Higher-Order and Symbolic Computation manuscript No.
(will be inserted by the editor)

A Shallow Scheme Embedding of ⊥-Avoiding Streams

William E. Byrd · Daniel P. Friedman · Ramana Kumar · Joseph P. Near

the date of receipt and acceptance should be inserted later

Abstract We present a shallow embedding of *ferns*, stream-like bottom-avoiding, shareable, and implicitly-parallel data structures proposed in 1979. The embedding is written as a portable library written in a small subset of R^{6}RS Scheme. The 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.

Keywords Ferns · Frons · Streams · Bottom-avoidance · Logic combinators

1 Introduction

We present a complete *shallow embedding* [2] of *ferns* [9], a shareable data structure designed to avoid divergence. Being a shallow embedding distinguishes the implementation from a *deep embedding*, where abstract syntax is interpreted. The embedding is in Scheme and is available as a portable R^{6}RS Scheme [18] library. In addition, we present an extended example demonstrating the power of ferns. In this example we provide a bottom-avoiding generalization of stream-based logic combinators.

Ferns are constructed with *cons* and $\mathbf{cons}_\perp$, originally called **frons** [8], and accessed by $car_\perp$ and $cdr_\perp$, generalizations of *car* and *cdr*, respectively. Ferns built with $\mathbf{cons}_\perp$ 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.

There have been several implementations of ferns [7,13,8,9,14,15,12], but each implementation has been a deep embedding, and each language has assumed its implementation was over an arbitrary number of processors. In our shallow embedding, we model the parallelism that inspired the ferns concept using first-class preemptible computations called *engines* [10], which in turn were inspired by the desire to have a tool to implement arbitrary multi-processing primitives. The modelling of the parallelism in the deep embeddings was done by an operator, *coax* [7], which functionally

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

advanced a suspension [5]. Thus, engines are a first-class manifestation of suspensions. Although the preemptible computations model doesn't capture all of the ramifications of the original ferns, the shallow embedding of ferns developed atop the shallow embedding of engines [11,3] reveals enough of the intricacies so that ferns can be rather straightforwardly adapted to other architectures.

The rest of the paper is as follows. Section 2 shows examples of familiar recursive functions using ferns. Section 3 defines generalized logic programming combinators. Section 4 describes the promotion algorithm [7] that characterizes the necessary sharing properties of ferns. Section 5 presents the definitions of $\mathbf{cons}_\perp$, $car_\perp$, and $cdr_\perp$. The remainder includes some related work, conclusions, and appendices to make the presentation self-contained.

2 Examples

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

2.1 Two Simple Programs

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

```
(let ((s (cons⊥ 0 (cons⊥ 1 '()))))
  (display (car⊥ s)) (display (cadr⊥ s)) (display (car⊥ 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 ((s1 (cons⊥ (! 6) ⊥)) (s2 (cons⊥ ⊥ (cons⊥ (! 5) ⊥))))
  (cons (car⊥ s1) (car⊥ s2)))
```

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

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. The second expression might instead return (2 1 2) however; 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 a stream 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 unbounded fern: the value of (*caddr*$_\perp$ (*map*$_\perp$ *add*$_1$ (*ints-from*$_\perp$ 0))) can be any positive integer.

Ferns work especially well with *annihilators*. True values are annihilators for *or*$_\perp$

```
(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 avoids divergence if it finds one: (*or*$_\perp$ (**list**$_\perp$ ⊥ (*odd?* 1) (! 5) ⊥ (*odd?* 0))) returns some true value, where **list**$_\perp$ is defined as follows.

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

Let's define *append*$_\perp$ for ferns.

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

To observe the behavior of *append*$_\perp$, we define *take*$_\perp$ whose first argument is either #f (all results) or $n > 0$ (no more than n results).

