

# Toward the Efficient Compilation of Fair Search

Marius Nita

`marius@cs.pdx.edu`

(joint with Andrew Tolmach)

# Outline

---

- ◆ (Brief) FLP intro
- ◆ Fair vs unfair search
- ◆ Implementing fair search
- ◆ First-class stores
- ◆ Design constraints
- ◆ Data representation
- ◆ Garbage collection
- ◆ Results, conclusion

# Functional Logic Programming

---

- ◆ One paradigm unifying functional and logic programming
- ◆ From the functional world (Curry syntax):

- ◆ (Higher-order) functions

`s f g x = f x (g x)`

- ◆ Algebraic datatypes, pattern matching

`data T = B Bool T T | L`

`flatten L = []`

`flatten (B x l r) = flatten l ++ x:(flatten r)`

- ◆ Haskell  $\subset$  Curry (or it can be)

# Functional Logic Programming

---

- ◆ From the logic world:

- ◆ Logic variables, built-in search

- `flatten x where x free`

- instantiates `x` to `L` or `(B _ _ _)` and looks for solutions

- ◆ Nondeterminism

- `coin = 0`

- `coin = 1`

- (evaluating `coin` yields either 0 or 1,  
nondeterministically – in theory)

# Built-in Search

---

Back to our example:

```
data T = B Bool T T | L
```

```
flatten L = []
```

```
flatten (B x l r) = flatten l ++ x:(flatten r)
```

We load this program in Pakcs and run the following goal:

```
test> flatten X
```

```
Free variables in goal: X
```

```
Result: ([])
```

```
Bindings:
```

```
X=L ?
```

(Capital letters denote free variables in the top level.)

# Built-in Search

---

In a nutshell: When a free variable  $X$  is demanded, e.g., our operation `(flatten X)`:

1. Find the *set* of values (constructors) that it can be instantiated to, based on its type.

The type of  $X$  is  $T$ . Possible values are  $L$  and  $(B\ X_1\ X_2\ X_3)$ , where  $X_i$ s are free.

2. For each value  $v$ , instantiate  $X$  to  $v$  and compute  $(f\ X)$ .

`flatten L`  $\Rightarrow$  `[]` (our solution!)

`flatten (B X1 X2 X3)`  $\Rightarrow$  `flatten X2 ++ X1:(flatten X3)`

The latter will spit out an infinite number of solutions.

# Fair vs. Unfair Search

---

How do we find solutions algorithmically? On previous slide:

For each value  $v$ , instantiate  $X$  to  $v$  and compute `(flatten X)`.

**Strategy A** Instantiate  $X=L$ , compute `(flatten X)` until no more computation can be done. Then instantiate  $X=(B \ X_1 \ X_2 \ X_3)$  and repeat.

**Strategy B** Spawn two parallel computations, one for  $X=L$  and one for  $X=(B \ X_1 \ X_2 \ X_3)$  and compute `(flatten X)` in each.

# Unfair Search

---

**Strategy A** (depth-first search) is the norm. Popular FLP implementations, including Pakcs and the Münster Curry Compiler (MCC), search for solutions depth-first.

A well-known defect of depth-first search is that it is *incomplete*. Most simply, when

$$f(B \ x \ r \ l) = \text{undefined}$$
$$f \ L \quad \quad = []$$

$(f \ X)$  should yield the solution  $[]$  at some point. If we search depth-first and pick the first branch before the second, this never occurs.

# Fair Search

---

**Strategy B** (breadth-first search) is *complete*. All solutions that can be computed in a finite amount of steps will be discovered in a finite amount of time.

- ◆ We compute all alternatives in (pseudo-)parallel.
- ◆ Each alternative reports its result when it is done computing.
- ◆ If some alternatives diverge, the program will naturally diverge. But not before reporting all computable solutions.

# Implementing Fair Search

---

Each search alternative computes independently from the others. Therefore, some data involved in the computation *must* be distinct for each alternative.

**A High-Level Approach** We are computing `(flatten X)`:

1. Replicate the term `n` times, where `n` is the number of possible values for `x`.
2. In each replica, substitute `x` for one of the values.
3. Proceed with computing all replicas in parallel.

