How Daisy is Lazy:
Suspending Construction at Target Levels

Steven D. Johnson
Computer Science Department
Indiana University

Abstract

This paper develops a virtual machine description supporting the programming language *Daisy*, a lazy descendent of Pure Lisp. The derivation isolates the sequential task entity for concurrent graph reduction based on *suspending construction*, a demand-driven scheme in which task creation is a transparent byproduct of record creation. This development carries the slogan, *"lazy machines make languages lazy,"* from the level of metainterpretation to the level of a compilation target. Key instructions are developed, and scheduling is abstractly considered from the task entity's point of view. The treatment is a prelude, both to a future compiler definition, and to an orthogonal study of scheduling. Here, the main purpose is description, and the motive is to propose a modeling framework for comparing related constructs, such as *delays, futures,* and *engines*.

1. Survey

Daisy exists to explore concurrent graph reduction based on *suspending construction*. In this paper, its underlying task model is isolated from details of language implementation on the one hand, and from task scheduling on the other. In the process, the implementation of suspensions is explained. An underlying motive is to establish a modeling framework for comparing related constructs, such as *delays, futures,* and *engines*. Subtle operational differences among these kinds of objects should be characterized, in order to find good generalizations for symbolic multiprocessing architecture.

Daisy is a lazy list processing language, implemented by a graph reduction interpreter. "Lazy" modifies "list" in the previous sentence. In [Friedman&Wise76], Friedman and Wise presented an interpreter, for Pure Lisp in Lisp, in which the primitive list operations are altered to make list construction non-strict. `CONS` initializes the fields of new cells with *suspended* evaluations. Suspensions are coerced to values by the list-access primitives, The result is a call-by-name semantics, safely optimized to call-by-need if there no side-effects.

From the outset, suspensions were considered as tasks; their transparency made them attractive for parallelism, especially in functional programming (e.g. [Friedman&Wise78a]), and also led to a data oriented treatment of indeterminacy (e.g. [Friedman&Wise80]). A graph multitasking system, with Daisy as the surface language, was developed to demonstrate and explore these results.

The starting point is an abbreviated language definition in Section 2. Though language implementation certainly influenced suspension representation, it is orthogonal to their implementation. This orthogonality is fundamental to the work: a lazy (or concurrent, or indeterminate, *etc.*) `CONS` *makes* Lisp lazy (or concurrent, or indeterminate, *etc.*); and similarly for other graph processing languages.

The exposition carries this principle below the level of language interpretation: A compiled language L inherits property P from the target machine. In Section 3, data operations are developed, in the course of refinement to store semantics, by an *ad hoc* extraction of combinators. An abstraction of global concurrency is then introduced and informally discussed. Between the language implementation and the scheduler lies a sequential task entity. Section 4 defines key instructions of its command oriented assembly language.

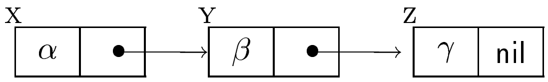
The treatment mimics Wand's method of compiler construction, where operations are identified as combinators [Wand82]. It also mimics Clinger's use of functions on state to model instructions [Clinger84]. However, these are metasyntactic comparisons because a compiler is neither defined nor proven. This work represents a step toward characterizing a concurrent target machine by precisely stating the interface between local and global control. This is a prerequisite to a compiler definition.

The conclusion explains the degree of transparency at the target machine level, sets the scene for future treatment of scheduling, and explains the direction of this work. Throughout, the paper discusses how *suspensions* differ operationally from related task constructs.

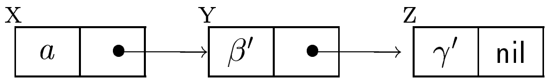
2. Daisy

Figures 1 through 3, defining the core of the language, are typical for the most part. Errors are values and function closures are modeled as objects, interpreted in application. By convention, a variable is reserved for each semantic type; elementary coercions, injections, and projections are suppressed. The underlying formulas in the final paper will be mechanically type-checked. Figure 4 illustrates most of the syntax presented, and shows that functions are values, scoping is lexical, and data structures may be recursively defined. A more complete description of Daisy is given in [Johnson88].

