Daisy, DSI and LiMP
Issues in Architecture for Suspending Construction*

Steven D. Johnson
Computer Science Department
Indiana University

Abstract

This report briefly describes the functional programming language Daisy, it's underlying computational model, DSI, and a hypothetical architecture, LiMP, for their implementation. Daisy is a simple list processing language, derived from Pure Lisp, which inherits a call-by-need semantics through its use of a suspending constructor. DSI is the hetergeneous 'data space' of suspensions and manifest graphs, modeling concurrent task execution on a a parallel virtual machine. LiMP stands for List Multi-Processor; an MIMD architecture for parallel graph processing. The primitive mechanisms of LiMP are explained, and the evolution of the machine model from language oriented research is discussed.

*Research reported herein was supported, in part, by the National Science Foundation, under grants numbered MCS 82-03978, DCR 84-05241, and DCR 85-21497.

1. Introduction

The article "CONS should not Evaluate its Arguments" [FrWi76a] was one of several papers to appear in the mid 1970s on the subject of laziness in Lisp-like languages. The approach offered by Friedman and Wise was operational. They presented an interpreter, for Pure Lisp in Lisp, in which the elementary operations of Lisp are subverted to make list construction non-strict. `CONS` is a special form that initializes the fields of newly allocated list cells with *suspended* valuations. Suspensions are coerced to *manifest* values by the list-access primitives, `CAR` and `CDR`. An otherwise standard Lisp interpreter using these primitives delivers a call-by-name semantics, safely optimized to call-by-need if there are no side-effects [Wi82].

Suspensions quickly turned from simple closures to processes and subsequent publications reflect the metamorphosis. The two motives were parallelism and indeterminacy. The transparency of suspensions makes them an tempting way to involve parallelism in language execution. Since the result of a suspension is determined *When* to schedule suspensions is simply a resource-management problem [FrWi78b]. At the same time, efforts to address "systems programming" problems raised the need for a concurrency construct. The approach was to endow *data* with a quality of indeterminacy by introducing *ferns*, or lists ordered by need [FrWi80a, FiFr84 (Chapter 5)]. The understanding is that the ordering is determined by concurrent evaluation: The suspensions—really "tasks" now—that comprise a fern's content compete for promotion in list.

Though the evolution of suspensions from *delay*-like objects [He80] to *future*-like objects [Ha85] is evident in published work, the mechanics suspending construction its implications for architecture is not. Elements of architecture are scattered among articles about applicative language and style, leading Vegdahl's taxonomy to list "Friedman and Wise's Reduction Machine" as a largely unspecified object [Ve84]. This paper assembles the architectural aspects of this work, as seen in the relationship of three objects.

The first is Daisy, a mutation of (pure) Lisp, also related to the dialect Scheme. Daisy is a lazy-list processing language, hence also a lazy list-processing language, based on lambda notation. It is presented here, in part, to explain representation techniques that have been explored in the implementation of its interpreter. Daisy's virtual machine architecture will be of interest to language and processor designers, quite apart from notions of parallel execution that are developed in this paper.

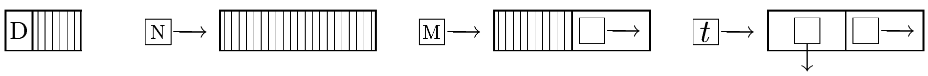
The second object is DSI (Data Space for the Interpreter) a model of list multiprocessing. DSI is Daisy's virtual machine. It maintains the transparency of suspended computations and is therefore a process management system. DSI models an idealized machine in which each suspension is a *processor* running of its own volition. An implementation of DSI, over which Daisy is now implemented, multitasks suspensions in a demand-driven fashion. Section 3 develops the interface between DSI and its processors as a small collection of storage transactions.

The third object is called a LiMP (for a List MultiProcessor). LiMP is a logical structure, whose purpose is to expose desiderata for a parallel implementation of DSI. The LiMP presented here has qualities in common with data flow architecture (communication via buffered routing) reduction systems (demand drive) and general purpose MIMD hosts (uniform global addressing). The elements of LiMP architecture described in Section 4 have to do with resource allocation. It is shown that the communication network contributes in essential ways to the balanced and timely dissemination of space resources. A weighted approach to demand propagation suggests that scheduling a dynamic and unbounded number of processes on fixed number of processors can be accomplished in a fully decentralized fashion.

LiMP remains, in Vegdahl's words, a "paper design only." It is one of many architectures considered for the implementation of DSI, to determine whether suspending construction is a viable approach to parallelism. However, its description exposes design issues of general applicability to related architectures and languages.

2. Daisy

Like Lisp, Daisy is a spelling out of expression representations. Daisy's "s-expressions" are different because tags are used to accelerate interpretation. A tagged reference is called a *citation.* The reference is a pointer unless the tag is $\boxed{\mathrm{D}}$, or *directive,* in which case the remaining content is data. Other citations refer to three storage formats, uniform cells with zero, one, or two citations.

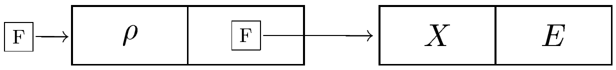


directive *nullary citation* *unary citation* *binary citation*

Binary cells have four distinct tags; $t \in \{\boxed{\mathrm{I}}, \boxed{\mathrm{L}}, \boxed{\mathrm{A}}, \boxed{\mathrm{F}}\}$.

Figure 1 diagrams how the seven tags are used to represent Daisy expressions. The second column in the figure gives Daisy's syntax. Directives denote machine actions, such as arithmetic operations and display primitives. Numerals represent numbers. Messages denote errors and print names. Literals are binary cells containing a print name and a citation to an *assigned value.* "Top level assignment" is currently permitted in Daisy's implementation. The names of primitives (e.g. `add`) are thus initially bound to their operations (e.g. the ADDITION directive). Quotations are assigned literals without print names; the directive QTE means spell-me-as-a-quotation to the display primitives. The binary objects are indistinguishable except through their citations; compare the list object `[`E_0 `!` E_1`]` with the application object $E_0 : E_1$. Since application is expressed in infix, parentheses are used for precedence. Parenthesized expressions are represented as applications an identity operation, IDY, which also means parenthesize-me to the display primitives.

A partial language definition is given in Figure 2.2. $\langle\!\langle E \rangle\!\rangle_\rho$ is the value of expression E in environment ρ. Daisy is lexically scoped; functions are values. “Record descriptions” like `[closure` ρ X E`]` are figurative; for instance, the closure of `\`X `.` E is actually represented as



The function $\mathcal{A}$ gives meanings of representative values when they are applied. $\mathcal{A}$ inherits the environment of an application in order to execute pseudo-operations like `let`, `rec`, and `val`.

Daisy treats errors as values (or perhaps anti-values). An *erron*—represented by message cells—is the result of an error and is treated as a detected divergence. That is, any computation that uses an erron is erroneous. There are currently no provisions for error recovery in Daisy.

Here are some Daisy expressions and their Lisp counterparts.

F`:[`A`]`	`(`F A`)`
`\`X`.`E	`(lambda` X E`)`
`[`E_0 `!` E_1`]`	`(cons` E_0 E_1`)`
`[`E_1 E_2 E_3`]`	`(list` E_1 E_2 E_3`)`
`if:[`P C A`]`	`(cond (`P C`) (true` A`))`
`let:^[[`X_1 X_2`]`	`(let ((`X_1 E_1`)`
`[`E_1 E_2`]`	`(`X_2 E_2`))`
E`]`	E`)`

However, Daisy's semantics is essentially normal-order because its `cons` is “lazy;” the value of `[`E_0 `!` E_1`]` is a list whose head is $\langle\!\langle E_0 \rangle\!\rangle$ whether or not E_1 has a value. Neither E_0 nor E_1 is evaluated unless and until the list is accessed. Hence, an application expression, such as F`:[`E_1 E_2`]`, develops call-by-name arguments, executing the text of F outside the valuation of E_1 and E_2. For example, an alternative conditional can be developed by coercing truth values to list-accesses.

```
head = \[H ! T]. H
tail = \[H ! T]. T
AnIF = \[P C A]. (if:[P head tail]):[C ! A]
```