```
(define take⊥
  (λt (n s)
    (cond
      ((null? s) '())
      (else (cons (car⊥ s)
                  (if (and n (= n 1)) '() (take⊥ (and n (- n 1)) (cdr⊥ s)))))))))
```

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

When determining the nth value, it is necessary to avoid taking the $cdr_\perp$ after the nth value is determined, since it is that $cdr_\perp$ that might not terminate and we already have n results.

The definition of $append_\perp$ appears to work as expected:

$(take_\perp\ 2\ (append_\perp\ (\mathbf{list}_\perp\ 1)\ (\mathbf{list}_\perp\ \perp\ 2))) \Rightarrow (1\ 2)$.

Moving $\perp$ from the second argument to the first, however, reveals a problem:

$(take_\perp\ 2\ (append_\perp\ (\mathbf{list}_\perp\ \perp\ 1)\ (\mathbf{list}_\perp\ 2))) \Rightarrow \perp$.

Even though 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 the 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. With the definition of $\mathbf{mplus}_\perp$ in Section 3.1, it becomes clear that 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⊥ (λt (e) (cons (car⊥ s1) e)) s2)
                    (Cartesian-product⊥ (cdr⊥ s1) s2))))))
```

$(take_\perp\ 6\ (Cartesian\text{-}product_\perp\ (\mathbf{list}_\perp\ \perp\ \texttt{'a}\ \texttt{'b})\ (\mathbf{list}_\perp\ \texttt{'x}\ \perp\ \texttt{'y}\ \perp\ \texttt{'z})))$

$\rightsquigarrow$ ((a . x) (a . y) (b . x) (a . z) (b . y) (b . z))

where $\rightsquigarrow$ 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

$(take_\perp\ 2\ (Cartesian\text{-}product_\perp\ (\mathbf{list}_\perp\ (\mathbf{begin}\ (display\ \#\texttt{t})\ 5))\ (\mathbf{list}_\perp\ \texttt{'a}\ \perp\ \texttt{'b})))$

$\rightsquigarrow$ ((5 . a) (5 . b))

prints #t *exactly once*. (There are more examples of the use of ferns [14,15,4,12].)

3 Extended Example: New Logic Combinators

In this section, we use 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 $\mathbf{mplus}_\perp$ and $bind_\perp$ over ferns[2], and go on to implement logic programming combinators in terms of these operators. The fern-based

[2] $\mathbf{mplus}_\perp$ appends two ferns, while $bind_\perp$ is Common Lisp's *mapcan* [20] defined over ferns. The names "mplus" and "bind" are taken from Haskell [17].

logic combinators are shown to be more general than the standard stream-based ones. (See Wand and Vaillancourt's historical account of logic combinators [21].)

3.1 **mplus**$_\perp$ and *bind*$_\perp$

To develop logic programming combinators in a call-by-value language, we make **mplus**$_\perp$ itself lazy to avoid diverging when one or more of its arguments diverge. This is accomplished 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⊥ (cadr⊥ p) (cdar⊥ p)))))))

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

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*$_\perp$ 13 (*bind*$_\perp$ (*ints-from*$_\perp$ 0) *ints-from*$_\perp$)) $\rightsquigarrow$ (0 1 2 1 3 4 5 6 7 8 9 2 10).

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 multiple unbounded ferns, such as this program inspired by Seres and Spivey [19] that searches for a pair a and b of divisors of 9 by enumerating the integers from 2 in a fern of possible values for a and similarly for b:

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

Using streams instead of ferns in this example, which would be like nesting "for" loops, would result in divergence since 2 does not evenly divide 9.

3.2 Implementation of Logic Programming Combinators

The combinators comprise three *goal constructors*: $\equiv_\perp$, which unifies terms; **disj**$_\perp$, which performs disjunction over goals; and **conj**$_\perp$, which performs conjunction 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)))))
```

A logic program evaluates to a goal; to obtain answers, this goal is 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. (See Appendix B for the rest of the definitions used in this section.)

Given two logic variables x and y, here are some simple logic programs that produce the same answers using both fern-based and stream-based combinators.