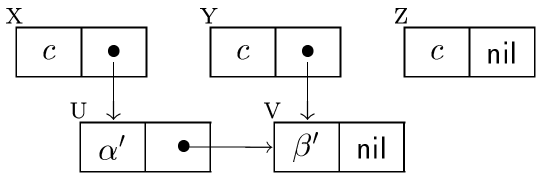
Equation (9) of Figure 3, for multisets, is intuitive and could not be used in equational reasoning. Formalizing Daisy's indeterminacy is beyond this paper's scope, but the operational treatment is an important factor later. Here is a sketch of how multisets work: an expression `{A B C}` produces a list structure containing the computations, α for `A`, β for `B`, and γ for `C`:



Should list object X be accessed, α, β, and γ execute concurrently. Should α be the first to produce a value, a, the effect is



Value a is the result of the access; β' and γ' are continuations of the other computations. Should γ converge first to c, the effect is



Since there may be external references, c is recorded in all the original cells, and both α' and β' are moved to new locations. More details can be found in [Friedman&Wise80].

The possibility that suspensions can change location influences their representation and distinguishes them from other task entities.

Daisy is a toy language, a simple interpreter coded on the machine characterized later. A parser specification would be almost as good a starting point, except that parsing had less influence on process design. As space permits, I will make brief remarks about the language and its various flaws. For instance, Equation (13) in Figure 3 states that application is strict—this mistake is retained as a reminder that laziness comes from construction—and hence prohibits undefined atoms [Cartwright&Donahue82].

2.1. Environments and Trajectories

Lambda parameters are identifier trees, and environment extension is

$$\begin{aligned}
\rho\left[{v \atop I}\right](J) &= \begin{cases} v, & \text{if } I = J \\ \rho(J), & \text{if } I \neq J \end{cases} \\
\rho\left[{v \atop \texttt{[]}}\right](J) &= \rho(J) \\
\rho\left[{v \atop \texttt{[}\,X\ \texttt{!}\ Y\,\texttt{]}}\right](J) &= \rho\left[{tail(v) \atop Y}\right]\left[{head(v) \atop X}\right](J)
\end{aligned}$$

In words, an identifier is bound at its position in the formal parameter. Environment look-up has two phases. First, the formal parameter is searched for the left most occurrence of the sought identifier. This would be done at compile-time if Daisy had a compiler. The search produces a *trajectory* or linear composition of *head*s and *tail*s, which is applied to the actuals. In what follows, it suffices to consider these implicit list accesses.

3. Suspensions

The development centers on determinate list expressions, and the effect of the environment probe *head*. By caveat, we define

$$\mathcal{V}[\![\, \texttt{[}E_0\ \texttt{!}\ E_1\texttt{]}\,]\!]\rho = cons\,\big(\mathcal{V}_\rho[\![\, E_0 \,]\!]\big)\,\big(\mathcal{V}_\rho[\![\, E_1' \,]\!]\big) \overset{\text{def}}{=} \langle\, \mathcal{V}_\rho[\![\, E_0 \,]\!] \,.\, \mathcal{V}_\rho[\![\, E_1' \,]\!] \,\rangle$$

The implementation of lists is to satisfy

$$\begin{gathered}
cons\, v_0\, v_1 \neq \perp_{\text{Val}}, \\
head\big(cons\, v_0\, v_1\big) \equiv v_0 \\
tail\big(cons\, v_0\, v_1\big) \equiv v_1
\end{gathered}$$

To accomplish this, evaluations are developed as objects. $\mathcal{V}$ is refined to a control semantics, then to a store semantics, and finally to a "task semantics."

3.1. Thunks are not Suspensions

Variables κ range over expression continuations, $\kappa\colon Kon = Val \to Val$ The *cons* combinator, which takes *expression closures* (or thunks), $\phi\colon Kon \to Val$, can be can be re-written in three other ways to be strict. In fact, Daisy has four versions of *cons*.

$$cons\, \phi_0\, \phi_1\, \kappa = \kappa\langle \phi_0, \phi_1 \rangle.$$

Head: $Val \to Kon \to Val$ becomes

$$head\, v\, \kappa = \begin{cases} \kappa\ error, & \text{if } v \text{ is not a list,} \\ (p_0)\kappa, & \text{otherwise, where } p = v|_{\mathrm{Lst}}. \end{cases}$$