In a prettier syntax,

$$\begin{aligned}
&\mathit{head}\,\texttt{[}H\;\texttt{!}\;T\texttt{]} \;\Leftarrow\; H.\\
&\mathit{tail}\,\texttt{[}H\;\texttt{!}\;T\texttt{]} \;\Leftarrow\; T.\\
&\mathit{AnIF}\,\texttt{[}P\;\;C\;\;A\texttt{]} \;\Leftarrow\; (\textbf{if}\ P\ \textbf{then}\ \mathit{head}\ \textbf{else}\ \mathit{tail})\,\texttt{[}C\;\texttt{!}\;A\texttt{]}
\end{aligned}$$

Thus, it is reasonable to embed recursions in argument structure, rather than in conditional structure, as in

```
SEARCH = \[Key Tree Fail].
   let:[ [Left [Ide Value] Right]
         Tree
         if:[ nil?:Tree
                 Fail
              same?:[Key Ide]
                 Value
                 SEARCH:[Key Left SEARCH:[Key Right Fail] ]
            ]
       ]
```

In a prettier syntax, the Daisy expression above might be expressed as

$$
\begin{array}{l}
\mathit{SEARCH}(\mathit{Key},\ \mathit{Tree},\ \mathit{Fail}) \leftarrow \\
\quad \textbf{let} \\
\qquad [\mathit{Left}\ [\mathit{Ide}\ \mathit{Value}]\ \ \mathit{Right}] = \mathit{Tree} \\
\quad \textbf{in} \\
\qquad \textbf{if } (\mathit{Tree} = \mathit{Nil}) \textbf{ then } \mathit{Fail} \\
\qquad \textbf{else if } (\mathit{Key} = \mathit{Ide}) \textbf{ then } \mathit{Value} \\
\qquad \textbf{else } \mathit{SEARCH}(\mathit{Key},\ \mathit{Left},\ \mathit{SEARCH}(\mathit{Key},\ \mathit{Right},\ \mathit{Fail}))
\end{array}
$$

As noted earlier and illustrated above, formal arguments, such as `[Left [Ide Value] Right]` need not be flat lists*. As illustrated above, the formal argument X in a function expression $(\backslash X\ .\ E)$ is, in general, a tree of literal identifiers.

$$\langle \mathit{Arg} \rangle ::= \mathit{Ide} \quad | \quad \texttt{[]} \quad | \quad \texttt{[}\mathit{Arg}\,\texttt{!}\quad \mathit{Arg}\texttt{]}$$

As usual, cascading dots are suppressed. Environments bind identifiers to their corresponding positions in arguments, according to

$$\rho \begin{bmatrix} v \\ i \end{bmatrix} (j) \;=\; \begin{cases} v, & \text{if } i = j \\ \rho(j), & \text{if } i \neq j \text{ or } i = \texttt{[]}. \end{cases}$$

$$\rho \begin{bmatrix} v \\ \texttt{[}X_0\texttt{!}X_1\texttt{]} \end{bmatrix} \;=\; \rho \begin{bmatrix} \mathit{tail}(v) \\ X_1 \end{bmatrix} \begin{bmatrix} \mathit{head}(v) \\ X_0 \end{bmatrix}$$

* Binding is deferred, so that structural mismatches are not erroneous unless they are used. Even so, the `let`-binding of `Tree` in `SEARCH` might better be declared *after* the `nil?`-test.

head and *tail* are field-access primitives that are erroneous if applied to atoms or errons. They are developed in more detail later. In representation, environments are maintained in two distinct structures, a *formal environment* of bound identifiers and an *actual environment* of their associated values. If F and A are a formal and an actual environment, and ρ is $\texttt{[}F\ \ A\texttt{]}$, then

$$\rho\begin{bmatrix} v \\ X \end{bmatrix} \text{ would be } \texttt{[[}X\ \texttt{!}\ F\texttt{]}\ \ \texttt{[}v\ \texttt{!}\ A\texttt{]]}$$

Identifier look-up has two phases: construction of a *probe*, or combination of *head*s and *tail*s, and then application of the probe to the actual environment.

$$\mathit{LookUp}(I, F, A) = \big(\mathit{Probe}(I, F, (\lambda v.v), (\lambda v.\,\textsc{error}))\big)(A)$$

where

$$\begin{aligned}
\mathit{Probe}(I, J, \mathit{ack}, \mathit{nak}) \ &= \begin{cases} \mathit{ack}, & \text{if } I = J \\ \mathit{nak}, & \text{if } I \neq J \text{ or } J = \texttt{[]}. \end{cases} \\
\mathit{Probe}(I, \texttt{[}X_0\texttt{!}X_1\texttt{]}, \mathit{ack}, \mathit{nak}) \ &= \mathit{Probe}(I, X_0, [\mathit{ack}\,;\mathit{head}], \\
&\qquad\qquad \mathit{Probe}(I, X_1, (\mathit{ack}\,;\mathit{tail}), \mathit{nak}))
\end{aligned}$$

If π_1 and π_2 are probes, $[\pi_1;\pi_2]$ is their composition, $\lambda v.\pi_2(\pi_1(v))$. The representation of a probe, called a *trajectory*, is an encoding of $[\pi_1;\ldots;\pi_n]$. Since Daisy is a lexically scoped language most trajectories can be computed prior to execution, but they aren't.

The non-strict list constructor permits creation of infinite structures and also recursive data definition. If `inc` is an increment operation, two ways of defining a list-of-all-numbers are

```
rec:^[ Z  (\N. <N ! Z:inc:[N]>)  Z:0 ]
```

and

```
rec:^[ Z  <0 ! Map1:[inc Z]>  Z]
```

The alternate delimiters `<...>` are simply for emphasis; they have the same meaning as `[...]`. In prettier syntax, these would be

$$Z(0) \text{ where } Z(N) \ \Leftarrow\ \texttt{[}N\ \texttt{!}\ Z(N+1)\texttt{]}$$

and

$$Z \text{ where } Z = \texttt{[0 !}\ \mathit{Map}_1(\mathit{inc}, Z)\texttt{]}.$$

`Map1` applies `inc` to every element of `Z` and might be defined as

```
Map1 = \[F [V ! Vs]]. <F:[V] ! Map1:[F Vs] >
```

Both versions of `Z` return the value `[0 1 2 3 ...]`, which, if printed in the usual way, produces the answer "`[0 1 2 3 ....`" The answer can be printed (in bounded space [FrWi76b]) because the sequence `Z` has enough initial content to define the whole sequence. This can be seen by symbolically expanding the definitions. A form of construction functional, called *functional combination* by Friedman and Wise [78a], is implicit in the application of a list (See Figure 2.2). It is a generalization of `Map1`. The data recursion above is equivalent* to

```
rec:^[ Z  <1 ! [inc*]:[Z]>  Z]
```

`[inc *]` evaluates to a non-terminating list of INCREMENTATION operations. A style of expression reminiscent of Lucid [WaAs86] is prevalent among Daisy programers. Iterations are embedded in recursively defined structure. For instance, one is more likely to see the fibonacci function defined as an indexing operation on a sequence,

```
FIB = \N. rec:^[ [U V]
                 [ <1 ! V>
                   <1 ! [add*]:[U V]> ]
                 N:F
               ]
```

than its function counterpart,

```
FIB = \n. rec:^[ F
                 \[U V W]. if:[ zero?:U V FIB:[dcr:U W add:[V W]] ]
                 F:[n 1 1]
               ]
```

With lists representing sequences, the first expression says

$$FIB(n) = f_n \text{ where } f_0 = f_1 = 1 \text{ and } f_{k+2} = f_k + f_{k+1}$$

The second expression is the loop

$$\begin{aligned}&FIB(n) \Leftarrow F(n,1,1)\\ &\quad\text{where}\\ &\qquad F(u,v,w) \leftarrow \textbf{if } (u=0) \textbf{ then } v \textbf{ else } F(u-1,w,v+w)\end{aligned}$$

Application is strict in both function and argument; hence, Daisy is not inherently a lazy language. For instance, the expression `(\X.3):rec:^[Z Z Z]` diverges

---

* Well, almost. Unary operations cannot be used in functional combinations. This point is discussed in [Jo84].

because `rec:^[Z Z Z]` diverges, but `(\X.3):[rec:^[Z Z Z]]` safely produces `3` because its argument is in a list.

Daisy views a peripheral device as a producer/consumer of stream-like character lists. Interactive `Daisy` is essentially

```
Daisy = \Prompt. screen:issues:[val *]:[parses:console:Prompt]
```

`Daisy` is a composition of primitives. `Console` produces a character list from the operator's key strokes; `parses` filters that stream into Daisy expressions; `[val *]` maps valuation over the parsed input; `issues` turns the resulting value-stream into a character stream; and `screen` displays it on the operator's terminal screen.