# Fair Search: Term Replication

---

(flatten X)

$\Rightarrow \left[ \begin{array}{c} \text{(flatten L)} \\ \text{(flatten (B X}_1 \text{ X}_2 \text{ X}_3)) \end{array} \right]$

$\Rightarrow \left[ \begin{array}{c} \square \\ \text{(flatten X}_2 \text{)} ++ \text{X}_1 : \text{(flatten X}_3 \text{)} \end{array} \right]$

$\Rightarrow \left[ \begin{array}{c} \text{(flatten L)} ++ \text{X}_1 : \text{(flatten X}_3 \text{)} \\ \text{(flatten (B X}_4 \text{ X}_5 \text{ X}_6)) ++ \text{X}_1 : \text{(flatten X}_3 \text{)} \end{array} \right]$

# Fair Search: Term Replication

---

Possible problem: duplicating large terms can be a big time and space expense.

The situation can be improved by avoiding replication of sub-terms which do not contain free variables.

E.g., `(not X, reallyBigTerm)` yields

`(True, E1)`

`(False, E2)`

If `reallyBigTerm` doesn't contain free variables, `E1` and `E2` will point to the same object, which was obtained by reducing `reallyBigTerm` *once*.

# Fair Search: Store Replication

---

## A Low-Level Approach

- ◆ We compile all functions to fixed machine code.
- ◆ We largely abandon notions of “term duplication” and “reduction.” (Machine code can’t be duplicated anyway.)
- ◆ We introduce a concept of *store*: mapping from logic variables to values. Each alternative has its own store that is independent from all other stores.
- ◆ When the value of a free variable is demanded, we **duplicate the store** instead of the term.

# Fair Search: Store Replication

$$(\text{flatten } \mathbf{X})_{s \equiv \{X \mapsto \text{Free}\}}$$

$$\Rightarrow \left[ \begin{array}{c} (\text{flatten } X)_{s' \equiv s[X \mapsto L]} \\ (\text{flatten } X)_{s'' \equiv s[X \mapsto (B \ X_1 \ X_2 \ X_3), X_1 \dots X_3 \mapsto \text{Free}]} \end{array} \right]$$

$$\Rightarrow \left[ \begin{array}{c} \square_{s'} \\ (\text{flatten } \mathbf{X}_2) \uparrow\uparrow X_1 : (\text{flatten } X_3)_{s''} \end{array} \right]$$

$$\Rightarrow \left[ \begin{array}{c} (\text{flatten } X_2) \uparrow\uparrow X_1 : (\text{flatten } X_3)_{s''[X_2 \mapsto L]} \\ (\text{flatten } X_2) \uparrow\uparrow X_1 : (\text{flatten } X_3)_{s''[X_2 \mapsto (B \ X_4 \ X_5 \ X_6), X_4 \dots X_6 \mapsto \text{Free}]} \end{array} \right]$$

# First-Class Stores

---

Logic variable stores are first-class stores (FCS):

- ◆ Stores are mappings from *store references* to *values*
- ◆ Stores are objects that can be passed in and out of functions, stored in variables, etc.
- ◆ Stores can be copied (or “checkpointed”)
- ◆ Store references can be dereferenced in any “valid” store.

# FCS: Interface

---

A C interface for our formulation of first-class stores:

```
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);
```

**Important:** If a reference exists in a store  $s$ , it also exists in any store  $s2 = \text{fcs\_checkpoint}(s)$ , for as long as  $s2$  is live.

# FCS in FLP

---

- ◆ We use stores to map logic variables to values.
- ◆ A store reference corresponds to a logic variable.
- ◆ The code manipulates only store references and is store-agnostic. (This will come in handy!)
- ◆ The `fcs_checkpoint` operation is used whenever stores must be duplicated. (Whenever we spawn parallel search alternatives.)
- ◆ Intuitively, two search alternatives running the same code (dereferencing the same references) will have different outcomes if their stores are different.

# Goal of This Work

---

We want an efficient *compiler* for FLC that employs *fair search* via *store replication*. Pieces involved:

- ◆ A compiler from a high level FLP to machine code
- ◆ An efficient data representation and implementation for first-class stores
- ◆ A runtime system composed of
  - ◆ concurrency primitives
  - ◆ a garbage collector that correctly and efficiently collects stores

This work addresses items in blue.

