Bottom-Avoiding Streams

Joseph P. Near[⋆], Ramana Kumar[⋆⋆], William E. Byrd, and Daniel P. Friedman

Indiana University, Bloomington, IN 47405, USA
{jnear,ramkumar,webyrd,dfried}@indiana.edu

Abstract. We present the first shallow embedding of *ferns*, stream-like bottom-avoiding, shareable, and implicitly-parallel data structures proposed in 1979. Our implementation is written as a portable library written in a small subset of R^{6}RS Scheme. Unlike the original conceptualization of ferns, our implementation is sequential but addresses most of the issues that would appear in a parallel implementation. We also present a non-trivial example showing the utility of ferns: a generalization of standard logic combinators.

1 Introduction

We present the first shallow embedding of *ferns* [1], a shareable data structure designed to avoid divergence. Our implementation is written in Scheme and is available as a portable R^{6}RS Scheme [2] library. In addition to this implementation, we also present an extended example demonstrating the power of ferns. In this example we provide a bottom-avoiding generalization of stream-based logic combinators.

In the early work of Friedman and Wise [3], the implementation of ferns assumed an arbitrary number of processors along with a proposed hardware interlock-free store instruction. We model the parallelism that inspired the ferns concept using preemptible computations on a single processor. Although the single processor model doesn't capture all of the ramifications of the original ideas, our implementation reveals enough of the intricacies so that ferns can be easily adapted to other architectures.

Ferns are constructed with *cons* and **cons**$_\perp$, originally called **frons** [4], and accessed by *car*$_\perp$ and *cdr*$_\perp$. Ferns built with **cons**$_\perp$ are like streams in that the *evaluation* of elements is delayed, permitting unbounded data structures. In contrast to streams, the *ordering* of elements is also delayed: convergent values form the prefix in some unspecified order, while divergent values form the suffix.

The rest of the paper is as follows. In Section 2, we give examples of familiar recursive functions using ferns. In Section 3, we give an extended example using ferns: a generalization of logic programming combinators. In Section 4 we describe the promotion algorithm [3] that characterizes the necessary sharing properties of ferns. In Section 5, we present our complete implementation

[⋆] now at MIT

[⋆⋆] on exchange from The Australian National University of **cons**$_\perp$, *car*$_\perp$, and *cdr*$_\perp$. In Section 6 we present related work. Finally, we present our conclusions and some problems for future research.

2 Examples

We begin by presenting several examples illustrating the nondeterministic properties of ferns, showing their similarities to and differences from traditional lists and streams. Later, we include examples that show that the natural recursive style can be used when programming with ferns and point out the advantages ferns afford the user.

2.1 Nondeterminism

Convergent elements of a fern form its prefix in some unspecified order. For example, evaluating the expression

(**let** ((s (**cons**$_\perp$ 0 (**cons**$_\perp$ 1 '()))))
  (*display* (*car*$_\perp$ s)) (*display* (*cadr*$_\perp$ s)) (*display* (*car*$_\perp$ s)))

prints either 010 or 101, demonstrating that the order of values within a fern is not specified in advance but remains consistent once determined, while

(**let** ((s_1 (**cons**$_\perp$ (! 6) $\perp$)) (s_2 (**cons**$_\perp$ $\perp$ (**cons**$_\perp$ (! 5) $\perp$))))
  (*cons* (*car*$_\perp$ s_1) (*car*$_\perp$ s_2)))

returns (720 . 120), demonstrating that accessing a fern avoids divergence as much as possible. ($\perp$ is any expression whose evaluation diverges.) In the latter example, each fern contains only one convergent value; taking the *cdr*$_\perp$ of s_1 or the *cadr*$_\perp$ of s_2 results in divergence.

Ferns are *shareable* data structures; sharing, combined with delayed ordering of values, can result in surprising behavior. For example, consider these expressions:

(**let** ((b (*cons* 2 '())))
  (**let** ((a (*cons* 1 b)))
    (**list** (*car* a) (*cadr* a) (*car* b))))

and

(**let** ((b (**cons**$_\perp$ 2 '())))
  (**let** ((a (**cons**$_\perp$ 1 b)))
    (**list** (*car*$_\perp$ a) (*cadr*$_\perp$ a) (*car*$_\perp$ b))))

where the first expression must evaluate to (1 2 2). The second expression may also return this value—as expected, the car of b would then be equal to the cadr of a. However, the second expression might instead return (2 1 2); in this case, the car of b would be equal to the car of a rather than to its cadr. Section 4 discusses sharing in detail.

2.2 Recursion

We now present examples of the use of ferns in simple recursive functions. Consider the definition of *ints-from*$_\perp$[1].

```
(define ints-from⊥
  (λt (n)
    (cons⊥ n (ints-from⊥ (+ n 1)))))
```

Then (*caddr*$_\perp$ (*ints-from*$_\perp$ 0)) could return any non-negative integer, whereas the streams version would return 2.

There is a tight relationship between ferns and lists, since every cons pair is a fern. The empty fern is also represented by (), and (*pair?* (**cons**$_\perp$ e_1 e_2)) returns #t for all e_1 and e_2. After replacing the list constructor *cons* with the fern constructor **cons**$_\perp$, many recursive functions operating on lists avoid divergence. For example, *map*$_\perp$ is defined by replacing *cons* with **cons**$_\perp$, *car* with *car*$_\perp$, and *cdr* with *cdr*$_\perp$ in the definition of *map*, and can map a function over an infinite fern (*caddr*$_\perp$ (*map*$_\perp$ *add*$_1$ (*ints-from*$_\perp$ 0))) where the result can be any positive integer.