A final Daisy construct has been omitted from this introduction, for want of a simple compositional description. An indeterminate constructor, builds a list whose order is determined, on demand, by concurrent valuation of its elements. The sooner an element-expression produces a result, the earlier its value occurs in the resultant ordering. These objects, called *ferns* and mentioned in Section 1, encapsulate concurrency (or indeterminacy) as a quality of data, just as a suspending constructor encapsulates qualities of laziness.

A formalization of ferns has proven to be more elusive than usual for such things, because ferns mush coexist with "deterministic" data and also are intended for implementation under fairly weak exclusion constraints. Their published explanations have always been operational [FrWi80b]. As an example of their use, the program below sketches a conditional which simplifies according to "$(p \to x, y) \Rightarrow x$" when x and y are equivalent list structures [FrWi78c, Jo77]. Assume an expression *DIVERGE*, whose valuation does not terminate. The function `GUARD` converges only if its predicate is true. The operation `same?` tests for atomic equality.

```
GUARD = \[P V]. if:[P V DIVERGE]

GIF = \[P C A].
   let:^[ H        GIF:[P head:C head:A]
   let:^[ T        GIF:[P tail:C tail:A]
        | in                                <-- This is a comment.
          head:{ if:[P C A]
                 GUARD:[ same?:[C A] C ]
                 GUARD:[ and:[list?:C  list?:A]  <H ! T> ]
               }
        ]]
```

3. DSI

DSI is the vehicle for Daisy interpretation. It is a heterogeneous space of *suspended* computations and *manifest* values. To execute, a computation requires a *state*, its registers and a control-point; and a *context*, the location where it resides. A context is a citation with a field designation, "head" or "tail." A *suspension* is a record of state, lacking only a context to be run as a computation. In representation, suspensions and values are distinguished by an *exists-bit* within the fields that cite them. That is, if the content of a field is a computation, its exists-bit is *false* and the field contains a pointer to the a suspension.

Computations do graph processing, and DSI represents their store. It admits the following operations:

- *New* obtains a citation to unallocated storage. New cells are always initialized, and the operation is usually expressed as `[`$\star$ `!` $\star$`]`.
- *Suspend*(*State*) obtains the suspension of a computation. A suspending *cons* looks like `[`*suspend*($State_0$) `!` *suspend*($State_1$)`]`
- *Sting*(*Context*, *Value*, *Condition*) is a storage update, which deposits *Value* at the location *Context*, provided *Condition* is true at that location. *Condition* is a simple test, always a test of the exists-bit in this paper. Stinging is often expressed as, for example,

  sting-head *Citation* **with** *Value* **unless it exists**

  The suffix "**-head**", turns *Citation* into a *Context* and "**it exists**" refers to the *Context*'s exists-bit is set.
- *Converge*(*Value*) stores the result of a computation in its context, in place of the computation's suspension.
- *Head*(*Citation*) returns *Citation*'s "`CAR` field" if it is manifest. If the field contains a suspension, the caller is blocked until it becomes manifest, and the cited suspension becomes a computation whose context is *Citation*'s `CAR`.
- *Tail*(*Citation*) is like *head*(*Citation*) except that the context specifies the "`CDR` field."

The presence of other computations is transparent to any individual, except when they are created. The interaction of computations is governed by field access, through the probes *head* and *tail*.

The recursive language specification in Figure 2 transforms to a sequential graph-reduction interpreter, using techniques detailed in [Jo84, Ch. 5]. The Daisy interpreter is outlined in Figure 3. The actual implementation unfolds *VALUE* at several points. In particular, its *LIST* is actually a loop approaching the specification

$$\langle\!\langle \texttt{[}E_0\ \ E_1\ \ \ldots\ \ E_n\texttt{]}\rangle\!\rangle = \texttt{[}\langle\!\langle E_0\rangle\!\rangle\ \langle\!\langle E_1\rangle\!\rangle\ \ldots\ \langle\!\langle E_n\rangle\!\rangle\texttt{]}$$

Each parameter is a citation whose name connotes its type. *Exp* is an expression; *Env* an environment; *FVal* and *AVal* are values; *Ctx* is the execution context; and *Cmd* is a stack-like object, and used to implement recursion. The construct "**per** B **viz** ..." is a generalized case-statement with actions predicated on tests against local variables. For example, *VALUE* performs a different action depending on the tag in the citation *Exp*.

Though Figure 3 has a functional form, it is essentially a register-transfer description. The branch $\mathit{RETURN}(\texttt{[closure}\ \mathit{EnvExp}\texttt{]}, \mathit{Nil}, \mathit{Cmd}, \mathit{Ctx})$ could be written as a parallel assignment

$$\begin{bmatrix} \mathit{Ctl} \\ \mathit{Exp} \\ \mathit{Env} \end{bmatrix} := \begin{bmatrix} \mathit{CONTINUE} \\ \texttt{[closure}\ \mathit{Exp}\ \mathit{Env}\texttt{]} \\ \texttt{[]} \end{bmatrix}$$

where *Ctl* is a program-point—the "micro-PC." Each right-hand term involves an elementary operation on state or a storage transaction.

Daisy interpretation is the primary influence in the DSI process design—what has until now been called a "computation." It contains a program register; three general citations accounting for *Exp*, *FVal*, *AVal*, and *Env*; a display of the continuation, (e.g. $\texttt{[converge}\ \ \mathit{Env}'\ \ \mathit{Cmd}'\texttt{]}$), and a context (a citation plus a bit). Local control is governed by tags, actions (e.g. `converge`), and the status of information processing operations. Figure 3 shows a logical process structure, the portion represented in a suspension, and a possible architecture for physical implementation.

3.1 Concurrency in DSI

As it has been described, Daisy is a deterministic language in which the invocation of suspensions results from the execution of linear trajectories against actual environments. Hence, the dependence among computations is linear and stack-like. At any point, there is only one suspension that needs to be run, the one exposed by the most recent *head* or *tail*. Prospects for concurrency arise in several ways, including the three mentioned below.

Multiple Outputs. A direct source of concurrency is the presence of several output devices, placing parallel demand on the data space to produce answers. These might be physically distinct, as in a time-sharing environment; or logically distinct, as in a multiple-window terminal, or both. Such systems exhibit independent sources of true need in the list space, exposing many suspensions simultaneously.

Explicit Concurrency. The fern was mentioned in Section 2 as a form of expressed concurrency. There are many possible variations, such as a collateral-*cons*, which has deterministic order but whose elements are concurrently evaluated.

Implicit Concurrency. This is the transparent infusion of "speculative" computation, probing suspensions when need is anticipated but not exhibited. Assuming the there are no side effects, arbitrary suspensions may be executed without changing a program's meaning (Even if side-effects are permitted, suspensions can still be run in parallel). Thus, a system with extra processors may invoke suspensions using criteria other than pure demand. However, the idea is to keep from indefinitely committing processors to unneeded computations, as these might consume them.

We shall now look at the other side of the DSI interface, the side concerned with managing processes. Suspensions—considered as representations—are subject to operations of creation, resumption, preemption, and completion.

Creation is expressed as *suspend*(*Computation*), as earlier. The typical *initial* suspension in Daisy interpretation is $\langle\!\langle E \rangle\!\rangle_\rho$, "Evaluate expression E in environment *Env*, then converge." This is $\mathit{VALUE}(E, \mathit{Env}, \texttt{[converge]}, \star)$ in Figure 3. The typical suspension is a preempted process; the six registers of Figure 3, though usually fewer than six are active. In any case, the operand *Computation* is relatively small, often one or two citations.

Resumption is encapsulated in an action called *coaxing*. To run, a processor needs a suspension and its context; but the suspension is cited by its context. The instruction

$$\mathit{coax}(\mathit{Context})$$

schedules a suspension for bounded execution*. The variations *coax-head*(*Citation*) and *coax-tail*(*Citation*) make contexts from their *Citations* and coax them.

The completion of a process is to converge, or displace itself with a value.

converge(*Value*) $\Rightarrow$
  sting *Context* **with** *Value* **unless it exists;**
  stop.