```
(run⊥ #f (≡⊥ 1 x)) ⇒ ({x/1})
(run⊥ 1 (conj⊥ (≡⊥ y 3) (≡⊥ x y))) ⇒ ({x/3, y/3})
(run⊥ 1 (disj⊥ (≡⊥ x y) (≡⊥ y 3))) ⇒ ({x/y})
(run⊥ 5 (disj⊥ (≡⊥ x y) (≡⊥ y 3))) ⇒ ({x/y} {y/3})
(run⊥ 1 (conj⊥ (≡⊥ x 5) (conj⊥ (≡⊥ x y) (≡⊥ y 4)))) ⇒ ()
(run⊥ #f (conj⊥ (≡⊥ x 5) (disj⊥ (≡⊥ x 5) (≡⊥ x 6)))) ⇒ ({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:

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

and given idempotent substitutions [16], the fern-based combinators can even avoid some circularity-based divergence without the occurs-check, while stream-based combinators cannot:

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

There are functions that represent relations. The relation *always-five*$_\perp$ 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*$_\perp$ x), obtaining an unbounded collection of answers. Using *run*$_\perp$, 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. This is because the definition of *always-five*$_\perp$ is left recursive. The fern-based combinators, however, compute as many answers as desired:

$(run_\perp\ 4\ (always\text{-}five_\perp\ x)) \Rightarrow (\{x/5\}\ \{x/5\}\ \{x/5\}\ \{x/5\})$.

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 [7]. The values in a fern are computed and *promoted* across the fern while ensuring that the correct values are available from each subfern, $\perp$'s are avoided, and non-$\perp$ 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))))))
```

$\rightsquigarrow$ ((6 120 720) (6 120 720) (6 720) (720))

assuming **list** evaluates its arguments left-to-right. Importantly, 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 along with the necessary changes to the fern

data structure during this process. Whenever we encounter a 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 $(car_{\perp}\ \alpha)$ is evaluated. We assume that, of the possible values for $(car_{\perp}\ \alpha)$, 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 $(car_{\perp}\ \beta)$ and $(car_{\perp}\ \gamma)$, 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 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 $(cadr_{\perp}\ \alpha)$ is evaluated. This time, (! 5) is chosen from $\perp$, (! 5), and (! 6). Since the value of (! 5) is also a possible value for $(cadr_{\perp}\ \beta)$, 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 and a pointer to δ. The cddr of β points to δ; no new fern with remaining possible values is needed because the value chosen for $(cadr_{\perp}\ \beta)$ was the first value available. As before, the pairs containing values become cons pairs.

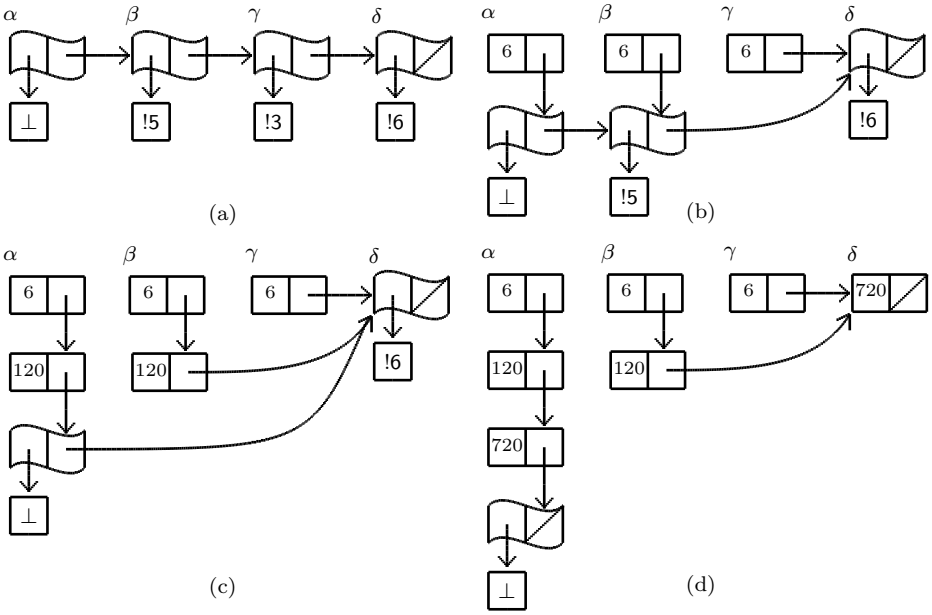

Figure 1 Fern α immediately after evaluation of cdrs, but before any cars have finished evaluation (a) and after the values, 6 (b), 120 (c), and 720 (d) have been promoted.

Figure 1d depicts the data structure after $(caddr_{\perp}\ \alpha)$ 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 $(car_{\perp}\ \delta)$ (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 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.

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 the others 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 Implementation of Ferns

In this section we present a complete, portable, R^{6}RS compliant implementation of ferns.[3] We begin with a description of *engines* [10], 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 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 rest of the fern. Taking the $car_\perp$ ($cdr_\perp$) of a cons pair is the same as taking its *car* (*cdr*).

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
  (λt (n)
    (cond
      ((zero? n) 'done)
      (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*: (e 5), which returns either a pair with false in the car and a new advanced engine (one advanced 5 ticks)

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

in the cdr or a pair with unused ticks (always true) in the car and the value of the computation (here done) in the cdr. In our embedding of engines, a tick is spent on each call to a procedure defined with λ_t. Consider

