in the first-class store implementation. In a chained hash table implementation for first-class stores, however, there are many (subtle) opportunities for extending the write barrier, since internal table operations may involve table resizing and bucket reassignment. Namely:

- When inserting a value in the table, which we place at the head of the bucket list, if the table is in *older*, the address of bucket goes on the write barrier.

- When updating, if the node whose value is to be replaced is in *older* and the new value is in *newer*, the address of the `value` field in that node goes on the write barrier.

- When resizing a table, and therefore re-shuffling nodes, if the `next` pointer of a node in *older* is assigned a bucket in *newer*, the address of the next pointer goes on the write barrier.

Finally, the hash table insertion operation may return a new table, acquired via an internal resize. In that case, if the store reference record which points to that table is in *older*, the address of its `table` field goes on the write barrier.

5.3 A new notion of liveness

As discussed in earlier, without special modifications to the collector, our first-class store implementation violates some of the properties we established in Section 4, most importantly the *Stores Correct Collection* property. A liveness estimate based strictly on reachability is therefore inadequate for our needs. To better understand why it is so crucial that we adjust the implementation to respect all established properties, consider Figure 12. The arrows represent computation paths, or threads; each "fork" represents a narrowing point, where an thread is split into two new threads. The right-side branches of the `checkpoints` keep computing, repeatedly hitting narrowing points. The left-side branches, however fail quickly after updating the reference `p` with a value. Upon failure,

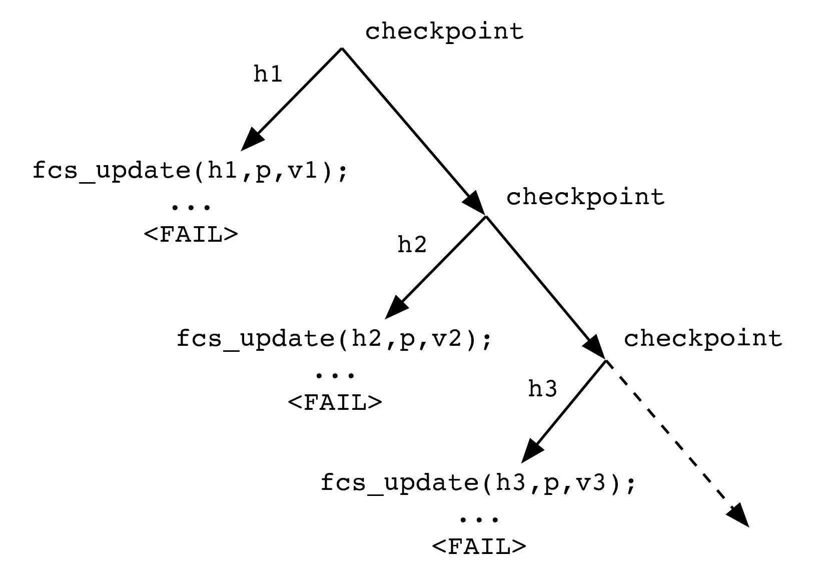

Figure 12: Bad checkpoint situation

the thread becomes garbage and so does its respective store (denoted by `h1`, `h2`, and `h3`). Since the stores die right away, if the reference `p` is still reachable somewhere else in the program (which is likely), the values `v1`, `v2`, etc., will be reachable though they are obviously garbage. This is clearly a problem if this "narrow and fail" pattern computes for a considerable amount of time and if the values `v1`, `v2`, etc., are large objects. The program `nat` (See Appendix A) behaves precisely in this manner. We will later see (Section 6) that modifying the collector in the manner described in this section is indeed essential.

To adjust our notion of liveness, we introduce a couple of definitions:

Definition(FLC Reachability): A record is *reachable* if a pointer to it can be acquired via a pointer traversal of any length starting at the roots or the write barrier.

Definition(FLC Garbage): A record is *garbage* if either

- It is not reachable, or
- It is reachable only from one or more store references in which it is associated with a store tag whose corresponding store is garbage.

Then, any record that is *not* garbage is said to be live. Without additional work, it is obvious that the second clause of the FLC Garbage definition is violated. We must augment the collector with code which explicitly forces those particular records to garbage.

5.3.1 Implementation

Given our "references hold data" approach to FLC data representation, a fairly simple, bookkeeping-free, garbage collection algorithm can be devised:

1. Copy all the live stores from the allocation space to the "to" space; all stores that are garbage remain in the allocation space.

2. Copy all other live data; for each store reference node, check if the tag belongs to a store that remained in the allocation space (and is therefore garbage). If the store is garbage, re-link the store reference list around that particular node.

The high level sketch may seem straightforward, but there are a number of subtle details.