Ferns work especially well with *annihilators*. True values are annihilators for *or*$_\perp$, so we can write

```
(define or⊥
  (λt (s)
    (cond
      ((null? s) #f)
      ((car⊥ s) (car⊥ s))
      (else (or⊥ (cdr⊥ s))))))
```

which searches in a fern for a true convergent value. and can avoid divergence if it finds one: (*or*$_\perp$ (**list**$_\perp$ $\perp$ (*odd?* 1) (! 5) $\perp$ (*odd?* 0))) returns some true value, where **list**$_\perp$ is defined as follows.

```
(define-syntax list⊥
  (syntax-rules ()
    ((_) '())
    ((_ e e* ...) (cons⊥ e (list⊥ e* ...)))))
```

We can define *append*$_\perp$ for ferns.

```
(define append⊥
  (λt (s1 s2)
    (cond
      ((null? s1) s2)
      (else (cons⊥ (car⊥ s1) (append⊥ (cdr⊥ s1) s2))))))
```

[1] λ_t is identical to λ, except it creates preemptible procedures. (See Section 5.)

To observe the behavior of $append_\perp$ we define $take_\perp$

```
(define take⊥
  (λt (n s)
    (cond
      ((null? s) '())
      ((not n) (cons (car⊥ s) (take⊥ n (cdr⊥ s))))
      ((zero? n) '())
      (else (cons (car⊥ s) (if (= n 1) '() (take⊥ (− n 1) (cdr⊥ s))))))))
```

which returns a list of the first n values in a fern. If $n = \#\mathsf{f}$, $take_\perp$ attempts to retrieve every element of the fern. When determining the nth value, it is necessary to avoid taking the $cdr_\perp$ after the nth value is determined.

Our definition of $append_\perp$ appears to work as expected: ($take_\perp$ 2 ($append_\perp$ (**list**$_\perp$ 1) (**list**$_\perp$ $\perp$ 2))) $\Rightarrow$ (1 2). Moving $\perp$ from the second argument to the first, however, reveals a problem: ($take_\perp$ 2 ($append_\perp$ (**list**$_\perp$ $\perp$ 1) (**list**$_\perp$ 2))) $\Rightarrow$ $\perp$. Even though we can see that the result of the call to $append_\perp$ should contain two convergent elements, taking the first two elements of that result diverges. This is because our current definition of $append_\perp$ requires that s_1 be completely exhausted before any elements from s_2 can appear in the result. If one of the elements of s_1 is $\perp$, then no element from s_2 will ever appear. The same is true if s_1 contains an unbounded number of convergent elements: since s_1 is never null, the result will never contain elements from s_2. As we will see with the definition of **mplus**$_\perp$ in Section 3.1, the solution to these problems is to interleave the elements from s_1 and s_2 in the resulting fern as in the next example.

Functional programs often share rather than copy data, and ferns are designed to encourage this programming style. Consider a procedure to compute the Cartesian product of two ferns:

```
(define Cartesian-product⊥
  (λt (s1 s2)
    (cond
      ((null? s1) '())
      (else (mplus⊥ (map⊥ (λ (e) (cons (car⊥ s1) e)) s2)
                    (Cartesian-product⊥ (cdr⊥ s1) s2))))))
```

```
(take⊥ 6 (Cartesian-product⊥ (list⊥ ⊥ 'a 'b) (list⊥ 'x ⊥ 'y ⊥ 'z)))

⇝ ((a . x) (a . y) (b . x) (a . z) (b . y) (b . z))
```

where ⇝ indicates *one* of the possible values. This definition ensures that the resulting fern shares elements with the ferns passed as arguments. Many references to a particular element may be made without repeating computations, hence the expression

```
(take⊥ 2 (Cartesian-product⊥ (list⊥ (begin (display #t) 5)) (list⊥ 'a ⊥ 'b)))
⇝ ((5 . a) (5 . b))
```

prints #t *exactly once.* (There are more examples of the use of ferns [5–8].)

3 Extended Example: New Logic Combinators

In this section, we use the task of logic programming as an extended example of the use of ferns in avoiding bottom while maintaining a natural recursive style. We compare two sets of logic combinators, one using streams and the other using ferns. We begin by describing and implementing operators **mplus**$_\perp$ and *bind*$_\perp$ over ferns, and go on to implement logic programming combinators in terms of these operators. The fern-based logic combinators are shown to be more general than the standard stream-based ones. (See the historical account of logic combinators of Wand and Vaillancourt [9].)

3.1 mplus and *bind*

To develop logic programming combinators in a call-by-value language, we must make **mplus**$_\perp$ itself lazy to avoid diverging when its arguments diverge. We accomplish this by defining **mplus**$_\perp$ as a macro that wraps its two arguments in **list**$_\perp$ before passing them to *mplus-aux*$_\perp$. In addition, **mplus**$_\perp$ must interleave elements from both of its arguments so that a fern of unbounded length in the first argument will not cause the second argument to be ignored.

```
(define-syntax mplus⊥
  (syntax-rules ()
    ((_ s1 s2) (mplus-aux⊥ (list⊥ s1 s2)))))

(define mplus-aux⊥
  (λt (p)
    (cond
      ((null? (car⊥ p)) (cadr⊥ p))
      (else (cons⊥ (caar⊥ p)
              (mplus-aux⊥ (list⊥ (cadr⊥ p) (cdar⊥ p))))))))

(define bind⊥
  (λt (s f)
    (cond
      ((null? s) '())
      (else (mplus⊥ (f (car⊥ s)) (bind⊥ (cdr⊥ s) f))))))
```

We use a fern constructor to make **mplus**$_\perp$ lazy: if one of the ferns in the argument to *mplus-aux*$_\perp$ is divergent, it can select the other one. For example, (*car*$_\perp$ (**mplus**$_\perp$ $\perp$ (**list**$_\perp$ 5))) returns 5. *bind*$_\perp$ avoids the same types of divergence as *map*$_\perp$ described in Section 2 but uses **mplus**$_\perp$ to merge the results of the calls to f. Thus, (*bind*$_\perp$ (*ints-from*$_\perp$ 0) *ints-from*$_\perp$) is an unbounded fern of integers; for every (nonnegative) integer n, it contains the integers starting from n and therefore every nonnegative integer n is contained $n + 1$ times. The interleaving leads to duplicates in the following example:
(*take*$_\perp$ 15 (*bind*$_\perp$ (*ints-from*$_\perp$ 0) *ints-from*$_\perp$)) $\rightsquigarrow$ (0 1 1 2 2 3 2 4 3 5 3 6 4 7 3).