Context refers to the caller's context-register. If a suspension can have more than one context, *converge*must become a loop. The meta-instruction **stop** represents

* To coax originally meant "run the suspension for a while, *now*" [FrWi80a, FrWi80b]. However, decoupling the scheduling event and eventual execution was considered as early as [Jo77], and done by [JoKo80]

the deactivation of the running process. A process may relinquish its processor by detaching:

$$\begin{array}{l} \mathit{detach}(\mathit{Computation}) \Rightarrow \\ \quad \textbf{sting } \mathit{Context} \textbf{ with } \mathit{suspend}(\mathit{computation}) \textbf{ unless it exists;} \\ \quad \textbf{stop.} \end{array}$$

The *sting* advances the state of the detaching process. The updates in convergence and detaching are conditioned on the presence of a suspension at the target location. A conditional-write is sufficient to coordinate all concurrency in Daisy programs, although a test-and-set would be useful. It must only be assured that values cannot regress to suspensions. **Sting** is a slightly weaker operation than test-and-set because it is a kind "store" and not a kind of "fetch." The necessary synchronization of Daisy's concurrency in DSI can all be localized to storage [FrWi78b]. Wise incorporates a **sting** implementation in his CMOS design of a self managing heap segment [Wi85b].

Probes are deterministic in the sense that the probing process is blocked until a value appears. One way to express this is with persistent coaxing:

$$\mathit{coerce}(\mathit{Context}) \Rightarrow \begin{array}{l} \textbf{repeat } \mathit{coax}(\mathit{Context})\textbf{; block} \\ \textbf{until } \mathit{Context}^{\uparrow} \textit{ exists} \end{array}$$

It is fundamental that processes are insulated from direct contact with each other. They interact through contexts.

$$\mathit{head}(\mathit{Citation}) \rightarrow \begin{array}{l} \textbf{repeat } \mathit{coax\text{-}head}(\mathit{Citation}) \\ \textbf{until } \mathit{Citation}^{\uparrow}.\mathrm{head} \textbf{ exists;} \\ \{\mathrm{return}\}\ \mathit{Citation}^{\uparrow}.\mathrm{head}\textbf{.} \end{array}$$

The suggestion of busy-waiting is not necessarily metaphorical. In a parallel implementation, where it is unlikely that perfect knowledge of process dependence can be decentralized, coaxing is the communication of "need" for a result in DSI. Repetitive coaxing, or *coercion,* communicates absolute need on behalf of the caller, modulo need *for* the caller. For example, in ferns a probe coaxes all the elements until one converges. Determinate element-computations concurrently probe their environments, coercing still more suspensions. When an element is found, the initial probe is satisfied and demand ceases. Without a supply of need, the remaining computations cease *their* probing activity.

At the virtual-machine level, probing is the only means a process has of affecting another process. After creating a suspension, there is no alternative but to place it in a data cell. As in Figure 3, a "lazy" *cons* looks like

$$\begin{array}{l} \mathit{cons}(\mathit{Computation}_0,\ \mathit{Computation}_1) \Rightarrow \\ \quad \texttt{[}(\mathit{suspend}(\mathit{Computation}_0)\ \ \texttt{!}\ \ \mathit{suspend}(\mathit{Computation}_1)\texttt{]}\ . \end{array}$$

Other varieties of construction place values in one or both of the fields. A stream-like constructor would first perform $Computation_0$ and then allocate a cell

$$\begin{aligned} &cons(Computation_0,\ Computation_1) \Rightarrow \\ &\qquad \{\text{perform } Computation_0 \text{ with result } V_0\} \\ &\qquad [V_0\ !\ suspend(Computation_1)]\ . \end{aligned}$$

4. A LiMP

LiMP is a hypothetical MIMD architecture, built from List Storage Elements (LSEs) and List Processing Elements (LPEs). These are connected by a rectangular routing network of buffered Routing Elements (REs). Figure 4 is a picture of overall architecture. Each of the elements actively contributes to DSI computation. The LSEs are not just passive storage; the LPEs are not just processors; the REs are not just switches. The nature of computational activity in LiMP is developed gradually as the DSI primitives are implemented.

Together, the LSEs comprise a shared, multiported heap of binary list cells. Each is responsible for a region of the heap space, and the regions partition the whole. An LSE locally maintains a free space of uncited cells in its region. It responds to storage transactions for fetching, updating, and allocating. In addition, LSEs initiate activity among the LPEs, as is discussed later. Similarly, the LPEs maintain a space of available suspensions. Each LPE allocates, schedules, and executes suspensions in its own region. LPEs and LSEs have roughly the same internal structure, depicted in Figure 4. Within each is a processor, called a Process Management Unit (PMU) in LPEs and a Storage Management Unit (SMU) in LSEs. The PMU contains the architecture of Figure 3(b) with additional state for swapping suspensions. The external behavior of an LPE, at its interface with the routing network, is a sequence of fetches, stores, and allocation requests. Input and Output devices are specialized LPEs; connecting connect directly to the routing network. Inputs produce and Outputs consume lists, using ordinary list manipulation primitives.

The connection network is a system of buffered REs, which carry instruction objects between the process space and the list space. These objects are of uniform size, representing elementary storage transactions. The network is rectangular; a system with N LPEs and N LSEs has $N \log_b N$ REs. For simplicity, assume $b = 2$, so that an RE surrounds a 2×2 cross-bar switch with staging capabilities. A banyan [TrLi79] configuration of REs for LiMP is attractive because it provides uniform addressing with identical routing algorithms in each RE. It is logically extensible to the limit of a fixed address size. There is a unique path between any origin and destination, so that the order of point-to-point communications is preserved if RE buffers are first-in-first-out. Destination-address bits determine successive switch settings. The address $(M \cdot 2^q) + n,\ n < 2^q$, selects LSE M regardless of its point of origin. As destination bits are consumed, the RE records the message path to recover the originating address. That is, if an object originates from LPE P, the destination $(M \cdot 2^q) + n$ is transformed to $(P^{-1} \cdot 2^q) + n$ by the time it reaches LSE M. If a response is required it is routed along the path P^{-1}.

An RE's external behavior is identical to that of an LSE on its "LPE side" and conversely. A minimal LiMP configuration is a single LSE and LPE, directly connected. It is extended by introducing REs within the banyan configuration until there are enough places to put additional elements.

The goal in computation is to trade turnaround against throughput, as in data flow architectures [DeLoBe80]. That is, the elapsed time of an individual transaction is offset by increasing the number of transactions done in parallel. The dominant factor in elapsed time is buffering in the routing network, but because of the buffering, the involvement of processing elements is small. The *profile* of an individual transaction is expressed as $u \cdot \Theta \cdot v$. Θ stands for the delay incurred by routing an object from its origin to its destination; u gives the involvement of the originating element and v the involvement of the target. The *involvement* in a transaction is $u + v$ if both participants can be gainfully employed during the routing delay. The elapsed time of a transaction is $u + \Theta + v$. The unit is the time it takes to stage an object between adjacent REs. This is a function both of electrical protocols and also the degree of blockage in the network. Profiles are woefully approximate.

We begin the implementation of DSI on LiMP by looking at basic storage transactions.

4.1 STINGs and RSVPs

A STING requires the routing network to convey a conditional store operation from the originating LPE toward the destination LSE. The typical STING object is

`[sting` *Condition Location Content*`]`

which represents the instruction **sting** *Context* **with** *Value* **unless** *Condition*. *Condition* is one of a fixed set of simple tests on the target content. The "store" is inhibited by the destination LSE if the condition is false when the STING arrives. There is no acknowledgment of the store.

An LPE (or RE) can issue a STING and then proceed without acknowledgement, provided the STING is guaranteed to reach its destination before any subsequent transaction *by the same process with the same destination.* As was noted earlier, this requirement is logically satisfied by a banyan network. In the absence of side effects, a STING establishes the assertion

$$\{Location \uparrow = Content \vee \neg Condition\}.$$

This is enough to implement Daisy because contention can be arbitrated in storage, as discussed earlier,

An RSVP requires the routing network to carry a "fetch" instruction to the destination LSE, and to return the target content to the originating LPE, assuming it exists. Typical RSVP objects are

$$[\texttt{head}\ \ Location] \quad \text{and} \quad [\texttt{tail}\ \ Location]$$

The elapsed time in an unblocked routing network is twice the length of the path between the parties to the transaction, say $P = 2\Theta \approx 2\ln N$, where N is the number of processors. It is reasonable to assume that RSVP transactions dominate in graph processing programs. Hence, an LPE executing a single process would make headway of about $\frac{1}{P,}$ at best.