```
(let loop ((p (e 5)))
  (cons (car p) (if (car p) (list (cdr p)) (loop ((cdr p) 5)))))
    ⇒ (#f #f #f #f 4 done).
```

In this example, (*wait* 20) calls *wait* a total of 21 times (including the initial call), so on the fifth engine invocation, it terminates with 4 unused ticks.

The delayed computation in an engine may involve creating and calling more engines. When a *nested engine* [11] consumes a tick, every frons-enclosing engine also consumes a tick. To see this, we define $\mathbf{choose}_\perp$ using engines:

```
(define-syntax choose⊥
  (syntax-rules ()
    ((_ exp1 exp2) (choose-aux⊥ (engine exp1) (engine exp2)))))

(define choose-aux⊥
  (λt (e1 e2)
    (let ((p (e1 1)))
      (if (car p) (cdr p) (choose-aux⊥ e2 (cdr p))))))
```

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

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. Thus, to manage effects, the locks prevent the same engine from being advanced in more than one computation.[4]

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))
```

[4] A lock creates a localized critical region that corresponds to the intended use of *sting-unless* [6].

The procedure $coax_d$ ($coax_a$) takes a frons pair with an unlocked tagged engine in the cdr (car) and locks and advances the tagged engine by *nsteps* ticks. If *coax*ing [7] the engine does not finish, the tagged engine is unlocked and updated with the advanced engine. If *coax*ing 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 coaxer
  (λ (get-te set-val!)
    (λ (q)
      (let ((te (get-te q)))
        (set-car! te 'L)
        (let ((p (coax (cdr te))))
          (let ((b (car p)) (v (cdr p)))
            (when b (set-val! q v))
            (replace! te 'U (if b (engine v) v))))))))

(define coax (λ (e) (e nsteps)))
(define coax_a (coaxer car set-car!))
(define coax_d (coaxer cdr set-cdr!))

(define replace!
  (λ (p a d)
    (set-car! p a)
    (set-cdr! p d)))
```

Now we present the implementation of the fern operators.

5.3 **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) 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.

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

$race_a$ 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, we call *promote* which ensures a value is promoted to the car of p, then return that value.

One subtlety of the definition of $race_a$ is that after coaxing an engine it does not check if the coaxing has led to completion. If it has, 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 subfern for the same number of ticks. During a race, a subfern of the fern in question is in a fair state: for some (potentially empty) prefix of the subfern there are no engines in the cdrs, so each potential value in a fair subfern 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.

Starting from p, *promote* (definition below) finds the first pair r whose car contains a convergent value, and propagates that value back to p. Each frons pair in this chain (excluding r) is transformed into a cons pair whose car is the convergent value. These new frons pairs are connected as a fern and the last one shares r's cdr. When *promote* is called from $race_a$, we know that q's car is a value but we don't know for certain that there is no other pair, say r, in the chain from p to q. Thus, we must search from p without preemption to find the closest value to p. This situation can arise when there are two calls to $car_\perp$ on the same fern competing: for example,