The addition of $unit_\perp$ and $mzero_\perp$ rounds out the set of operators typically used to implement logic programs in functional languages.

```
(define unit⊥ (λt (σ) (cons σ '())))
(define mzero⊥ (λt () '()))
```

Using these definitions, we can run programs that require both nondeterminism and multiple unbounded ferns, such as this variant of Seres and Spivey [10]:

```
(car⊥ (bind⊥ (ints-from⊥ 2)
         (λt (a)
           (bind⊥ (ints-from⊥ 2)
             (λt (b)
               (if (= (* a b) 9) (unit⊥ (list a b)) (mzero⊥)))))))
⇒ (3 3).
```

In this example, the streams version diverges, since 2 does not evenly divide 9.

3.2 Implementation of Logic Programming Combinators

Our combinators comprise three *goal constructors*: $\equiv_\perp$, which unifies terms; **disj**$_\perp$, which performs disjunction over goals; and **conj**$_\perp$, which performs conjuction over goals. These goal constructors are required to terminate, and they always return a goal. A *goal* is a procedure that takes a substitution and returns a fern of substitutions.

```
(define-syntax ≡⊥
  (syntax-rules ()
    ((_ u v)
     (λt (σ)
       (let ((σ (unify u v σ)))
         (if (not σ) (mzero⊥) (unit⊥ σ)))))))

(define-syntax disj⊥
  (syntax-rules ()
    ((_ g1 g2) (λt (σ) (mplus⊥ (g1 σ) (g2 σ))))))

(define-syntax conj⊥
  (syntax-rules ()
    ((_ g1 g2) (λt (σ) (bind⊥ (g1 σ) g2)))))
```

Logic programs evaluate to goals; to obtain answers, these goals are applied to the empty substitution. The result is a fern of substitutions representing answers. We define $run_\perp$ in terms of $take_\perp$, described in Section 2, to obtain a list of answers from the fern of substitutions

```
(define run⊥
  (λt (n g)
    (take⊥ n (g empty-σ))))
```

where n is a non-negative integer (or #f) and g is a goal.

Given two logic variables x and y, we present some simple logic programs that produce the same answers in both fern-based and stream-based combinators.

$(run_\bot\ \#\mathsf{f}\ (\equiv_\bot\ 1\ x)) \Rightarrow (\{x/1\})$
$(run_\bot\ 1\ (\mathbf{conj}_\bot\ (\equiv_\bot\ y\ 3)\ (\equiv_\bot\ x\ y))) \Rightarrow (\{x/3, y/3\})$
$(run_\bot\ 1\ (\mathbf{disj}_\bot\ (\equiv_\bot\ x\ y)\ (\equiv_\bot\ y\ 3))) \Rightarrow (\{x/y\})$
$(run_\bot\ 5\ (\mathbf{disj}_\bot\ (\equiv_\bot\ x\ y)\ (\equiv_\bot\ y\ 3))) \Rightarrow (\{x/y\}\ \{y/3\})$
$(run_\bot\ 1\ (\mathbf{conj}_\bot\ (\equiv_\bot\ x\ 5)\ (\mathbf{conj}_\bot\ (\equiv_\bot\ x\ y)\ (\equiv_\bot\ y\ 4)))) \Rightarrow ()$
$(run_\bot\ \#\mathsf{f}\ (\mathbf{conj}_\bot\ (\equiv_\bot\ x\ 5)\ (\mathbf{disj}_\bot\ (\equiv_\bot\ x\ 5)\ (\equiv_\bot\ x\ 6)))) \Rightarrow (\{x/5\})$

It is not difficult, however, to find examples of logic programs that diverge when using stream-based combinators but converge using fern-based combinators:

$(run_\bot\ 1\ (\mathbf{disj}_\bot\ \bot\ (\equiv_\bot\ x\ 3))) \Rightarrow (\{x/3\})$
$(run_\bot\ 1\ (\mathbf{disj}_\bot\ (\equiv_\bot\ \bot\ x)\ (\equiv_\bot\ x\ 5))) \Rightarrow (\{x/5\})$

and given idempotent substitutions [11], the fern-based combinators can even avoid some circularity-based divergence without the occurs-check, while stream-based combinators cannot:

$(run_\bot\ 1\ (\mathbf{disj}_\bot\ (\equiv_\bot\ (\mathbf{list}\ x)\ x)\ (\equiv_\bot\ x\ 6))) \Rightarrow (\{x/6\})$

We can also write functions that represent relations. The relation *always-five*$_\bot$ associates 5 with its argument an unbounded number of times:

```
(define always-five⊥
  (λt (x)
    (disj⊥ (always-five⊥ x) (≡⊥ x 5))))
```

Because both stream and fern constructors do not evaluate their arguments, we may safely evaluate the goal (*always-five*$_\bot$ x), obtaining an unbounded collection of answers. Using $run_\bot$, we can ask for a finite number of these answers. Because the ordering of streams is determined at construction time, however, the stream-based combinators cannot even determine the first answer in that collection, while the fern-based combinators compute as many answers as desired:
$(run_\bot\ 4\ (always\text{-}five_\bot\ x)) \Rightarrow (\{x/5\}\ \{x/5\}\ \{x/5\}\ \{x/5\})$. This is because the definition of *always-five*$_\bot$ is left recursive.

In the next section we look at how the sharing properties of ferns are maintained alongside bottom-avoidance.