Thus, take P to be the maximal degree of *multiplexing* an LPE can do in its local process space.

This is similar to process pipelining in HEP architecture [Sm78], and is also discussed by Halstead as a direction for Multilisp [Ha85]. The true degree of multiplexing cannot reach P because this would flood the routing network, increase blockage, and degrade performance. It is essential to leave enough "holes" to ameliorate blockage.

With multitasking, an LPE manifests a *set* of active processes and must maintain the status of their pending RSVPs. An LPE need only associate each process with the location whose content it awaits. The LSE's response to an RSVP,

$$[\texttt{headof}\ \ Origin{\cdot}Offset\ Content]$$

includes the local offset of the content being returned. The REs reconstruct the region address by recording the path of the response. The concatenation of LSE region and offset gives the original location.

4.2 The NEW sink

A NEW instruction object, [new *Citation*], requests a fresh citation. The effective elapsed time for NEWs is minimized if the routing network participates in storage allocation. Each RE contains a register for a citation to one new cell. When this buffer is empty and the RE is idle, it issues a NEW instruction to either of its LSE-side ports. New citations thus migrate from LSEs toward LPEs as traffic permits. The network becomes a reservoir of available cells, or NEW *sink* [Jo81].

Staging new citations in the network accomplishes several ends. If the NEW sink is filled, NEW transactions are serviced in unit time. Allocation is fully distributed, so that no global coordination is needed to synchronize accesses to "free space." Surges in allocation are dissipated by the network reserve. There is a path from any LPE to all LSEs; the sub-network servicing a given LPE is a full binary tree of REs. A heavy drain draws citations from throughout the list space, spreading accesses to all the LSEs. Since new citations are the media for future graph processing, the effect is to delocalize structures emanating from a given computation. If an LSE is unable to supply new citations—either because it has none or because it is busy with STINGs and RSVPs—those that can take up the slack. "Hot spots" in the heap partitioning cool off as computation advances in the graph space.

The degree of potential parallelism in LiMP cannot be measured simply by counting processors and memory banks. One must also include the participation of the routing network. Where an LPE is multitasking, the effective headway approaches $2\Theta N$, where $\Theta \approx \ln N$ is the unidirectional network delay. A STING has profile $1\cdot\Theta\cdot 1$; a (successful) RSVP has profile $1\cdot\Theta\cdot 1\cdot\Theta\cdot 1$; and a NEW has profile $1\cdot 1$ because REs anticipate allocations in a NEW sink.

4.2 The Process Sink

Processes objects—suspensions—are allocated in the same manner as free cells, except that it is the LPEs which are the allocators. They provide new-suspension citations to a *process sink* in the routing network. These resources migrate toward the LSEs, the network being a reservoir to dissipate demand by consumers. As in the NEW-list sink, the reservoir is supplied by less active allocators, with the effect of involving them in present computation. An LPE schedules and multitasks suspensions it has allocated in its own region of the process space.

To put this all together, let us consider the history of an individual process, β, created by a second process, α, and probed by a third process, γ. The suspensions of α, β, and γ are at locations A, B, and C in the process space and are run by processing elements, LPE_A, LPE_B, and LPE_C. Suspension B will have context N from the storage element LSE_N. Scenarios for the creation and coercion of β are shown in Figure 4, which depicts the involvment of the four elements.

At the outset, the network has already absorbed citations to a new cell, N, and a new suspension, B. The creation scenario is initiated by α, active on LPE_A. LPE_A first obtains the context N from the NEW-list sink. It then issues a STING, `[suspend` β N`]`. After that, LPE_Amay proceed, perhaps continuing the execution of α. When `[suspend` β N`]` arrives at LSE_N, citation B is obtained from the NEW-suspension sink and is stored at location N. LSE_N immediately issues `[create` β`]` to the network; this is also a STING transaction. When `[create` β`]` arrives at LPE_B, β is recorded in the suspension B. If both sinks are populated, the transaction profile is

$$1\cdot 1\cdot 2\cdot \Theta\cdot 3\cdot \Theta\cdot 1$$

Figure 4(b) is a coercion scenario. Process γ in processor LPE_Cexecutes an RSVP, `[head` N`]`. The initial RSVP reaches LSE_N, where it is determined that the target field is suspended. LSE_N issues `[coax` B N`]`, which is routed to LPE_B; the operand N is the execution context for β.

Should LSE_N return a negative acknowledgment of the attempted fetch? To do so would help LPE_C adjust its local schedule but add traffic in the routing network. It is assumed that there is no acknowledgement.

When `[coax` B N`]` arrives at LPE_B, it schedules β for finite execution. If β produces no value, *and* γ *remains active,* another `[head` N`]` is issued by LPE_C, resulting in another `[coax` B N`]` from LSE_N to LPE_C. The cycle may even overlap. When β does converge, it deposits its value at location N. This value is returned by LSE_N to LPE_C, once it arrives. The coercion profile is

$$[1\cdot \Theta\cdot 2\cdot \Theta\cdot S]^z\cdot 1\cdot \Theta\cdot 1\cdot \Theta\cdot 1,$$

where z is the number of coaxes required to coerce β to existence and $S > 1$ is the multitasking overhead. If T_β is β's execution time, total involvement is on the order of $3z + Sz + T_\beta$, with elapsed time adding $2(z+1)\Theta$. Clearly, suspension of a computation is a poor way to implement a subroutine call in LiMP. Even supposing we can make z, the degree of polling, zero, so that every RSVP is successful, the best elapsed time to coerce a suspended computation greater than 4Θ, which is the time it takes to transfer relevant contents to their locations, and involves about ten instructions among the involved parties. In DSI, on the other hand, the costs of a subroutine call and a suspension invocation are much closer, and it is a reasonable design objective to bring them to equality. We shall next look at three approaches to the minimization of z.

The LiMP allocation tactics discussed so far achieve three important goals. First, anticipatory allocation in the NEW sinks reduces bottlenecks in allocation. The cost for this improvement is the need for regularity in representations. An RE could not anticipate needs for variably sized objects. Second, the NEW sinks spread allocation so that LiMP is statistically self balancing. The reservoir tends to draw processing and storage resources where they are more readily available. Since newly allocated resources embody present computational activity, activity on the whole is distributed among LSEs and LPEs. Third, assignment of processors to processes is a function of system configuration; it transparent to software. As the two scenarios illustrate, no process ever has direct reference to another; the interplay is exclusively maintained by LSEs.

4.3 Scheduling LiMP

The optimal scheduler for DSI runs those processes at the "leaves" of the dependence graph revealed by *head*s and *tail*s. This set of suspensions is called *the fringe.* LiMP has the problem of mapping physical processors to a potentially larger fringe set of processes. Perfect knowledge of the fringe is unlikely without a shared representation, and a goal in LiMP is to do decentralize this kind of information. In the extreme, we assert that scheduling should be fully decentralized, so that each LPE maintains a local schedule. If not, The Schedule becomes a critical resource and access to it a critical section.

The coercion scenario in Figure 4 illustrates a first approximation to decentralized scheduling: an LPE runs a suspension provided an incoming coax has exhibited a need for it. Demand-driven computation is very sensitive to how well need can be propagated. Both elapsed time and involvement are compounded when suspensions arrange themselves in chains, each coaxing the next. The optimization problems are to safely minimize the number of suspensions and to minimize coaxing without overcommitting processors. In DSI, the former means using strict versions of *cons* where possible. Hall's strictness analysis develops this perspective [Hall87]; the issue is not pursued here.

One way to make coercion more efficient is to propagate more need with every coax. A *demand coefficient* is incorporated in each instruction object. An RSVP carries some or all of the originator's reserve of demand. Should the targeted content exist, its LSE returns this quantity to the originating LPE. Otherwise, it is sent to cited suspension, telling how far to run. Suspensions accumulate demand if they are shared.

Demand is a resource depleted by execution, and propagated by probes. It is the bound on how far a process may run. Output devices have an inexhaustible supply, meted out in finite chunks. A polling device refreshes demand, so that, concurrent with substantive computation, need is continually emanating toward the fringe. The demand coefficient has other uses. It can contribute to routing decisions in heavy traffic. It can be a factor in local scheduling, and the total demand absorbed by an LPE should be a measure of resistance to NEW requests.