3.2. Delays are not Suspensions

Getting from call-by-name to call-by-need requires a store, where evaluations, like *delays,* record their results [Henderson&Morris76]. Lists are simply two-delay records. Stores map locations (ℓ) to binary pairs (p), whose two fields contain either values or delays. Stores come with an allocate operation, $new\colon Mem \to Loc \times Mem$, and an update operation, denoted $\sigma[{}^{p}_{\ell}]$. New locations are always initialized, so that the pattern of allocation always looks like

$$\cdots\ \ell\ \cdots\ \sigma'\left[{}^{p}_{\ell}\right]\ \cdots \quad \text{where} \quad \left[\begin{array}{l} p = \langle \star, \star \rangle \\ \langle \ell, \sigma' \rangle = new(\sigma) \end{array}\right.$$

This says that the new location ℓ has been initialized to the list $\langle \star, \star \rangle$. Since Daisy is founded on a binary list-space, it is convenient to augment the notation further. A *context,* χ, specifies a location and a field. For location ℓ the two contexts are written as ℓ_0 and ℓ_1. Field accesses and updates are expressed as follows.

$$\sigma\, \ell_0 \quad \textit{really means} \quad (\sigma\, \ell\,)_0$$

$$\sigma\left[{}^{v}_{\ell_0}\right] \quad \textit{really means} \quad \sigma\left[{}^{p}_{\ell}\right] \text{ where } p = \langle v, \sigma(\ell_1) \rangle$$

Adjusting the domains,

$Val = Atm + Loc$	(v)	Values
Loc	(ℓ)	Locations
$Ctx = Loc + Loc$	(χ, ℓ_0, ℓ_1)	Contexts
$Mem = Loc \to Lst$	(σ)	(List) Stores
$Lst = (Del + Val) \times (Del + Val)$	(p)	Lists
$Del = Kon \to Mem \to Val$	(δ)	Delays
$Kon = Val \to Mem \to Val$	(κ)	Continuations

Now at this point, $\mathcal{V}_{\text{Env}}\colon \mathit{Exp} \to \mathit{Del}$.

$$\mathcal{V}_\rho[\![\ \mathtt{[}E_0\ \mathtt{!}\ E_1\mathtt{]}\ ]\!]\kappa\sigma = \mathit{cons}(\mathcal{V}_\rho[\![\, E_0\,]\!])(\mathcal{V}_\rho[\![\, E_1\,]\!])\kappa\sigma$$

Accordingly, *cons* allocates a list.

$$\mathit{cons}\,\delta_0\,\delta_1\,\kappa\,\sigma = \kappa\,\ell\,\sigma'\big[{}^{p}_{\ell}\big] \text{ where } \left[\begin{array}{l}\langle \ell, \sigma'\rangle = \mathit{new}(\sigma)\\ p = \langle\,(\mathit{delay}_0\,\delta_0),\ (\mathit{delay}_1\,\delta_1)\,\rangle\end{array}\right.$$

$\mathit{delay}_\star\colon \mathit{Del} \to \mathit{Loc} \to \mathit{Del}$ imposes an update-continuation,

$$\mathit{delay}_0\,\delta\,\ell = \lambda\kappa\sigma\ .\ \delta(\mathit{converge}\,\ell_0\,\kappa)\sigma$$

$\mathit{converge}\colon \mathit{Ctx} \to \mathit{Kon} \to \mathit{Kon}$ updates the store,

$$\mathit{converge}\,\chi\,\kappa = \lambda\,v\,\sigma.\ \kappa v\sigma\big[{}^{v}_{\chi}\big]$$

and $\mathit{head}\colon \mathit{Val} \to \mathit{Kon} \to \mathit{Mem} \to \mathit{Val}$ becomes

$$\mathit{head}\,v\,\kappa\,\sigma = \begin{cases}\kappa\ \mathit{error}\ \sigma, & \text{if } v \text{ is not a list location,}\\ \kappa(\sigma\,\ell_0)\sigma, & \text{if } (\sigma\,\ell_0) \text{ is a value,}\\ (\sigma\,\ell_0)\,\kappa\,\sigma, & \text{if } (\sigma\,\ell_0) \text{ is a suspension.}\end{cases} \qquad \text{where } \ell = v|_{\text{Loc}}$$

In words, *head* verifies that its operand is a location, and returns the value, if present, at that location. Otherwise, the computation there is invoked. The resulting value is stored at ℓ_0 and also returned to the original caller. This level of description is accurate in terms of the representations used. Suspensions are distinguished from values by a bit, located in the list cell that holds them. When the evaluation done by a suspension concludes, the suspension's reference is overwritten with a value and the bit inverted. The *head* primitive verifies this bit before returning a value. One consequence is that no computation can retrieve a suspension; this is one difference with both delays and futures.