4 Sharing and Promotion

In this section, we provide examples and a high-level description of the *promotion algorithm* of Friedman and Wise [3]. The values in a fern are computed and *promoted* across the fern while ensuring that the correct values are available from each subfern, $\bot$'s are avoided, and non-$\bot$ values are computed only once. Ferns have structure, and there may be references to more than one subfern of a particular fern. Consider the example expression

```
(let ((δ (cons⊥ (! 6) '())))
  (let ((γ (cons⊥ (! 3) δ)))
    (let ((β (cons⊥ (! 5) γ)))
      (let ((α (cons⊥ ⊥ β)))
        (list (take⊥ 3 α) (take⊥ 3 β) (take⊥ 2 γ) (take⊥ 1 δ))))))

⇝ ((6 120 720) (6 120 720) (6 720) (720))
```

assuming **list** evaluates its arguments left-to-right. Since the subfern δ is itself a fern, accessing δ cannot retrieve values in the prefix of the enclosing fern α. We now describe in detail how the result of ($take_\perp$ 3 α) is determined, and the necessary changes to the fern data structure during this process. Whenever we encounter a nondeterministic choice, we shall assume a choice consistent with the value returned in the example.

During the first access of α the cdrs are evaluated, as indicated by the arrows in Figure 1a. Figure 1b depicts the data structure after ($car_\perp$ α) is evaluated. We assume that, of the possible values for ($car_\perp$ α), namely $\perp$ (which is never chosen), (! 5), (! 3), and (! 6), the value of (! 3) is chosen and *promoted.* Since the value of (! 3) might be a value for ($car_\perp$ β) and ($car_\perp$ γ), we replace the cars of all three pairs with the value of (! 3), which is 6. We replace the cdrs of α and β with new frons pairs containing $\perp$ and (! 5), which were not chosen. The new frons pairs are linked together, and linked at the end to the old cdr of γ. Thus α, β, and γ each become a fern with 6 in the car and a fern of the rest of their original possible values in their cdrs. As a result of the promotion, α, β, and γ become cons pairs, represented in the figures by rectangles.

Figure 1c depicts the data structure after ($cadr_\perp$ α) is evaluated. This time, (! 5) is chosen from $\perp$, (! 5), and (! 6). Since the value of (! 5) is also a possible value for ($cadr_\perp$ β), we replace the cadrs of both α and β with the value of (! 5), which is 120, and replace the cddr of α with a frons pair containing the $\perp$ that was not chosen. The cddr of β points to δ; no new fern with remaining possible values is needed because the value chosen for ($cadr_\perp$ β) was the first value available. As before, the pairs containing values become cons pairs.

Figure 1d depicts the data structure after ($caddr_\perp$ α) is evaluated. Of $\perp$ and (! 6), it comes as no surprise that (! 6) is chosen. Since the value of (! 6), which is 720, is also a possible value for ($car_\perp$ δ) (and in fact the only one), we update the car of δ and the car of the cddr of α with 720. The cdr of δ remains as the empty list, and the cdr of the cddr of α becomes a new frons pair containing $\perp$. The cdr of the new frons pair is the empty list copied from the cdr of δ. The remaining values are obvious given the final state of the data structure. No further manipulation of the data structure is necessary to evaluate the three remaining calls to $take_\perp$.

In Figure 1d we see that each of the ferns α, β, γ, and δ contains some permutation of its original possible values, and $\perp$ has been pushed to the end of α. Furthermore, if there are no shared references to β, γ, and δ, the number of accessible pairs is linear in the length of the fern. If there are references to subferns, for a fern of size n, the worst case is $(n^2 + n)/2$. But, as these shared references vanish, so do the additional cons pairs.

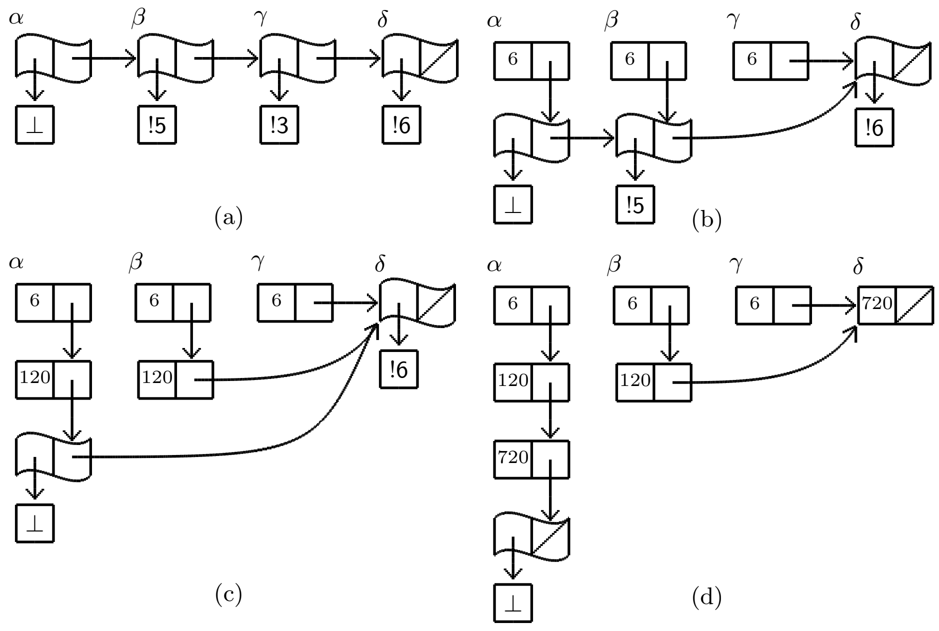

Figure 1. Fern α immediately after evaluation of cdrs, but before any cars have finished evaluation (a) and after the values, 6 (b), 120 (c), and 720 (d) have been promoted.