First of all, how do we copy all the live stores to the "to" space while ignoring all other data? Since all pointers to live stores in the allocation space are either in the current thread's context or on the global thread queue, they are known to be on the roots array. In a first phase, we could walk the roots array and copy to the to space only stores and nothing else:

```
for each root r
  if r is a store
    forward r to the "to" space
```

Then, in a forwarding phase, we could forward all the stores' `next` pointers, making sure that all live stores are ultimately in the to space:

```
while there are more records to consider
  scan <- next record
  if scan is a store
    forward scan->next to the "to" space
```

Finally, we could begin a normal collection.

Another question is, how can we identify whether a store is garbage given only its tag? Our answer involved simply using store pointers, as opposed to tags, to tag values in store references. We then have a simple way of determining whether a reference node should be re-linked around: a reachable reference node is said to be garbage, hence it can be safely discarded, if its store pointer points to a store record which resides in the allocation space and has not been forwarded to the "to" space.

The approach we took avoids walking the roots array several times, at the expense of some code complexity. In the initial phase, we forward all the roots and the write barrier, which includes all the live stores. However, when encountering a store reference, we put it on a worklist for future consideration. When done forwarding all the live data, we turn to the worklist, processing it via an algorithm similar to the following:

```
for each item $s$ on the worklist
  for each element $b$ in $s$->$table$
    for each node $n$ in $b$
      if $n$ is garbage
        link prev($n$) to next($n$)
      else
        forward $n$ to the "to" space
```

We then alternate between forwarding and worklist processing until no more data remains to be scanned. In the forwarding phase, we put all the store references on the worklist. We then process the worklist, which may uncover yet more store references.

5.4 Conclusion

We have presented the design implementation of a generational copy collector for FLC, and augmented it with code that explicitly forces to garbage some reachable values that are known to be garbage. Our decision to choose a copy collector was based on the observation that allocation in FLC must be very fast; a contiguous allocation space fulfilled this requirement. Our implementation, especially of the aforementioned augmentation, is by no means optimized, and we intend to improve it in future work (see Section 7).

6 System Evaluation

Figures 13, 14, and 15 give basic performance comparisons between our system running without a collector, with a basic collector, and with a collector modified to account for our adjusted definition of liveness (Section 5.3). The numbers show averages over ten different runs in each case, and the tests were performed on a 1.25 GHz Apple Powerbook G4 with 1GB of RAM.

We found it hard to perform a highly accurate, head-to-head comparison due to the fact that our augmented collector is different than the normal collector in ways besides the addition of the worklist algorithm. First, data representation for a few of the runtime structures is slightly different between the two collectors, causing the augmented one to allocate at a slightly slower rate, thus causing fewer collections for an identical program run in identical conditions. Even after

	`ndtest 7`	`perms 7`	`nat 400 400`	`fibo 27`
no collector	1.51	1.53	2.15	4.75
collector	(30M) 1.98	(30M) 2.31	(40M) 5.93	(100M) 13.90
augm. collector	(30M) 2.21	(30M) 2.63	(40M) 1.87	(100M) 10.81

Figure 13: Evaluation: basic run times (seconds) with heap sizes

	`ndtest 7`	`perms 7`	`nat 400 400`	`fibo 27`
collector	0.300/0.546	0.690/0.482	3.546/0.509	5.254/1.195
augm. collector	1.219/0.019	0.817/0.481	0.020/0.355	4.302/0.089

Figure 14: Evaluation: time spent in collector/`deref` (seconds)

	`ndtest 7`	`perms 7`	`nat 400 400`	`fibo 27`
collector	75.83	74.77	76.33	73.12
augm. collector	21.07	22.89	7.43	71.54

Figure 15: Evaluation: percentage of live data (calculated at majors)

adjusting heap sizes so that the two collectors performed an equal number of collections, we still saw notable differences in runtime behavior. The main effect of these differences can be seen in the `(fibo 27)` column in Figure 13. We should expect this program to exhibit identical run times in the two collectors, possibly slower on the augmented collector, due to the fact that the worklist code does nothing useful in pure functional programs (i.e., programs which run in only one store on which `checkpoint` is never invoked). Nonetheless, our final collector is slightly faster; we suspect that low level data locality and layout differences play a major role in the discrepancy. Finally, GC performance is highly dependent on GC parameters. Since, as it stands, GC parameters are fully static (e.g., heap size is picked once at compile time), numbers between the uncollected and collected systems should be taken with a grain of salt.