# Data Representation

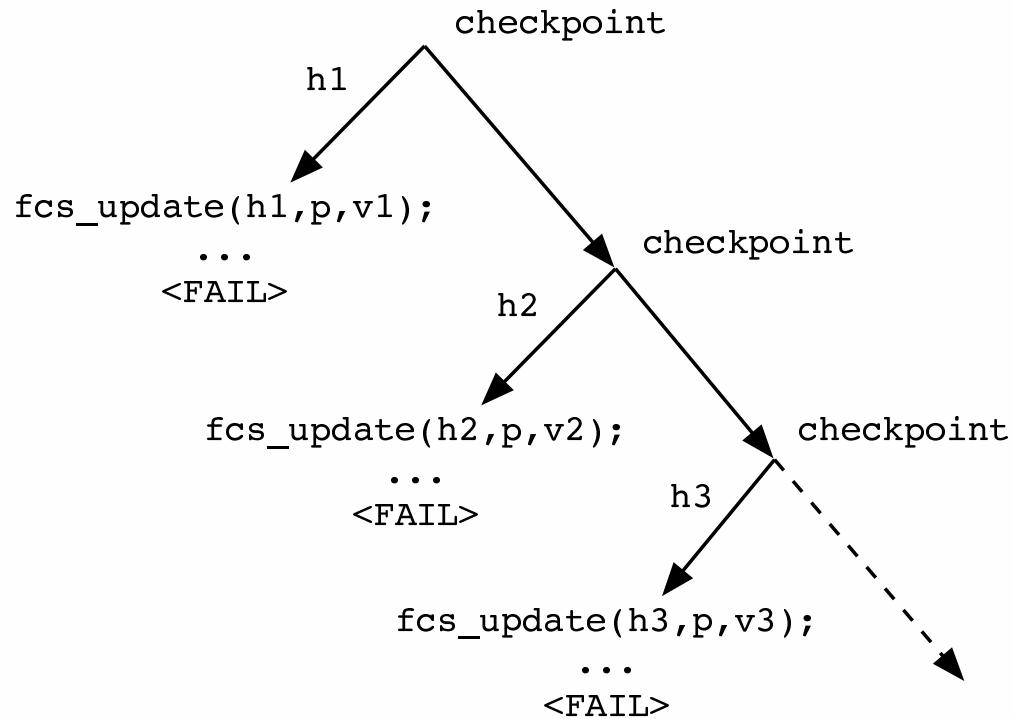
---

- ◆ One *crucial* observation: For a store value to be live, it must be reachable via a reference *and* a store.
- ◆ If it is only reachable through a reference, it is garbage.
- ◆ If it is only reachable through a store, it is garbage.
- ◆ Intuitively, we need both a store and a reference in order to pull out the value and use it. Otherwise it's out of our reach.

**Consequence:** Some values may be garbage but still reachable!

# Data Representation

Consider the following situation:



If  $p$  remains live as the computation proceeds, we must make sure that  $v1$ ,  $v2$ , etc., do not.

# FCS: A Simple Approach

---

We could represent first-class stores as contiguous arrays and references as integer offsets into stores.

**Pros:** Fast dereferencing.

**Cons:** Checkpointing is inefficient, especially space-wise.

**Serious cons:** Proper collection is intractable! When a reference becomes garbage, we need to kill off its corresponding values in all the stores that it can be dereferenced in.