If **list** evaluated from right-to-left instead of evaluating from left-to-right, the example expression would return ((720 6 120) (720 6 120) (720 6) (720)). Each list would be independent of one another and the last pair of α would be a frons pair with $\perp$ in the car and the empty list in the cdr. This demonstrates that if there is sharing of these lists, the lists contain four pairs, three pairs, two pairs, and one pair, respectively. If the example expression just returned α, then only four pairs would be accessible.

The example presented in this section provides a direct view of promotion. When a fern is accessed by multiple computations, the promotion algorithm must be able to handle various issues such as multiple values becoming available for promotion at once. The code presented in the next section handles these details.

5 Ferns Implementation

In this section we present a complete, portable, R^{6}RS compliant implementation of ferns[2]. We begin with a description of *engines* [12], which we use to handle suspended, preemptible computations. We then describe and implement frons pairs, the building blocks of ferns. Next we present $car_\perp$ and $cdr_\perp$ which work on both frons pairs and cons pairs. Taking the $car_\perp$ of a frons pair involves non-deterministically choosing one of the possible values in the fern and promoting the chosen value. Taking the $cdr_\perp$ of a frons pair ensures the first value in the pair is determined and returns the second value (usually the rest of the fern). Taking the $car_\perp$ ($cdr_\perp$) of a cons pair is the same as taking its *car* (*cdr*).

[2] Our ferns library is available at `http://www.cs.indiana.edu/~webyrd/ferns.html`

5.1 Engines

An engine is a procedure that computes a delayed value in steps. To demonstrate the use of engines, consider the procedure

```
(define wait
  (λ_t (n)
    (cond
      ((zero? n) #t)
      (else (wait (− n 1))))))
```

To create an engine e to delay a call to (*wait* 20), we write:

```
   (define e (engine (wait 20)))
```

To partially compute (*wait* 20), we call e with a number of *ticks* and two handler procedures: (e 5 (λ (t v) (**list** t v)) (λ ($\hat{e}$) #f)) $\Rightarrow$ #f. This call runs the expression (*wait* 20) with 5 ticks. In our implementation, a tick is spent on each call to a procedure defined with λ_t. If the expression is not completely evaluated after all the ticks are spent, the second handler is called with a new engine containing the rest of the computation. In this case, we discard the new engine and return #f. We could, however, call the new engine to complete the computation. If the computation finishes, the first handler is called with the unspent ticks and the computed value. For example, consider

```
(let ((succeed (λ (t v) (list t v))))
  (letrec ((fail (λ (ê) (cons #f (ê 5 succeed fail)))))
    (e 5 succeed fail)))
⇒ (#f #f #f #f 4 #t).
```

In this example, (*wait* 20) calls *wait* a total of 21 times (including the initial call), so 4 ticks are unspent after the last call to $\hat{e}$.

The delayed computation in an engine may involve creating and calling more engines. When a *nested engine* [13] consumes a tick, every enclosing engine also consumes a tick. To see this, we can define the **amb** operator [14] using engines:

```
(define-syntax amb
  (syntax-rules ()
    ((_ exp_1 exp_2) (amb-aux (engine exp_1) (engine exp_2)))))


(define amb-aux
  (λ_t (e_1 e_2)
    (e_1 1 (λ (_ v) v) (λ (e_1) (amb-aux e_2 e_1)))))
```

Nested calls to **amb**, for example (**amb** v_1 (**amb** v_2 v_3)), rely on nestable engines. This implementation of **amb** is fair because our implementation of nested engines is fair: every tick given to the second engine in the outer call to *amb-aux* is passed on to exactly one of the engines, alternating between the engines for v_2 and v_3, in the inner call to *amb-aux*.

5.2 The Ferns Data Type

We represent a frons pair by a cons pair that contains at least one tagged engine (*te*). Engines are tagged with either L when locked (being advanced by another computation) or U when unlocked (runnable). We distinguish between locked and unlocked engines because the $car_\perp$ of a fern may be requested more than once simultaneously, for example in determining the $car_\perp$ of a fern whose values both depend on the $car_\perp$ of another fern: ($car_\perp$ (**list**$_\perp$ ($car_\perp$ *s*) ($cdar_\perp$ *s*))). To manage effects, we prevent the same engine from being advanced in more than one computation.

We define simple predicates $L_a?$, $U_a?$, $L_d?$, and $U_d?$ for testing whether one side of a frons pair contains a locked or unlocked engine.

```
(define engine-tag-compare
  (λ (get-te tag)
    (λ (q)
      (and (pair? q) (pair? (get-te q)) (eq? (car (get-te q)) tag)))))

(define La? (engine-tag-compare car 'L))
(define Ua? (engine-tag-compare car 'U))
(define Ld? (engine-tag-compare cdr 'L))
(define Ud? (engine-tag-compare cdr 'U))
```

The procedure $step_d$ ($step_a$) takes a frons pair with an unlocked tagged engine in the cdr (car) and locks and advances the tagged engine by *nsteps* ticks. If the engine does not finish, the tagged engine is unlocked and updated with the advanced engine. If the engine finishes with value *v*, then *v* becomes the frons pair's cdr (car). In addition, the tagged engine will be updated with an unlocked dummy engine that returns *v*. We do this because the cdrs of multiple frons pairs may share a single engine, as will be explained at the end of this section. Although the cars of frons pairs never share engines, we do the same for the cars.

```
(define step
  (λ (get-te set-val!)
    (λ (q)
      (let* ((te (get-te q)) (set-te! (λ (e) (set-car! te 'U) (set-cdr! te e))))
        (set-car! te 'L)
        (let ((e (cdr te)))
          (e nsteps (λ (_ v) (set-val! q v) (set-te! (engine v))) set-te!))))))

(define stepa (step car set-car!))
(define stepd (step cdr set-cdr!))
```

Now we present the implementation of the fern operators.