4.4 Speculative Computation in LiMP

Indirect demand propagation is a global mechanism for speculative computation. In [FrWi76b], Friedman and Wise proposed that processing resources be applied in the vicinity of "important" computations. As they describe it, idle *sergeant* processors search for suspensions that are close to *colonel* computations. The sergeants coax computations in the hope of producing values just in advance of colonels' accesses. The idea is exploit locality; a printer, for example, is likely to coerce both fields of every list cell it visits. Thus, when The Printer visits the head of a list, a sergeant would do well to coax the tail. The problem in this scheme is determining proximity* to a colonel without getting in its way by tying up a critical LSE.

The sergeant/colonel effect is attained by a local LSE mechanism, which diverts a portion of the demand coefficient. On receiving the instruction δ`[head` *Citation*`]` with demand coefficient δ, the LSE coaxes *both* of *Citation's* fields demand coefficients totaling δ. In the event that the two cited suspensions are in different LPEs, one is given to a needed computation, the other a speculative one. Since this is the desired condition, process allocation should be spread, adding to the rationale for a NEW-*Suspension*sink. Since *Cons* suspends two computations in rapid succession, the suspensions are likely to be drawn from different regions of the process space.

No individual coax can cause divergence because the demand coefficient is finite. The average number of coaxes needed in coercion may be brought beneath one if a coax is associated with the *creation* of a process. A *speculative cons* includes a demand coefficient in the `[suspend` $\star$ $\star$`]` instruction object. The coefficient is incorporated by the LSE in the ensuing `[create` $\star$`]`, which must also carry a context, telling the allocating LPE to run the suspension as soon as it is initialized. With luck, the suspension will converge before ever being coaxed.

The demand-driven computational scheme for LiMP can decentralize scheduling decisions, propagating demand from its source in proportion to the strength of the need. A polling discipline seems a reasonable approach to maintaining collective knowledge about the fringe of essential computations, without resorting to a central scheduling agent. A demand coefficient can also be used to engage processing resources in speculative computations.

---

* Distance from The Printer has come to be called the O'Riley metric, after the character Corporal Walter "Radar" O'Riley in the motion picture *M*A*S*H* ( ), later a television series. Corp. O'Riley, a company clerk, earned his nickname by carrying out orders before they were given, answering the telephone before it rang, and so on.

5. Summary

Daisy exists for two reasons, examination of the thesis that functional programming is a practicable vehicle for mundane applications and exploration of a specific operational model of parallelism, based on suspending construction. The architectural aspects of this work, presented here, have little motivation unless it can also be shown that general purpose programming can be done in a purely functional setting. Our formulation of the question is this: If a programming *had* to be done on a functional vehicle, what kind of "culture" would evolve? What flavors of operating systems and programming environments would develop? If assignment statements did not exist, would it be necessary for programmers to invent them? The majority of articles reporting this work address these kinds of questions.

Our experience tends to support the thesis that functional programming is practicable, provide certain issues are resolved in architecture. Performance is one, of course, but beneath that is maintaining a functional view of the world. For instance, a purely functional system must present a functional view of the peripheral environment. A substantial amount of software in DSI's implementation is devoted to overriding the conventional behavior of input/output. Despite the effort however, an alarming amount of overhead in Daisy execution is due to device management. What this reveals to its implementers is their own naiveté in thinking that language execution is the keystone of process design.

Even so, the DSI process process is tailored to language interpretation. The virtual-machine employs tags in order to begin interpretation of an object before looking *in* that object. We suffer the fates of all who experiment with tags: chronic indecision about how to use them; simultaneous desires for more and fewer, resolved by overloading; the effort of their explanation; and persistent conviction that they make good sense in architecture. In Daisy interpretation, tags are used to discriminate syntactic entities (i.e. expressions), semantic entities (e.g. closures), and machine actions (e.g. tests and operations). Only the first is presented here. There is little reason to promote Daisy's specific uses for tags; in fact, their meanings have often changed.

In DSI's third implementation done in 1980, the main design goal was to carry the transparency of suspensions to a target level. If suspending construction is transparent to language specification, or any specification for that matter, it could just as easily be transparent to implementations. It is essentially the 1980 design that is reported here, although the implementation has since been superficially revised. The sequential machine that interprets Daisy cannot access suspended computations, except to create them. A probe must resolve to a value before it arrives in a register. Thus, programming at machine-level in DSI is just as oblivious to the presence of processes as is programming in Daisy.

The hardware goals in this work are far less ambitious than a LiMP. DSI is now implemented with multitasking in 'C' and assembly, a successful *head* takes roughly fifty instruction bytes on a VAX host. It could probably be done in ten to twenty bytes, three to five instructions, to verify the operand, retrieve the content, test the exists-bit, and branch around the escape for suspension-access. All this could be done with parallel hardware monitoring transactions with memory. It is a near-term goal implement this mechanism, STINGS, and Figure 3 for a single-processor system.

LiMP is a metaphor to help us think beyond hardware archetypes. Though its physical realization is, perhaps, plausible, its true purpose is to expose questions of general import in parallel architecture research. Some of these are summarized below.

- More attention should be given to designing parallel systems oriented toward heap manipulation. The prospective benefits include opening the way to parallel symbolic computation and reducing the intellectual effort needed to choreograph parallel storage usage. One way to dynamically resolve storage contention is to disassociate information from its location. A program advances by producing new data instead of overwriting old data. A system that balances itself according to resource allocations may avoid the serialization caused by "hot spots." Even if arrays are the best way to engineer physical storage, it does not necessarily follow that concatenating arrays is the best way to organize parallel storage, nor that vectorization of sequential processes is the best route to parallel programming.

- It is not just data that is communicated in a parallel system. Hardware is needed to support resource management. In Section 4 we have looked at allocation tactics for three resources: data, processes, and processors. The NEW-list sink and NEW-suspension sink dissipate fluctuations in consumption. This is a basic quality of dataflow architecture turned around*; outputs are routed toward producers rather than inputs toward consumers.

The logical device of buffered routing is not fundamental to LiMP behavior. Current guesses at the crossover point, at which gross performance with buffered routing might surpass circuit switching, are in systems with more than 128; and the number seems to rise with experience in implementation. It is hard to imagine engaging that many processors to advantage in a *typical* functional program. A Θ of 7 or 8 is a high price to pay for a simple iterative loop. However, even if routing is circuit-switched, a NEW sink effect can be built in by presetting paths to available sources.

- Demand propagation is an viable approach to decentralized scheduling. LiMP's communication network is also a purveyor of need, disseminating scheduling information to decentralized schedulers. In general purpose contexts, demand driven

* The mechanism similar to the NEW sink is briefly mentioned by Ackerman for dataflow computer memories [Ac77, Section 5.0.5].

computation is a dual to data-driven computation; each basks in the light of the other's dispair; each seeks design compromises in the direction of the other's advantages. The blessing of demand-drive is its inhibition of supply, its curse is the propagation of need, and its compromise is speculative computation. Parallel vehicles must find general ways to localize and distribute the judgement of when to apply processors to processes. Here too architecture must play a role.

- Attain random access to shared space. Some preliminary simulations of LiMP communication show that NEW sinks have the describe effect of balancing allocations and hence subsequent activity [Jo80]. Some experiments showed that better physical locality, supported by pairing each LPE with a preferred LSE, degrades routing performance by protracting the non-local transactions. The problem was resolved by randomizing routing priorities; making locality moot.

The simulations just mentioned modeled a cruder architecture than the LiMP described in Section 4, and they were driven by a simple intuitive estimate of LPE behavior. The continuation of this work must proceed on the three interdependent fronts named in the title. To converge on a parallel architecture for functional programming, we must know more about functional abstractions of parallelism. As speculations about architecture confront electronic realities, models of parallelism must reflect them. This work is more important in its means than its end. The effort to describe hardware support for purist languages exposes basic issues of general import.

References

[Ac77] W. B. Ackerman. *A Structure Memory for Data Flow Computers.* M. S. Thesis, Dept. of Electrical Engineering and Computer Science, MIT, 1977. Published as MIT/LCS/TR-186, MIT Laboratory for Computer Science, 1977.

[DeBoLe80] J. B. Dennis, G. A. Boughton, and C. K. C. Leung. Building blocks for data flow prototypes. *Proc. ACM-SIGARCH Seventh Annual Symp. on Computer Architecture* (1980), 1–8.

[FiFr84] Robert Filman and Daniel P. Friedman. *Coordinated Computing: Tools and Techniques for Distributed Software.* McGraw-Hill, New York, 1984.