# FCS: Getting Trickier

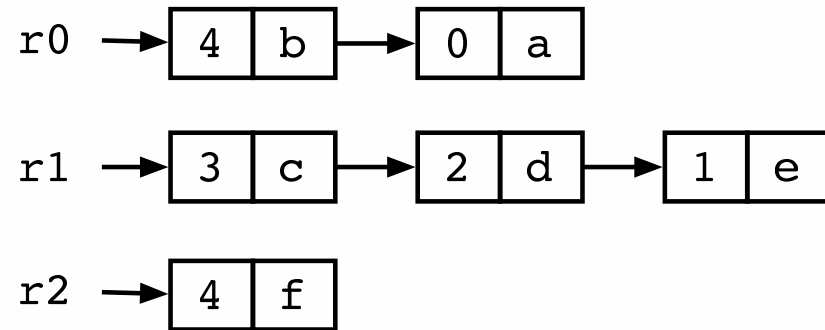
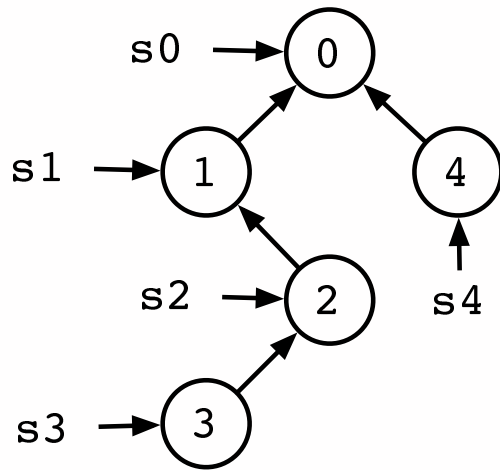
---

We represent first-class stores as integer lists and references as lists of possible values.

- ◆ A reference is a list of  $(t, v)$  pairs, where  $t$  is a tag that uniquely identifies a store.
- ◆ A store  $s$  is a list  $t_1, t_2, \dots, t_n$  of tags, where  $t_1$  is the (unique) tag of  $s$ ,  $t_2$  is the tag uniquely identifying its “checkpoint parent,” and so on.

This trick exploits an important invariant in FLC: after a  $s$  is duplicated in the process of spawning two or more search alternatives,  $s$  becomes unreachable!

# FCS: Linked Lists

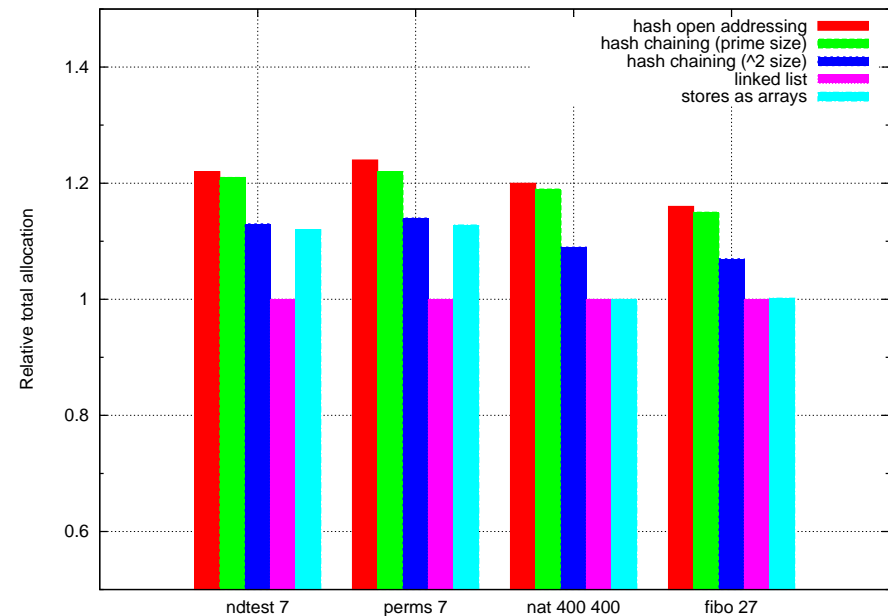
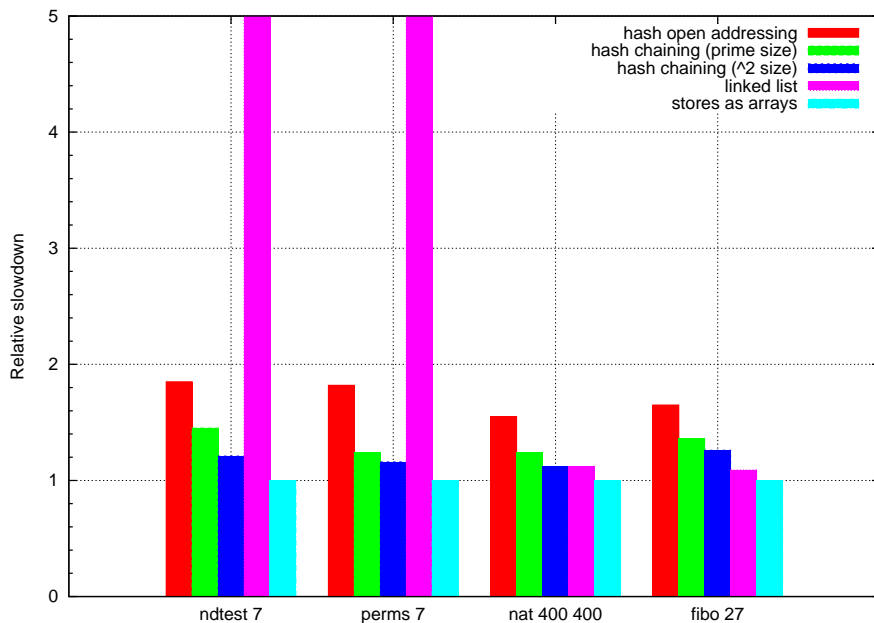