5.3 $\mathbf{cons}_\perp$, $car_\perp$, and $cdr_\perp$

cons$_\perp$ constructs a frons pair by placing unlocked engines of its unevaluated operands in a cons pair.

```
(define-syntax cons⊥
  (syntax-rules ()
    ((_ a d) (cons (cons 'U (engine a)) (cons 'U (engine d))))))
```

When the $car_\perp$ (definition below) of a fern is requested, parallel evaluation of the possible values is accomplished by a round-robin *race* of the engines in the fern. During its turn, each engine is advanced a fixed, arbitrary number of ticks until a value is produced. The race is accomplished by two mutually recursive functions: $race_a$, which works on the possible values of the fern, and $race_d$, which moves onto the next frons pair by either following the cdr of the current frons pair or starting again at the beginning.

$race_a$ dispatches on the current pair or value q. When the car of q is a locked engine, $race_a$ waits for it to become unlocked by waiting *nsteps* ticks and then calling $race_d$. The call to *wait* is required to allow $race_a$ to be preempted at this point, so the owner of the lock does not starve. When the car is an unlocked engine, $race_a$ advances the unlocked engine *nsteps* ticks, then continues the race by calling $race_d$. When q is not a pair, $race_a$ simply starts the race again from the beginning. This happens when racing over a finite fern and emerges from the **else** clause of $race_d$. When the car contains a value, that value is promoted to the front of all the subferns in the chain from p to q.

```
(define car⊥
  (λ (p)
    (letrec ((race_a
               (λ (q)
                 (cond
                   ((L_a? q) (wait nsteps) (race_d q))
                   ((U_a? q) (step_a q) (race_d q))
                   ((not (pair? q)) (race_a p))
                   (else (promote p q) (car p)))))
             (race_d
               (λ (q)
                 (cond
                   ((L_d? q) (race_a p))
                   ((U_d? q) (step_d q) (race_a p))
                   (else (race_a (cdr q)))))))
      (race_a p))))
```

One subtlety of $race_a$ is that it does not care, after advancing an engine *nsteps* ticks, whether that engine completed. The value will be picked up the next time the race comes around, if necessary. Calling *promote* immediately would be incorrect because an engine may be preempted while advancing, at which point promotion from p may be performed by another computation with a different value for the car of p.

$race_d$ also dispatches on q, this time examining its cdr. When the cdr of q is a locked engine, $race_d$, being unable to proceed further down the fern, restarts the race by calling $race_a$ on p. When the cdr of q contains an unlocked engine, $race_d$ advances the engine *nsteps* ticks as in $race_a$, and then restarts the race. If that engine finishes with a new frons pair, the new pair will then be competing in the race and will be examined next time around. When the cdr of q is a value, usually a fern, $race_d$ continues the race by passing it to $race_a$; if a non-pair value is at the end of a fern it will be picked up by the third clause in $race_a$.

$car_\perp$ avoids starvation by running each engine in a car for the same number of ticks. During a race, a subfern of the fern in question is in a fair state: up to a point, there are no engines in the cdrs, so each potential value in a car is considered equally. When this fair subfern is not the entire fern, the race devotes the same number of ticks to lengthening the fair subfern as it does to each element of that subfern. Since cdr engines often evaluate to pairs quickly, the entire fern usually becomes fair in a number of races equal to the length of the fern. When cdr engines do not finish quickly, however, the process of making the entire fern fair can take much longer, especially for long ferns. The cost of finding the value of an element occurring near the end of such a fern can be much greater than the cost for an element near the beginning.

promote (definition below) propagates a convergent value found in the pair q across the subfern from p, whose car was requested, to q. Each frons pair in this chain is transformed into a cons pair whose car is the convergent value and whose cdr is a copy of that frons pair. These new frons pairs are connected as a fern and terminate at the cdr of q. After promotion, the subferns from p to the cdr of q contain the convergent value in their cars and a fern containing the remaining possible values in their cdrs.

```
(define promote
  (λ (p q)
    (cond
      ((eq? p q) (cdr q))
      (else (let ((new-p (cons (car p) (promote (cdr p) q))))
              (set-car! p (car q))
              (set-cdr! p new-p)
              new-p)))))
```

The cdr of a fern (definition below) cannot be determined until the fern's car has been determined. Once the car has been determined, there is no longer parallel competition between potential cdrs. Thus, we can use cdr_s, which takes the cdr of a stream. Then, since q's car has been determined, q has therefore become a cons pair, so $cdr_\perp$ returns the value in q's cdr. (car_s's definition follows by replacing all ds by as. $\mathbf{cons}_s$ is the same as $\mathbf{cons}_\perp$, and the definitions of the other stream operators: $\mathbf{mplus}_s$, $mplus\text{-}aux_s$, $bind_s$, $\mathbf{disj}_s$, and $\mathbf{conj}_s$ follow the definitions of Section 3 with operators $f_\perp$ replaced by f_s.)

```
(define cdr⊥ (λ (q) (car⊥ q) (cdrs q)))

(define cdrs
  (λ (q)
    (cond
      ((Ld? q) (wait nsteps) (cdrs q))
      ((Ud? q) (stepd q) (cdrs q))
      (else (cdr q)))))
```

If the engine being advanced by $cdr_\perp$ completes, $cdr_\perp$ indicates that $step_d$ should replace the tagged engine in p by the computed value. The value will be picked up after *loop* terminates. $cdr_\perp$ also indicates that the pair representing the tagged engine should be unlocked and the engine should be replaced with a new engine that simply returns the value.