3.3. Suspensions as Tasks Entities

In developing concurrency, it is necessary to account for the preemption and scheduling of computations. This paper is concerned with the task entity's view of global coordination. Scheduling is informally discussed, with details left for a future paper. Schedules $\Sigma\colon \mathit{Sdl}$ may be thought of sets of contexts, with operations

$$\begin{array}{l}\mathit{put}\colon \mathit{Ctx} \to \mathit{Sdl} \to \mathit{Sdl}\\ \mathit{get}\colon \mathit{Sdl} \to \mathit{Ctx} \times \mathit{Sdl}\end{array}$$

for adding and deleting elements. In implementation, schedules include a means of prioritization, based on dependence and I/O events. Ideally, *get* produces a good choice according to demand, as measured by proximity to an output task.

The function *run* invokes a task; and *swap* does a task swap. *Run* must check whether a scheduled task has already converged.

$$\begin{aligned}
&\mathit{run}\colon \mathit{Mem} \to \mathit{Sdl} \to \mathit{Val}\\
&\mathit{run}\sigma\Sigma = \begin{cases} \mathit{run}\sigma\Sigma', & \text{if } \sigma\,\chi \text{ is a value,}\\ (\sigma\chi)\chi\sigma\Sigma', & \text{if } \sigma\,\chi \text{ is a suspension} \end{cases} \qquad \text{where } \langle\chi, \Sigma'\rangle = \mathit{get}\,\Sigma\\
&\mathit{swap}\colon \mathit{Ctx} \to \mathit{Mem} \to \mathit{Sdl} \to \mathit{Val}\\
&\mathit{swap}\,\chi\,\sigma\,\Sigma = \mathit{run}\,\sigma\,(\mathit{put}\,\chi\,\Sigma)
\end{aligned}$$

The last refinement to suspensions can be inferred from *run*. It is that the suspension's context—the list object in which it resides—is supplied at the point of invocation. This is another difference with futures, accounting for the possibility that suspensions may be moved, as is the case in multiset evaluation (Section 2). Fixing the relevant domains again,

$$\begin{array}{lll}
\mathit{Spn} = \mathit{Ctx} \to \mathit{Mem} \to \mathit{Sdl} \to \mathit{Val} & (\phi) & \text{suspensions}\\
\mathit{Kon} = \mathit{Val} \to \mathit{Spn} & (\kappa) & \text{continuations}\\
\mathit{Sdl} \approx \ \mathit{sets\ of\ contexts} & (\Sigma) & \text{schedules}
\end{array}$$

Now, the constructor suspends as before, but the definition below also describes an indeterminate "time-out," once the new cell has been allocated. *Cons* takes two evaluation closures, ψ_0, ψ_1: $\mathit{Kon} \to \mathit{Spn}$. *Suspend* simply initializes a closure's continuation, thus making it into a suspension. *Converge* updates the current context and invokes the scheduler.

$$\begin{aligned}
&\mathit{cons}\,\psi_0\,\psi_1\,\kappa\,\chi\,\sigma\,\Sigma = \begin{cases} \mathit{swap}\,\chi\,\sigma'\big[{p \atop \ell}\big]\big[{\kappa\,p \atop \chi}\big]\,\Sigma,\\ \kappa\,\ell\,\chi\,\sigma'\big[{p \atop \ell}\big]\Sigma \end{cases} \quad \text{where } \left[\begin{array}{l} \langle\ell, \sigma'\rangle = \mathit{new}\,\sigma\\ p = \langle(\mathit{suspend}\,\phi_0), (\mathit{suspend}\,\phi_1)\rangle \end{array}\right.\\
&\mathit{suspend}\,\psi = \psi(\mathit{converge})\\
&\mathit{converge}\,v\,\chi\,\sigma\,\Sigma = \mathit{run}\,\sigma'\,\Sigma \text{ where } \sigma' = \sigma\big[{v \atop \chi}\big]
\end{aligned}$$

Finally, *head* too may be interrupted for a task swap; it may find a value; or it may spawn a dependent task.