[FrWi76a] Daniel P. Friedman and David S. Wise. `CONS` should not evaluate its arguments. In S. Michaelson and R. Milner (Eds.) *Automata, Languages and Programming,* Edinburgh University Press (Edinburgh, 1976), 257–284.

[FrWi76b] Daniel P. Friedman and David S. Wise. Output driven interpretation of recursive programs, or writing creates and destroys data structures. *Information Processing Letters* **5**(6):155–160.

[FrWi78a] Daniel P. Friedman and David S. Wise. Functional combination. *Computer Languages* **3**, 1 (1978), 31–35.

[FrWi78b] Daniel P. Friedman and David S. Wise. Aspects of applicative programming for parallel processing. *IEEE Trans. Comput.* **C-27**, 4 (April, 1978), 289–296.

[FrWi78b] Daniel P. Friedman and David S. Wise. Sting-unless: a conditional, interlock-free store instruction. in M. B. Pursley and J. B. Cruz (eds.), *Proc. 16th Annual Allerton Conf. on Communication, Control, and Computing,* University of Illinois (Urbana-Champaign, 1978), 578–584.

[FrWi78b] Daniel P. Friedman and David S. Wise. A note on conditional expressions. *Comm. ACM* **21**(1):931–933.

[FrWi80a] Daniel P. Friedman and David S. Wise. An indeterminate constructor for applicative multiprogramming. *Record 7th ACM Symp. on Principles of Programming Languages* (January, 1980), 245–250.

[FrWi80b] Daniel P. Friedman and David S. Wise. Fancy ferns require little care. in S. Holmstrom, B. Nordstrom, and A Wikstrom (eds.), *Symp. on Functional Languages and Computer Architecture,* Laboratory for Programming Methodology, Goteborg, Sweden, 1981.

[Hall87] Cordelia Hall. Compiling strictness into streams. *Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages,* 1987, 132–144.

[Ha85] Robert H. Halstead, Jr.. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems* **7**:4(501–538) (1985).

[He76] Peter Henderson and James H. Morris. A lazy evaluator. *Proc. 3rd ACM Symposium on Principles of Programming Languages,* 1976, 95–103.

[He80] Peter Henderson. *Functional Programming: Application and Implementation.* Prentice-Hall, Englewood Cliffs, New Jersey, 1980

[Jo77] Steven D. Johnson. *An Interpretive Model for a Language Based on Suspending Construction.* M.S. Thesis, Indiana Univeristy Computer Science Dept., Bloomington, 1977.

[Jo85] Steven D. Johnson. Storage allocation for list multiprocessing. Technical Report No. 168, Computer Science Dept., Indiana University (March, 1985).

[Jo84] Steven D. Johnson. *Synthesis of Digital Designs from Recursion Equations.* The MIT Press, 1984.

[Jo81] Steven D. Johnson. Connection Networks for Output-Driven List Multiprocessing. Indiana University Computer Science Dept. Technical Report No. 114, (1981).

[JoKo81] Steven D. Johnson and Anne T. Kohstaedt. DSI Program Description. Indiana University Computer Science Dept. Technical Report No. 120, (1981).

[Sm78] B. J. Smith. A pipelined, shared resource MIMD computer. in *Proc. International Conf. on Parallel Processing, 1978.*

[TrLi79] A. R. Tripathy and G.J. Lipovski. Packet switching in banyan networks. *Proc. ACM-SIGARCH Sixth Annual Symp. on Computer Architecture* (1979) 160–167.

[Ve84] Steven R. Vegdahl. A survey of proposed architectures for the execution of functional languages. *IEEE Transactions of Computers* **C-33**:12(1050–1071).

[WaAs86] Wadge and Aschroft. *Lucid, the Data Flow Programming Language.* 1986.

[Wa82] Mitchell Wand. Semantics-Directed Machine Architecture. *Conf. Rec. 9th ACM Symp. on Principles of Programming Languages* (1982) 234–241.

[Wi82] David S. Wise. Interpreters for functional programming. In Darlington, J., Henderson P., and Turner, D. (eds.), *Functional Programming and its Applications,* Cambridge University Press (1982), 253–280.

[Wi85a] David S. Wise. Representing matrices a quadtrees for parallel processors. *Information Processing Letters* **20** (May, 1985) 195–199.

[Wi85b] David S. Wise. Design for a multiprocessing heap with on-board reference counting. in J.-P. Jounnaud (ed.), *Functional Programming Languages and Computer Architecture,* Berlin, Springer (1985), 289–304. Representing matrices a quadtrees for parallel processors, *Information Processing Letters* **20** (May, 1985) 195–199.

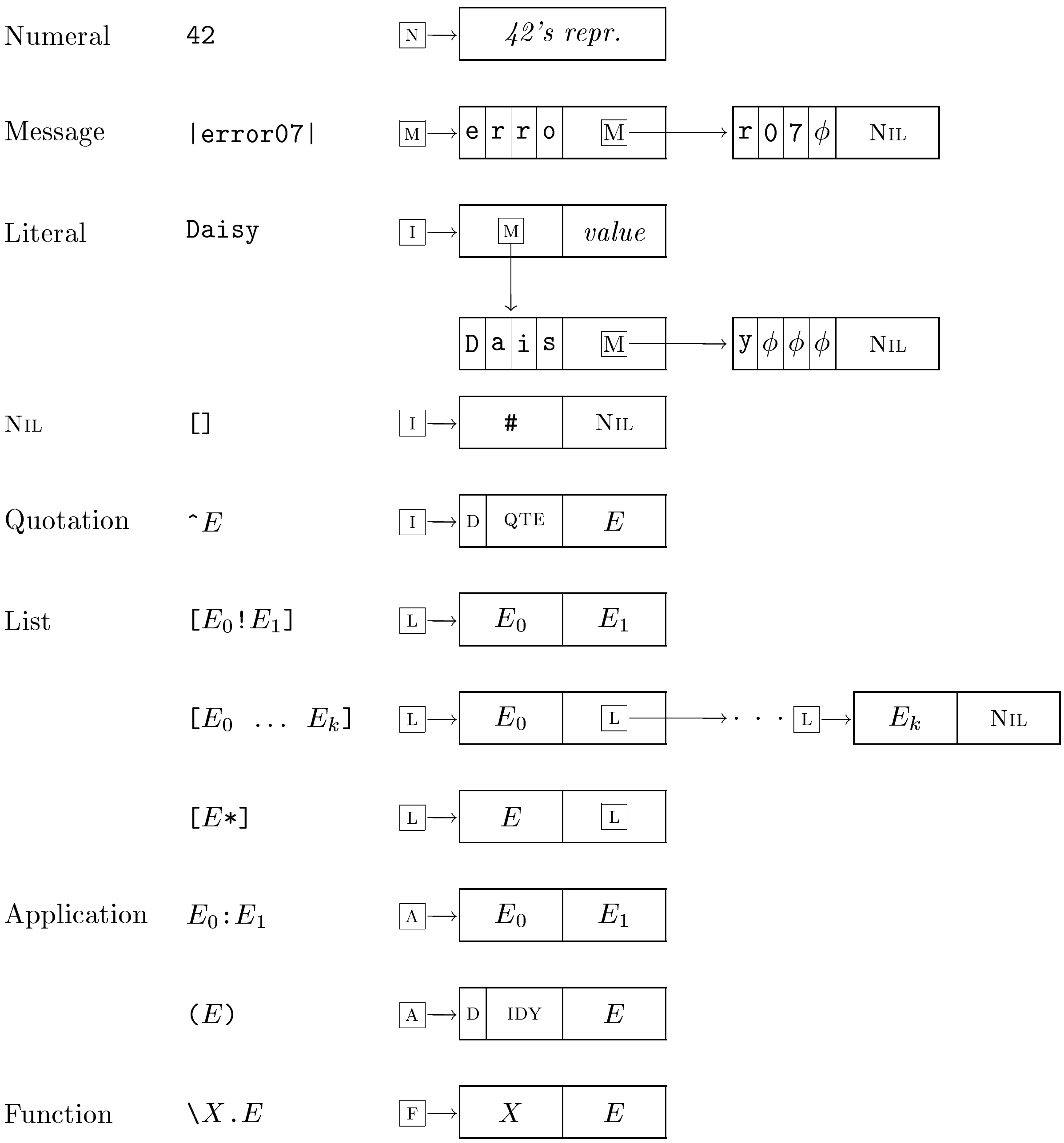


FIGURE 1. **Daisy Expressions and their Representations.**
In the second column, variables E and X are expressions. Function parameter X is expected to be a tree of identifiers.