$race_d$ and $cdr_\perp$ are required to not only update the frons pair with the calculated value, but also they must update the tagged engine because there might be a fern other than p sharing this engine. Consider the following expression where we assume **list** evaluates its arguments from left to right.

```
(let ((β (cons⊥ 1 (ints-from⊥ 2))))
  (let ((α (cons⊥ ⊥ β)))
    (list (car⊥ α) (cadr⊥ β) (cadr⊥ α))))

⇝ (1 2 2)
```

Figure 2 below shows the data structures involved in evaluating the expression. Figure 2a shows α immediately after it has been constructed, with engines delaying evaluation of $\perp$ and β. In evaluating $(car_\perp\ \alpha)$, the engine for β finishes, resulting in Figure 2b. β can now participate in the race for $(car_\perp\ \alpha)$. Suppose the value 1 found in the car of β is chosen and promoted from α to β. The result is Figure 2c, in which the engine delaying (*ints-from*$_\perp$ 2) is shared by both β and the cdr of α. $(cadr_\perp\ \beta)$ forces calculation of (*ints-from*$_\perp$ 2), which results in a fern, γ, whose first value (in this example) is 2. Figure 2d now shows why $step_d$ updates the current pair (β) and creates a new engine with the calculated value (γ): the cddr of α needs the new engine to avoid recalculation of (*ints-from*$_\perp$ 2). In Figure 2e when $(cadr_\perp\ \alpha)$ is evaluated, the value 2, calculated already by $(cadr_\perp\ \beta)$, is promoted and the engine delaying (*ints-from*$_\perp$ 3) is shared by both α and β.

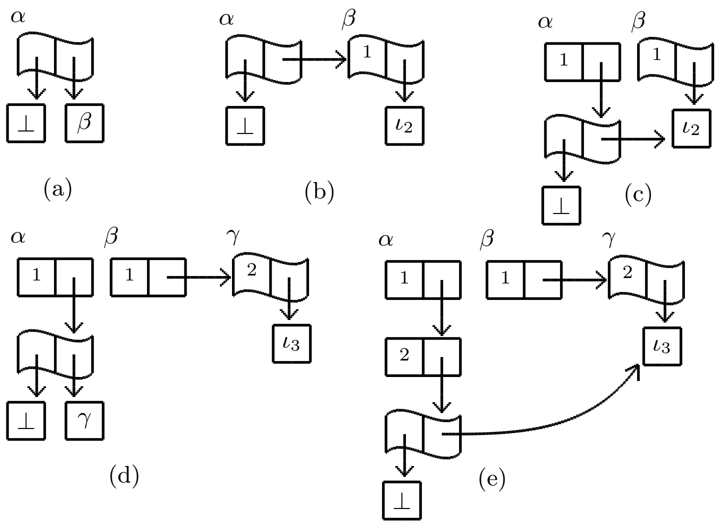

Figure 2. Fern α after construction (a); after β in the cdr of α has been evaluated (b); after 1 from the car of β has been promoted to the car of α, resulting in a shared tagged engine (c); after the shared engine is run, while evaluating $(cadr_\perp\beta)$, to produce a fern α (d); after 2 from the car of γ has been promoted to the cadr of α (e).

6 Related Work

Previous implementations of ferns have been for a call-by-need language. The work of Friedman and Wise [3, 4, 1] differs from our shallow embedding in two major respects. First, the languages were implemented using an interpreter. Second, the laziness of the *cons* operator resulted in the operands to *cons* competing in the race described in Section 5.3.

Johnson's master's thesis [15] under Friedman's direction presents an interpreter implemented in Pascal for a lazy ferns language. Subsequently, Johnson and his doctoral student Jeschke implemented a series of native C symbolic multiprocessing systems based on the Friedman and Wise model. This series culminated with the parallel implementation Jeschke describes in his dissertation [8]. In their *Daisy* language, ferns are the means of expressing explicit concurrency [5, 6]. This work differs from our shallow embedding approach in the same ways as the earlier work of Friedman and Wise.

7 Conclusion

We have presented a shallow embedding of ferns, which are shareable, bottom-avoiding data structures. Ferns are useful in avoiding divergence; elements that do not converge are avoided until it is no longer possible to do so. Ferns are shareable, meaning that many different computations can use the elements of a fern without repeating computation. Since the fern constructor $\mathbf{cons}_\perp$ is similar to the constructor *cons*, writing bottom-avoiding functions is often intuitive.

We have also presented some motivating examples for ferns, including a generalization of logic programming combinators. We have shown that fern-based combinators avoid more types of divergence than stream-based combinators.

Ferns were originally conceptualized as a data structure for the abstraction of multiprogramming, and were specified by a formal characterization rather than a concrete implementation. Future research includes proving the correctness and fairness of our implementation, proving completeness of the fern-based combinators, and implementing a multicore shallow embedding of ferns. We hope that the relative simplicity of our work along with the ease of defining bottom-avoiding recursive functions will encourage others to take up some of these challenges.

8 Acknowledgements

Guy Steele Jr.'s inspiring keynote address at Dan Friedman's 60th birthday celebration renewed our interest in ferns. In addition, Guy made a critical observation that simplified *promote*, which in turn allowed us to simplify $car_\perp$ and $cdr_\perp$. We thank Kevin Millikin, Olivier Danvy, Mitch Wand, Steve Johnson, Chung-chieh Shan, and Oleg Kiselyov for their comments on drafts of this paper. Oleg's clever use of thunks to represent "incomplete" computations in miniKanren led to our interest in avoiding divergence whenever possible. Once again, we have found Dorai Sitaram's excellent SLaTeX package invaluable for typesetting our code.