$$head\, v\, \kappa\, \chi\, \sigma\, \Sigma = \begin{cases} swap\, \chi\, \sigma\left[{\phi \atop \chi}\right]\, \Sigma, & \text{arbitrarily,} \\ \kappa\; error\, \chi\, \sigma\, \Sigma & \text{if } v \text{ is not a location,} \\ \kappa(\sigma\ell_0)\, \chi\, \sigma\, \Sigma, & \text{provided } (\sigma\ell_0) \text{ is a value,} \\ (\sigma\ell_0)\, \ell_0\, \sigma\left[{\phi \atop \chi}\right](put\, \chi\Sigma), & \text{provided } (\sigma\ell_0) \text{ is a suspension.} \end{cases}$$

$$\text{where} \left[\begin{array}{l} \phi = head\, \ell\, \kappa \\ \ell = v|_{\mathrm{Loc}} \end{array}\right.$$

3.4. Conditional Stores and Speculative Computation

Since they describe a single thread of control, the descriptions are adequate for multitasking but not necessarily for parallel execution. The contentions that arise are resolved if the store prohibits overwriting values. This is the *sting* operation, described by Friedman and Wise [Friedman&Wise78b, Wise85].

$$\sigma\left[{v \atop \ell_0}\right] = \begin{cases} \sigma, & \text{if } \sigma\, \ell_0 \text{ is a value,} \\ \sigma\left[{p \atop \ell_0}\right], & \text{if } \sigma\, \ell_0 \text{ is a suspension.} \end{cases} \quad \text{where } p = \langle v, \sigma(\ell_1)\rangle$$

Speculative parallelism would be introduced in *cons*, by adding the newly created contexts to the schedule:

$$cons\, \psi_0\, \psi_1\, \kappa\, \chi\, \sigma\, \Sigma = swap\, \chi\, \sigma'\left[{p \atop \ell}\right]\left[{\kappa\, p \atop \chi}\right]\, \Sigma',$$

$$\text{where} \left[\begin{array}{l} \langle \ell, \sigma' \rangle = new\, \sigma \\ p = \langle (suspend\, \phi_0), (suspend\, \phi_1)\rangle \\ \Sigma' = put\, \chi\, (put\, \ell_0\, (put\, \ell_1\, \Sigma)) \end{array}\right.$$

This description is, indeed, speculative since no ordering on Σ has been described.

4. An Abstract Processor

This section sketches instructions for a sequential-control processor. A task *state* R: *Registers* is modeled as a tuple of n values and a command continuation, each denoted by $R_1, \ldots, R_n, R_{\mathrm{cc}}$. Register assignment is expressed $R[i \leftarrow v]$.

A program π: *Reg* $\rightarrow$ *Spn* maps the local plus global state to a value; recall that $\mathit{Spn} = \mathit{Ctx} \rightarrow \mathit{Mem} \rightarrow \mathit{Sdl} \rightarrow \mathit{Val}$. A concrete syntax is used to express these functions, in which a semicolon stands for serial composition.

The command continuation (or control stack) γ is represented by a finite string of programs. For our purposes, it is enough to write $\gamma = \langle \pi, \gamma' \rangle$, in order to "push" a program, and $\langle \pi, \gamma' \rangle = \gamma$, in order to "pop" one.

- If π is (denoted by) $[\![\, \mathtt{r}_i \ \mathtt{:=}\ \mathtt{suspend(}\pi_1\mathtt{);}\ \pi_2 \,]\!]$, then

$$\pi\, R = \pi_2\, R[i \leftarrow \phi] \quad \text{where} \quad \phi = \pi_1\, R\big[\mathrm{cc} \leftarrow \langle [\![\, \mathtt{converge}_n\, \mathtt{.} \,]\!] \rangle\big]$$

- If π is $[\![\, \mathtt{continue;}\ \pi_1 \,]\!]$, then

$$\pi\, R = \pi'\, R' \text{ where } \left[\begin{array}{l} \langle \pi', \gamma \rangle = R_{\mathrm{cc}} \\ R' = R[\mathrm{cc} \leftarrow \gamma] \end{array}\right.$$

- If π is $[\![\, \mathtt{converge}\ \mathtt{r}_i\mathtt{;}\ \pi_1 \,]\!]$, then

$$\pi\, R = \mathit{converge}\, R_i$$

- If π is $[\![\, \mathtt{r}_i \ \mathtt{:=}\ \mathtt{new(r}_j\mathtt{,}\ \mathtt{r}_k\mathtt{);}\ \pi_1 \,]\!]$, then

$$\pi\, R\chi\sigma = \pi_1\, R'\, \chi\, \sigma' \big[{}^{p}_{\ell}\big] \text{ where } \left[\begin{array}{l} \langle \ell, \sigma' \rangle = \mathit{new}\, \sigma \\ p = \langle r_j, r_k \rangle \\ R' = R[i \leftarrow \ell] \end{array}\right.$$