$$\langle\!\langle D \rangle\!\rangle_\rho = \mathrm{D}$$
$$\langle\!\langle N \rangle\!\rangle_\rho = N$$
$$\langle\!\langle \texttt{[]} \rangle\!\rangle_\rho = \texttt{[]}$$
$$\langle\!\langle \texttt{^}E \rangle\!\rangle_\rho = E$$
$$\langle\!\langle \texttt{(}E\texttt{)} \rangle\!\rangle_\rho = \langle\!\langle E \rangle\!\rangle_\rho$$

$$\langle\!\langle I \rangle\!\rangle_\rho = \begin{cases} v_I, & \text{if } v_I \text{ has been globally assigned to } I; \\ \rho(I), & \text{otherwise.} \end{cases}$$

$$\langle\!\langle E_f : E_a \rangle\!\rangle_\rho = (\mathcal{A}\langle\!\langle E_f \rangle\!\rangle_\rho)(\langle\!\langle E_a \rangle\!\rangle_\rho)\,\rho$$
$$\langle\!\langle \backslash X\,.\,E \rangle\!\rangle_\rho = \texttt{[closure}\ \rho\ X\ E\texttt{]}$$
$$\langle\!\langle \texttt{[}E_0\ E_1\ \ldots\ E_n\texttt{]} \rangle\!\rangle_\rho = \texttt{[}\langle\!\langle E_0 \rangle\!\rangle_\rho\ \langle\!\langle E_1 \rangle\!\rangle_\rho \ldots \langle\!\langle E_n \rangle\!\rangle_\rho\texttt{]}$$
$$\langle\!\langle \texttt{[}E_h\ \texttt{!}\ E_t\texttt{]} \rangle\!\rangle_\rho = \texttt{[}\langle\!\langle E_h \rangle\!\rangle_\rho\ \texttt{!}\ \langle\!\langle E_t \rangle\!\rangle_\rho\texttt{]}$$
$$\langle\!\langle \texttt{[}E_0\ \texttt{*]} \rangle\!\rangle_\rho = \texttt{[}\langle\!\langle E_0 \rangle\!\rangle_\rho\ \texttt{*]}$$

$$\mathcal{A}\texttt{add} = \lambda\,\texttt{[}u\ \ v\texttt{]}\,\rho.(u+v)$$
$$\mathcal{A}\texttt{val} = \lambda\, v\, \rho\ .\ \langle\!\langle v \rangle\!\rangle_\rho$$
$$\mathcal{A}\,N = \lambda\,\texttt{[}v_0\ \ v_1\ \ \ldots\texttt{]}\,\rho\ .\ v_{\mathrm{N}}$$
$$\mathcal{A}\texttt{[closure}\ \ \rho'\ \ X\ \ E\texttt{]} = \lambda\, v\, \rho\ .\ \langle\!\langle E \rangle\!\rangle \rho' \begin{bmatrix} v \\ X \end{bmatrix}$$

$$\mathcal{A}\texttt{let} = \lambda\,\texttt{[}X\ \ E_A\ \ E\texttt{]}\,\rho\ .\ \langle\!\langle E \rangle\!\rangle \rho \begin{bmatrix} \langle\!\langle E_A \rangle\!\rangle_\rho \\ X \end{bmatrix}$$

$$\mathcal{A}\texttt{rec} = \lambda\,\texttt{[}X\ \ E_A\ \ E\texttt{]}\,\rho\ .\ \langle\!\langle E \rangle\!\rangle \rho',\ \textit{where}\ \begin{cases} \rho' = \rho \begin{bmatrix} v \\ X \end{bmatrix} \\ v = \langle\!\langle E_0 \rangle\!\rangle_{\rho'} \end{cases}$$

$$\mathcal{A}\texttt{if} = \lambda\,\texttt{[}p_0\ \ v_0\ \ \ldots\ \ p_n\ \ v_n\ \ w\texttt{]}\,\rho\ .\ \begin{cases} v_i, & \text{if } p_i \neq \texttt{[]} \text{ and } p_j = \texttt{[]} \text{ for } j < i, \\ w, & \text{if } p_i = \texttt{[]} \text{ for all } i. \end{cases}$$

$$\mathcal{A}\texttt{[}v_0\ \texttt{!}\ v_1\texttt{]} = \lambda\,\texttt{[[}u_0^1\ \texttt{!}\ u_1^1\texttt{]}\ldots\texttt{]}\,\rho\ .\ \texttt{[}\ \ (\mathcal{A}v_0\,\texttt{[}u_0^1\ldots\texttt{]}\,\rho)\ \ \texttt{!}\ \ \ (\mathcal{A}v_1\,\texttt{[}u_1^1\ldots\texttt{]}\,\rho)\ \ \texttt{]}$$

FIGURE 2. **The Daisy Language**

$\langle\!\langle E \rangle\!\rangle_\rho$ is the value of expression E in environment ρ. Variables D stand for directives, N for numerals, I for literal identifiers, E for expressions, and X for function arguments, lower case variables stand for values. Environment extension is expressed as $\rho\begin{bmatrix} \star \\ \star \end{bmatrix}$. The formula $\lambda\,\texttt{[}x\ \ y\texttt{]}.\mathcal{E}$ says the argument is a list of two elements, x and y.

$VALUE(Exp, Env, Cmd, Ctx)$:
per $Tag(Exp)$ **viz**

$\boxed{\text{D}}$: $CONTINUE(Exp, \texttt{[]}, Cmd, Ctx)$
$\boxed{\text{N}}$: $CONTINUE(Exp, \texttt{[]}, Cmd, Ctx)$
$\boxed{\text{M}}$: {Error}
$\boxed{\text{I}}$: $LOOKUP(Exp, Env, Cmd, Ctx)$
$\boxed{\text{F}}$: $CONTINUE(\texttt{[closure}\ Env\ \ Exp\texttt{]}, \texttt{[]}, Cmd, Ctx)$
$\boxed{\text{L}}$: $LIST(Exp, Env, Cmd, Ctx)$
$\boxed{\text{A}}$: $VALUE(head(Exp), Env, \texttt{[arg}\ Exp\ \ Env\ \ Cmd\texttt{]}, Ctx)$

$LIST(Val, Env, Cmd, Ctx)$: $CONTINUE(cons(Val, Env), \texttt{[]}, Cmd, Ctx)$

$$\left\{ cons(Val, Env) = \begin{array}{l} \texttt{[}\ suspend(VALUE_0(Val, Env,\ \texttt{[converge]}, \star) \\ \texttt{!}\ suspend(VALUE_1(Val, Env,\ \texttt{[converge]}, \star)\texttt{]} \end{array} \right\}$$

$VALUE_0(Val, Env, Cmd, Ctx)$: $VALUE(head(Val), Env, Cmd, Ctx)$

$VALUE_1(Val, Env, Cmd, Ctx)$: $VALUE(tail(Val), Env, Cmd, Ctx)$

$APPLY(FVal, AVal, Env, Cmd, Ctx)$:
per $Tag(FVal)$ **viz**

$\boxed{\text{D}}$: $EXECUTE(FVal, AVal, Env, Cmd, Ctx)$
$\boxed{\text{N}}$: {Indexing operation}
$\boxed{\text{M}}$: {Error}
$\boxed{\text{I}}$: {Error}
$\boxed{\text{A}}$: {Error}
$\boxed{\text{L}}$: {Construction Functional}
$\boxed{\text{F}}$: $VALUE(TextOf(FVal), ExtendClosure(FVal, AVal), \texttt{[restore}\ Env\ \ Cmd\texttt{]}, Ctx)$

$CONTINUE(Val, Env, Cmd, Ctx)$:
per Cmd **viz**
`[converge]`: {store Val at Ctx}
$\texttt{[arg}\ Exp\ \ Env'\ \ Cmd'\texttt{]}$: $VALUE(tail(Exp), Env', \texttt{[apply}\ Val\ \ Env'\ \ Cmd'\texttt{]}, Ctx)$
$\texttt{[apply}\ Val'\ \ Env'\ \ Cmd'\texttt{]}$: $APPLY(Val', Val, Env', Cmd, Ctx)$
$\texttt{[restore}\ Env'\ \ Cmd'\texttt{]}$: $CONTINUE(Val, Env', Cmd', Ctx)$

FIGURE 3. **Daisy Interpreter Outline.**