**Pros:** Makes good use of space and fixes our GC problem.  
(We will see how in a few slides.)

**Cons:** Dereferencing is slow. A value may be looked up in several stores before it is found.

# FCS: Hash Tables

The linked list approach is inadequate due to very slow dereferencing. Replacing linked lists with hash tables in the reference representation produced significantly better results.



# GC: General Guidelines

---

- ◆ Most data die young. Approx. 80% of the data is garbage by the time a collection should happen.
- ◆ Allocation must be *very* fast. The allocation routine holds the highest call percentage in the runtime.
- ◆ Frequent allocation  $\Rightarrow$  frequent collections. Since most data die young, we would like to avoid collecting “too early.”

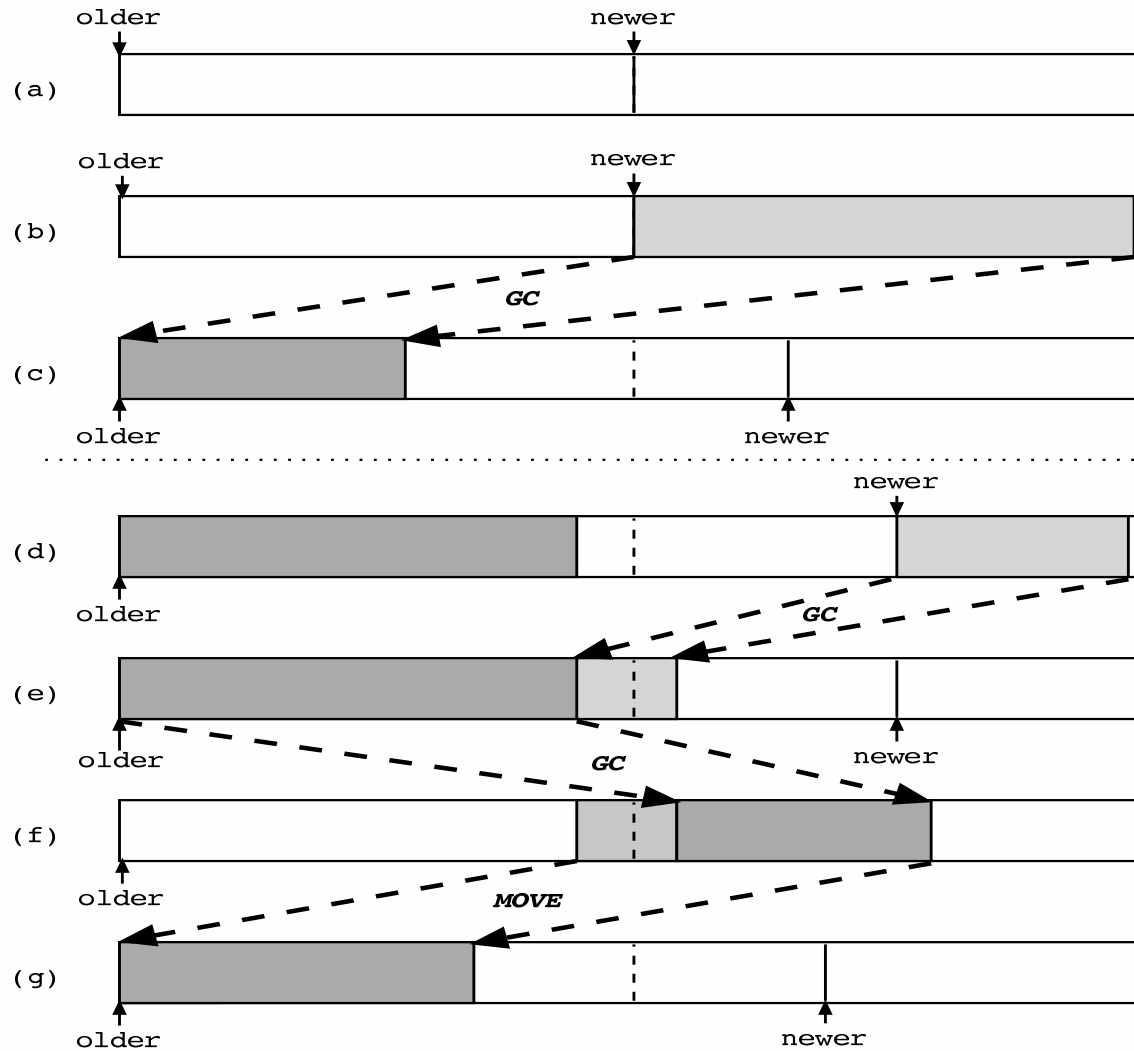
# Generational GC

---

We decided on a generational copy collector, following Appel's well known model [Appel '89].

- ◆ Implicitly compacting, so allocation happens in a contiguous free space. I.e., allocation is fast.
- ◆ Generational means collection is divided in two phases:
  - ◆ Minor: collect a little data at a time, frequently.
  - ◆ Major: collect the entire heap, much less frequently.
- ◆ Gives data more time to die.

# GC: Mechanics



# GC: Liveness Adjustment

---

- ◆ We represent references as hash tables from store tags  $t$  to elements  $(p, v)$ , where  $p$  is a *pointer* to the store object uniquely identified by  $t$ .
- ◆ Before doing a full collection, we first forward all live stores to the “to” space.
- ◆ During collection, when scanning live references, for each  $(p, v)$ , we can quickly determine if  $p$  is garbage:  $p$  is in the “from” space and lacks a forwarding pointer.