Figure 13 shows basic run times on the three systems on four test programs, where the number in parentheses shows the (fixed) heap size that the program was run in, where applicable. Going from an uncollected system to the basic collector, we see a collector-induced performance overhead from about 30% in programs with a significant amount of non-determinism to an extreme 290% in the case of `(fibo 27)`. The latter is due to the fact that the time spent in the collector is largely wasted, since not much of the (very large amount of) data is garbage at any point. Again, better performance could have been easily achieved with a larger heap. Going from the simple collector to the augmented one, we see a worklist-induced performance overhead of about 12% in the `ndtest` and `perms` tests and, as discussed, a surprising speedup in the `fibo` case. The `nat` test, however, exhibits a significant speedup over both the simple collector and the original uncollected system.

Admittedly engineered to exhibit exactly the problem that the augmented collector fixes, `(nat` n l`)` (see Appendix A for the source) narrows n times, and at each narrowing point, it allocates a list of length l in a store whose corresponding thread dies immediately. The augmented collector easily wins because it identifies all the lists as garbage and collects them, while the simple collector keeps *every* allocated list live and scans them on each cycle. The reason that the augmented collector also outperforms the uncollected system is in part because heap operations are less expensive in the collected system: pointers contain fewer values, which cuts down on hash table size and number of times a table is resized during the program run.

The latter two figures compare the collected systems. Figure 14 shows collection overhead in time spent in the collector. The amount of time spent in the `deref` operation is also shown, to give an idea of how the added code affected its performance. The `ndtest` and `perms` examples show why the slowdown rate in Figure 13 was so small: the significant increase in collection overhead (due to the added worklist code) is proportional to a significant increase in `deref` *performance*; again, due to the fact that `deref` traverses fewer nodes on average. Once again, `nat` shows a drastic overhead decrease in *both* `deref` and the collector. The former is due to the fact that, as explained above, the collector avoids scanning any of the allocated lists. The `fibo` test does not exhibit a big difference, as expected.

Finally, Figure 15 shows the percentage of live data throughout the program run. We calculated the live data at major collections; the numbers presented are averages over all majors in each run. The data indicates that the augmented collector does indeed collect more garbage in nondeterministic programs than the simple collector can. The difference is most evident in `nat`, as expected, but significant improvements can be seen in both `ndtest` and `perms`. Again, `fibo` exhibits no big difference.

7 Conclusions and Future Work

We have designed, implemented, evaluated, and presented an efficient, hashing-based data representation for first-class stores, and a complete generational copy collector that is novel in its treatment of first-class store data structures. In order to make our system exhibit intuitive space behavior, we found it necessary to augment the collector with code that explicitly forces to garbage some values which are reachable but known to be garbage. Our evaluation of the completed system shows promising results. In particular, the performance overhead in nondeterministic programs is manageable (especially given that our collector has not been optimized for speed), and the collector induces a performance boost in highly nondeterministic programs with a high rate of search failure. Though we are not not ready for a formal comparison between FLC and other popular FLP systems, a quick informal survey that FLC programs are between four and eight times slower than equivalent programs running in the Münster Curry Compiler. Given that our system has not yet been optimized, we believe that it has very good potential.

The most obvious area of the collector that we can (and plan to) improve in the immediate future is the code that forces dead values to garbage. Figure 14 shows that, when the these values are small (e.g., boxed integers), the collection overhead increases significantly. We have a number of ideas for how to improve this code. One is to keep all the stores on a side list (or chain them together) to allow a fast way to forward them all in one step. When forwarding the rest of the data, we would already know which stores are live and which are not.

In order to begin comparing our system to the Münster Curry Compiler, Pakcs, and the FLVM, there are several standard optimizations that we would like to apply. First, we would like to unbox integers. We expect this to complicate the collector significantly, but it should yield a significant speedup in numerically intensive code. Second, the runtime code, as it stands, is not optimized. Our queue representation is relatively inefficient, for example, and several routines in the runtime (including the collector) are inefficient first-approximations. Finally, we would like to improve the FLC compiler backend with several static analyses.

Acknowledgements

Many thanks to my research advisor, Andrew Tolmach, for his very effective guidance, useful criticism, and helpful comments on this paper. Many thanks to Prof. Sergio Antoy for his very generous support during this project.

References

[1] Antoy, S., Hanus, M., Liu, J., and Tolmach, A. A Virtual Machine for Functional Logic Computations. In *Proc. of the 16th International*

Workshop on Implementation and Application of Functional Languages (IFL 2004) (2004), Technical Report 0408, University of Kiel, pp. 169–184.

[2] Appel, A. W. Simple Generational Garbage Collection and Fast Allocation. *Software Practice and Experience 19*, 2 (1989), 171–183.

[3] Appel, A. W. *Compiling with Continuations.* Cambridge University Press, 1992.