```
(let ((α (list⊥ (! 5) (! 6))))
  (car⊥ (list⊥ ⊥ (car⊥ α) ⊥ (car⊥ α) ⊥)))
```

If a call to $race_a$ finds the value 720 and tries to promote it, but the value 120 has already been promoted, we don't want to change the car of α. Instead, the call to *promote* when 720 is found will find the 120 first and stop.

```
(define-syntax let_t
  (syntax-rules ()
    ((_ ((x e) ...) b0 b ...) ((λ_t (x ...) b0 b ...) e ...))))

(define promote
  (λ_t (p)
    (cond
      ((La? p) (wait nsteps) (promote p))
      ((Ua? p)
       (set-car! (car p) 'L)
       (let_t ((te (car p)))
         (let_t ((r (promote (cdr p))))
           (replace! p (car r) (cons te (cdr r)))
           (set-car! te 'U)
           p)))
      (else p))))
```

The $cdr_\perp$ of a fern (definition below) cannot be determined until the fern's $car_\perp$ 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 p's car has been determined, p has therefore become a cons pair, so $cdr_\perp$ returns the value in p'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 follow the definitions with operators $f_\perp$ replaced by f_s.)

```
(define cdr⊥ (λ_t (p) (car⊥ p) (cdrs p)))

(define cdrs
  (λ_t (p)
    (cond
      ((Ld? p) (wait nsteps) (cdrs p))
      ((Ud? p) (coaxd p) (cdrs p))
      (else (cdr p)))))
```

If the engine being advanced by cdr_s completes, cdr_s indicates that $coax_d$ should replace the tagged engine in p by the computed value. $race_d$ and $cdr_\perp$ are required not only to update the frons pair with the calculated value, however but also to 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**$_\perp$ 1 (*ints-from*$_\perp$ 2))))
  (**let** ((α (**cons**$_\perp$ $\perp$ β)))
    (**list** (*car*$_\perp$ α) (*cadr*$_\perp$ β) (*cadr*$_\perp$ α))))

$\rightsquigarrow$ (1 2 2)

Figure 2 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*$_\perp$ α), the engine for β finishes, resulting in Figure 2b. β can now participate in the race for (*car*$_\perp$ α). Suppose the value 1 found in the car of β is chosen and promoted. 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*$_\perp$ β) 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 $coax_d$ updates the current pair (β) and creates a new dummy 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*$_\perp$ α) is evaluated, the value 2, calculated already by (*cadr*$_\perp$ β), is promoted and the engine delaying (*ints-from*$_\perp$ 3) is shared by both α and β.

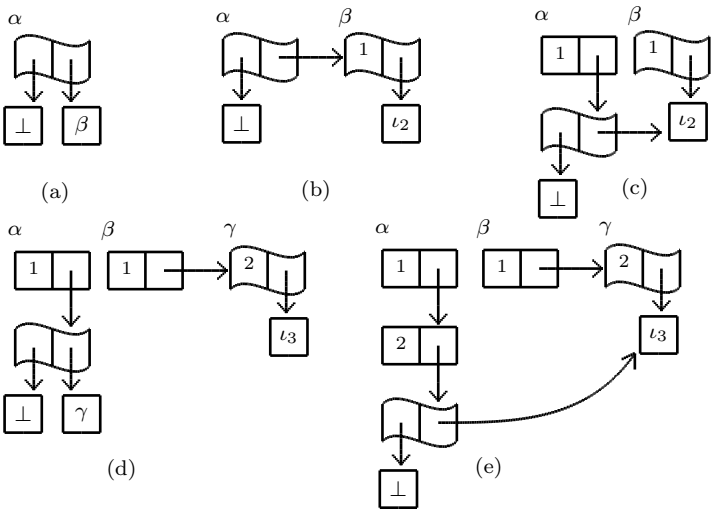

Figure 2 Fern α after construction (a); after β in the cdr of α has been evaluated (b); after 1 from the car of β has been promoted to the car of α, resulting in a shared tagged engine (c); after the shared engine is run, while evaluating (*cadr*$_\perp\beta$), to produce a fern γ (d); after 2 from the 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 [7–9] presumes a deep embedding whereas this approach is a shallow embedding. The function *coax* is taken from their conceptualization [7]:

> COAX is a function which takes a suspension as an argument and returns a field as a value; that field may have its *exists* bit *true* and its pointer referring to its *exist*ent value, or it may have its *exists* bit false and its pointer referring to another suspension.

Thus, engines are a user-level, first-class manifestation of suspensions where *true* above corresponds to the unused ticks. Johnson's master's thesis [13] under Friedman's direction presents a deep embedding 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 [12]. In their *Daisy* language, ferns are the means of expressing explicit concurrency [14, 15].

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 this embedding and implementing a multicore shallow embedding of ferns. We hope that the relative simplicity of this 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'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, Kent Dybvig, Chung-chieh Shan, and Oleg Kiselyov for their comments on drafts of this paper. Once again, we have found Dorai Sitaram's excellent SLaTeX package invaluable for typesetting programs.

References

1. Baader, F., Snyder, W.: Unification theory. In: A. Robinson, A. Voronkov (eds.) Handbook of Automated Reasoning, vol. I, chap. 8, pp. 445–532. Elsevier Science (2001)
2. Boulton, R., Gordon, A., Gordon, M., Harrison, J., Herbert, J., Tassel, J.V.: Experience with embedding hardware description languages in HOL. In: Theorem Provers in Circuit Design: Proceedings of the IFIP TC10/WG 10.2 International Conference, pp. 129–156. North-Holland (1992)