References

1. Friedman, D.P., Wise, D.S.: Fancy ferns require little care. In Holmström, S., Nordström, B., Wikström, Å., eds.: Symposium on Functional Languages and Computer Architecture, Göteborg, Sweden, Laboratory for Programming Methodology, University of Göteborg and Chalmers University of Technology (June 1981) 124–156
2. Sperber, M., Dybvig, R.K., Flatt, M., van Straaten, A. (eds.): Revised^6 report on the algorithmic language Scheme (September 2007)
3. Friedman, D.P., Wise, D.S.: An approach to fair applicative multiprogramming. In Kahn, G., ed.: Semantics of Concurrent Computation: Proceedings of the International Symposium. Volume 70 of Lecture Notes in Computer Science (LNCS)., Evian, France, Springer-Verlag (Berlin/Heidelberg/New York) (July 1979) 203–225
4. Friedman, D.P., Wise, D.S.: An indeterminate constructor for applicative programming. In: Conference Record of the Seventh ACM Symposium on Principles of Programming Languages (POPL '80), New York, USA, ACM Press (January 1980) 245–250
5. Johnson, S.D.: Circuits and systems: Implementing communication with streams. IMACS Transactions on Scientific Computation, Vol. II (1983) 311–319
6. Johnson, S.D., Jeschke, E.: Modeling with streams in Daisy/The SchemEngine Project. In Sheeran, M., Melham, T., eds.: Designing Correct Circuits, (DCC'02), ETAPS 2002 (2002) Presentation at the Workshop on Designing Correct Circuits, held on 6–7 April 2002 in Grenoble, France.
7. Filman, R.E., Friedman, D.P.: Coordinated Computing: Tools and Techniques for Distributed Software. McGraw-Hill (1984)
8. Jeschke, E.R.: An Architecture for Parallel Symbolic Processing Based on Suspending Construction. PhD thesis, Indiana University Computer Science Department (May 1995) Technical Report No. 445, 152 pages.
9. Wand, M., Vaillancourt, D.: Relating models of backtracking. In: Proc. 9th Int. Conf. on Functional Programming, ACM Press (2004) 54–65
10. Spivey, M., Seres, S.: Combinators for logic programming. The Fun of Programming (2003) 177–200
11. Lloyd, J.W.: Foundations of logic programming. Second extended edn. Springer-Verlag, New York (1987)
12. Haynes, C.T., Friedman, D.P.: Abstracting timed preemption with engines. Journal of Computer Languages **12**(2) (1987) 109–121
13. Hieb, R., Dybvig, K., Anderson, III, C.W.: Subcontinuations. Lisp and Symbolic Computation **7**(1) (1994) 83–110
14. McCarthy, J.: A basis for a mathematical theory of computation. In Braffort, P., Hirschberg, D., eds.: Computer Programming and Formal Systems, North Holland (1963) 33–69
15. Johnson, S.D.: An interpretive model for a language based on suspended construction. Master's thesis, Indiana University Computer Science Department (1977) Indiana University Computer Science Department Technical report No. 68.
16. Dybvig, R.K., Hieb, R.: Engines from continuations. Comput. Lang **14**(2) (1989) 109–123
17. Baader, F., Snyder, W.: Unification theory. In Robinson, A., Voronkov, A., eds.: Handbook of Automated Reasoning. Volume I. Elsevier Science (2001) 445–532

A Nestable Engines

Our implementation of ferns requires nestable engines [16, 13], which we present here with minimal comment. We use two global variables: *timer*, which holds the number of ticks available to the currently running engine and uses #f to represent infinity, and *throw*, which holds an escape continuation. *make-engine* makes an engine out of a thunk. **engine** is a macro that makes an engine from expressions. λ_t is like λ except that it passes its body, as a thunk, to *expend-tick-to-call*, which ensures a tick is spent before the body is evaluated and passes the suspended body to *throw* if no ticks are available.

```
(define timer #f)
(define throw 0)
(define make-engine
  (λ (thunk)
    (λ (ticks sk fk)
      (let* ((gift (or (and timer (min timer ticks)) ticks))
             (saved-timer (and timer (− timer gift)))
             (saved-throw throw)
             (caught (call-with-current-continuation
                       (λ (k)
                         (set! timer gift)
                         (set! throw k)
                         (let ((result (thunk)))
                           (throw (cons timer result)))))))
        (set! timer saved-timer)
        (set! throw saved-throw)
        (let ((owed (− ticks gift)))
          (cond
            ((pair? caught)
             (and timer (set! timer (+ timer (car caught))))
             (sk (+ (car caught) owed) (cdr caught)))
            (else (let ((e (make-engine caught)))
                    (if (zero? owed) (fk e)
                      (let ((th (λ () (e owed sk fk))))
                        ((call-with-current-continuation
                           (λ (k̂) (throw (λ () (k̂ th))))))))))))))))

(define-syntax engine
  (syntax-rules ()
    ((_ e) (make-engine (λ () e)))))

(define-syntax λt
  (syntax-rules ()
    ((_ formals b0 b ...) (λ formals (expend-tick-to-call (λ () b0 b ...))))))
```

```
(define expend-tick-to-call
  (λ (thunk)
    ((call-with-current-continuation
       (λ (k)
         (let th ()
           (case timer
             ((#f) (k thunk))
             ((1) (throw (λ () (k thunk))))
             ((0) (throw th))
             (else (set! timer (− timer 1)) (k thunk)))))))))
```

B Logic Programming Auxiliaries

To complete the implementation of the logic programming combinators presented in Section 3 we provide a logic variable constructor *make-var*, a unification algorithm *unify*, and substitution helpers *empty-σ*, *ext-σ*, and *walk*. We represent logic variables by R^{6}RS [2] records (syntactic layer); defining the record type *var* creates the constructor *make-var* automatically. We represent substitutions as association lists, and use the triangular substitution model [17].

```
(define-record-type var)

(define unify
  (λ (t1 t2 σ)
    (let ((t1 (walk t1 σ)) (t2 (walk t2 σ)))
      (cond
        ((eq? t1 t2) σ)
        ((var? t1) (ext-σ t1 t2 σ))
        ((var? t2) (ext-σ t2 t1 σ))
        ((and (pair? t1) (pair? t2))
         (let ((σ (unify (car t1) (car t2) σ)))
           (and σ (unify (cdr t1) (cdr t2) σ))))
        (else (if (equal? t1 t2) σ #f))))))

(define empty-σ '())

(define ext-σ
  (λ (x t σ)
    `((,x . ,t) . ,σ)))

(define walk
  (λ (t σ)
    (let ((b (assq t σ)))
      (if b (walk (cdr b) σ) t))))
```