[4] Appel, A. W., and MacQueen, D. B. Standard ML of New Jersey. In *Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming* (1991), J. Maluszyński and M. Wirsing, Eds., no. 528, Springer Verlag, pp. 1–13.

[5] Driscoll, J. R., Sarnak, N., Sleator, D. D., and Tarjan, R. E. Making Data Structures Persistent. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing* (1986), ACM Press, pp. 109–121.

[6] G. B. Leeman, J. A Formal Approach to Undo Operations in Programming Languages. *ACM Trans. Program. Lang. Syst. 8*, 1 (1986), 50–87.

[7] Hanus, M. A Unified Computation Model for Declarative Programming. In *Proc. 1997 Joint Conference on Declarative Programming (APPIA-GULP-PRODE'97)* (1997), pp. 9–24.

[8] Hanus, M., Antoy, S., Höppner, K., Koj, J., Niederau, P., Sadre, R., and Steiner, F. PAKCS: The Portland Aachen Kiel Curry System, 2002.

[9] Hanus (ed.), M. Curry: An Integrated Functional Logic Language (Vers. 0.7). Available at `http://www.informatik.uni-kiel.de/~curry`, 2000.

[10] Johnson, G. F., and Duggan, D. Stores and Partial Continuations as First-Class Objects in a Language and its Environment. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1988), ACM Press, pp. 158–168.

[11] Leroy, X., Doligez, D., Garrigue, J., Remy, D., and Vouillon, J. The Objective Caml System.

[12] Lloyd, J. Programming in an Integrated Functional and Logic Language. *Journal of Functional and Logic Programming 1999*, 3 (1999), 1–49.

[13] López-Fraguas, F., and Sánchez-Hernández, J. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99* (1999), Springer LNCS 1631, pp. 244–247.

[14] Lux, W. The Münster Curry Compiler, 2004.

[15] Morrisett, J. G. Refining First-Class Stores. In *Workshop on State in Programming Languages* (Copenhagen, Denmark, June 1993).

[16] Okasaki, C. *Purely Functional Data Structures.* Cambridge University Press, 1998.

[17] Stallman, R. M. Using and Porting GCC. Technical report, The Free Software Foundation, 1993.

[18] Thomas H. Cormen, Charles E. Leiserson, R. L. R. C. S. *Introduction to Algorithms.* McGraw-Hill, 2001.

[19] Tolmach, A., Antoy, S., and Nita, M. Implementing Functional Logic Languages Using Multiple Threads and Stores. In *Proc. of the 2004 International Conference on Functional Programming (ICFP)* (Snowbird, Utah, USA, September 2004), ACM, pp. 90–102.

[20] Tolmach, A. P., and Appel, A. W. A Debugger for Standard ML. *Journal of Functional Programming 5*, 2 (1995), 155–200.

[21] Wilson, P. R. Uniprocessor Garbage Collection Techniques. In *Proc. Int. Workshop on Memory Management* (Saint-Malo (France), 1992), no. 637, Springer-Verlag.

[22] Wilson, P. R., and Moher, T. G. Demonic Memory for Process Histories. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation* (1989), ACM Press, pp. 330–343.

A Source Code for Tests

```
-- because mcc and pakcs disagree on the type of (&>)
onSuccess :: Success -> a -> a
onSuccess u v | u = v

norm x = seq x x -- x can't be free
normList [] = []
normList (x:xs) = (norm x):(norm (normList xs))

-- nat --

data Nat = Zero | Succ Nat

nat :: Int -> Int -> Int
nat n l = nat' n x z
  where nat' :: Int -> Nat -> [Int] -> Int
        nat' n Zero     z = onSuccess (z =:= normList [0..l]) failed
        nat' n (Succ x) z
          | n == 0       = 1
          | otherwise    = nat' (n-1) x z
        x,z free

-- ndtest --

member :: a -> [a] -> Bool
member y []      = False
member y (x:xs)
  | x == y       = True
  | otherwise    = member y xs

permute :: [a] -> [a]
permute []     = []
permute (z:zs) = onSuccess (u ++ v =:= permute zs) (u ++ (z:v))
                   where u,v free

ndtest :: Int -> Int
ndtest n = onSuccess (member 0 (permute [0..n]) =:= True) failed
```

```
-- perms --

perms :: Int -> Success
perms n = permute [0 .. n] =:= []

-- fact --

fact :: Int -> Int
fact n | n == 0    = 1
       | otherwise = n * (fact (n-1))

-- fibo --

fibo :: Int -> Int
fibo n | n < 2     = 1
       | otherwise = fibo (n-1) + fibo (n-2)
```