- ◆ If  $p$  is garbage, we remove  $(p, v)$  from the table and fail to scan it.

# Some Numbers

	ndtest 7	perms 7	nat 400 400	fibo 27
none	1.51	1.53	2.15	4.75
coll	1.98	2.31	5.93	13.90
augm	2.21	2.63	1.87	10.81

Micro-benchmark speed comparisons (seconds):

- ◆ **none**: no collector at all
- ◆ **coll**: generational copy collector
- ◆ **augm**: generational copy collector with adjusted notion of liveness

# Some Numbers

	ndtest 7	perms 7	nat 400 400	fibonacci 27
coll	0.30/0.54	0.69/0.48	3.54/0.51	5.25/1.19
augm	1.22/0.02	0.82/0.48	0.02/0.35	4.30/0.09

Time spent in the collector (seconds), accompanied by time spent in the dereference operation.

	ndtest 7	perms 7	nat 400 400	fibonacci 27
coll	75.83	74.77	76.33	73.12
augm	21.07	22.89	7.43	71.54

Percentage of live data (calculated at majors).

# Future Work

---

- ◆ Speed everything up!
- ◆ We are working to improve the GC code that does the explicit unlinking, as it is central to GC performance.
- ◆ Strictness analysis to avoid thunking and to unbox integers.
- ◆ Mode analysis/inference to eliminate superfluous checks. E.g., if a variable is known to be *bound*, avoid unnecessary store dereferencing operations and unnecessary concurrency.
- ◆ Serious comparisons with Pakcs, MCC, FLVM.

# Related Work

---

Most notably:

- ◆ The Münster Curry Compiler: a native code (unfair) compiler for Curry.

<http://danae.uni-muenster.de/~lux/curry/>

- ◆ The FLVM: a (fair) virtual machine for FLP.

<http://redstar.cs.pdx.edu/~antoy/flp/vm/>

- ◆ See ICFP'04 paper by Tolmach *et al.* for details on an interpreter closely related to FLC.

# The End

---

Full paper is at <http://www.cs.pdx.edu/~marinus/>.