3. Dybvig, R.K., Hieb, R.: Engines from continuations. Comput. Lang **14**(2), 109–123 (1989)
4. Filman, R.E., Friedman, D.P.: Coordinated Computing: Tools and Techniques for Distributed Software. McGraw-Hill (1984)
5. Friedman, D.P., Wise, D.S.: CONS should not evaluate its arguments. In: S. Michaelson, R. Milner (eds.) Third International Colloquium on Automata, Languages and Programming, pp. 257–284. Edinburgh University Press, University of Edinburgh (1976)
6. Friedman, D.P., Wise, D.S.: Sting-unless: a conditional, interlock-free store instruction. In: 16th Annual Allerton Conf. on Communication, Control, and Computing, pp. 578–584. University of Illinois, Urbana-Champaign (1978)
7. Friedman, D.P., Wise, D.S.: An approach to fair applicative multiprogramming. In: G. Kahn (ed.) Semantics of Concurrent Computation: Proceedings of the International Symposium, *Lecture Notes in Computer Science (LNCS)*, vol. 70, pp. 203–225. Springer-Verlag (Berlin/Heidelberg/New York), Evian, France (1979)
8. 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), pp. 245–250. ACM Press, New York, USA (1980)
9. Friedman, D.P., Wise, D.S.: Fancy ferns require little care. In: S. Holmström, B. Nordström, Å. Wikström (eds.) Symposium on Functional Languages and Computer Architecture, pp. 124–156. Laboratory for Programming Methodology, University of Göteborg and Chalmers University of Technology, Göteborg, Sweden (1981)
10. Haynes, C.T., Friedman, D.P.: Abstracting timed preemption with engines. Journal of Computer Languages **12**(2), 109–121 (1987)
11. Hieb, R., Dybvig, K., Anderson, III, C.W.: Subcontinuations. Lisp and Symbolic Computation **7**(1), 83–110 (1994)
12. Jeschke, E.R.: An architecture for parallel symbolic processing based on suspending construction. Ph.D. thesis, Indiana University Computer Science Department (1995). URL `http://www.cs.indiana.edu/Research/techreports/`. Technical Report No. 445, 152 pages.
13. Johnson, S.D.: An interpretive model for a language based on suspended construction. Master's thesis, Indiana University Computer Science Department (1977). URL `http://www.cs.indiana.edu/Research/techreports/`. Indiana University Computer Science Department Technical report No. 68
14. Johnson, S.D.: Circuits and systems: Implementing communication with streams. IMACS Transactions on Scientific Computation, Vol. II pp. 311–319 (1983)
15. Johnson, S.D., Jeschke, E.: Modeling with streams in Daisy/The SchemEngine Project. In: M. Sheeran, T. Melham (eds.) Designing Correct Circuits, (DCC'02). ETAPS 2002 (2002). URL `http://www.math.chalmers.se/~ms/DCC02/`. Presentation at the Workshop on Designing Correct Circuits, held on 6–7 April 2002 in Grenoble, France
16. Lloyd, J.W.: Foundations of logic programming, second extended edn. Springer-Verlag, New York (1987)
17. Peyton Jones, S., et al.: The Haskell 98 language and libraries: The revised report. Journal of Functional Programming **13**(1), 0–255 (2003). `http://www.haskell.org/definition/`
18. Sperber, M., Dybvig, R.K., Flatt, M., van Straaten, A. (eds.): Revised^6 report on the algorithmic language Scheme (2007). URL `http://www.r6rs.org/`
19. Spivey, M., Seres, S.: Combinators for logic programming. The Fun of Programming pp. 177–200 (2003)
20. Steele, G.L.: Common Lisp: The Language, Second Edition. Digital Press, Bedford, Massachusetts (1990)
21. Wand, M., Vaillancourt, D.: Relating models of backtracking. In: Proc. 9th Int. Conf. on Functional Programming, pp. 54–65. ACM Press (2004)

A Nestable Engines

This embedding of ferns requires nestable engines [3,11], which we present here with minimal comment. The implementation uses a global variable, *state*, which holds two values: the number of ticks available to the currently running engine or #f representing infinity; and a continuation. *make-engine* makes an engine out of a thunk. **engine** is a macro that makes an engine from an expression. 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 the continuation if no ticks are available. Programs that use this embedding of nestable engines (and by extension our embedding of $\mathbf{cons}_\perp$) should not use *call/cc*, because the uses of *call/cc* in the nestable engines implementation may interact with any other uses in ways that are difficult for the programmer to predict.

```
(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 state (cons #f 0))

(define expend-tick-to-call
  (λ (thunk)
    ((call/cc
       (λ (k)
         (let th ()
           (cond
             ((not (car state)) (k thunk))
             ((zero? (car state)) ((cdr state) th))
             (else (set-car! state (- (car state) 1)) (k thunk)))))))))

(define make-engine
  (λ (thunk)
    (λ (ticks)
      (let* ((gift (if (car state) (min (car state) ticks) ticks))
             (saved-state (cons (and (car state) (- (car state) gift)) (cdr state)))
             (caught (call/cc
                       (λ (k)
                         (replace! state gift k)
                         (let ((result (thunk)))
                           ((cdr state) (cons (car state) result)))))))
        (replace! state (car saved-state) (cdr saved-state))
        (let ((owed (- ticks gift)))
          (cond
            ((pair? caught)
             (and (car state) (set-car! state (+ (car state) (car caught))))
             (cons (+ (car caught) owed) (cdr caught)))
            (else (let ((e (make-engine caught)))
                    (if (zero? owed) (cons #f e)
                        (let ((th (λ () (e owed))))
                          ((call/cc (λ (k̂) ((cdr state) (λ () (k̂ th))))))))))))))))
```

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 [18] 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 [1].

```
(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))))
```