- The program $[\![\, \mathtt{r}_i \ \mathtt{:=}\ \mathtt{cons(}\pi_0\mathtt{,}\ \pi_1\mathtt{);}\ \pi_3 \,]\!]$ expands to

$$[\![\, \mathtt{r}_j \ \mathtt{:=}\ \mathtt{suspend(}\pi_0\mathtt{);}\ \mathtt{r}_k \ \mathtt{:=}\ \mathtt{suspend(}\pi_1\mathtt{);}\ \mathtt{r}_i \ \mathtt{:=}\ \mathtt{new(r}_j\mathtt{,}\ \mathtt{r}_k\mathtt{);}\ \pi_3 \,]\!]$$

- If π is $[\![\, \mathtt{r}_i \ \mathtt{:=}\ \mathtt{head(r}_j\mathtt{);}\ \pi_1 \,]\!]$, then

$$\pi\, R = \mathit{head}\, R_i\, (\lambda\, v\, .\, \pi_1\, R[i \leftarrow v])$$

4.1. Input and Output

Conventional I/O is integrated in the list space, with device handlers represented as special-purpose suspensions. The input and output handlers sketched below illustrate the language developed above.

```
GETC: { Obtain a character in r1 }
      r2  := suspend(GETC);
      r2  := new(r1 , r2 );
      converge2 .


PUTC: r1  := head(r2 );
      { Display the character in r1 };
      r2  := tail(r2 );
      goto PUTC.
```

5. Concluding Remarks

The refinement to an operational semantics, dealing with a store and a scheduler, carries the transparency of task dependence to the target machine level. As indicated in Section 3.4, this separation allows us to talk, in fairly precise terms, about concurrency issues, without disturbing the task model.

The paper concludes with a discussion of demand propagation. We feel that the key is to incorporate a dynamic reference count in the task state. This is similar to what is done in *engines* [Haynes&Friedman87]

References

[Cartwright&Donahue82] Robert Cartwright and James Donahue. The semantics of lazy (and industrious) evaluation. *Conference Record of the 1982 Symposium on Lisp and Functional programming* Pittsburgh, 1982, 253–264.

[Clinger84] William D. Clinger. The Scheme 311 compiler: an exercise in denotational semantics. *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming,* Austin, 1984, 356–364.

[Friedman&Wise76] 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.

[Friedman&Wise78a] Daniel P. Friedman and David S. Wise. Aspects of applicative programming for parallel processing. *IEEE Trans. Comput.* **C-27**, 4 (April, 1978), 289–296.

[Friedman&Wise78b] Daniel P. Friedman and David S. Wise. Sting-unless: a conditional, interlock-free store instruction. In M. B. Pursley and J. B. Cruz, Jr. (eds.), *Proc. 16th Annual Allerton Conf. on Communication, Control, and Computing,* University of Illinois (Urbana-Champaign, 1978), 578–584.

[Friedman&Wise80] 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.

[Haynes&Friedman87] Chris T. Haynes and Daniel P. Friedman. Abstracting timed preemption with engines. *Computer Languages*, **Vol. 12**, No. 2 (1987), 109–121..

[Halstead85] Robert H. Halstead, Jr.. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems,* **7**(4):501–538 (October, 1985).

[Henderson&Morris76] Peter Henderson and James H. Morris, Jr.. A lazy evaluator. *Conf. Rec. 3rd ACM Symp. on Principles of Programming Languages,* 1976, 95–103.

[Johnson84] Steven D. Johnson. *Synthesis of Digital Designs from Recursion Equations.* The MIT Press, Cambridge, 1984.

[Johnson88] Steven D. Johnson. Daisy Programming Manual (working title). draft in progress, available on request.

[Miller87] James S. Miller. MultiScheme: A Parallel Processing System Based on MIT Scheme. Ph.D. Dissertation, Technical Report MIT/LCS/TR-402, Massachusetts Institute of Technology, 1987.

[Wand82] Mitchell Wand. Deriving Target Code as a representation of continuation semantics. *ACM Trans. on Prog. Lang. and Systems* **4** (1982), 496–517.

[Wise85] David S. Wise. Design for a multiprocessing heap with on-board reference counting. In J.-P. Jouannaud (Ed.), *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science* **201**, Berlin, Springer (1985), 289–304.

Expressions, $E\colon Exp$

numerals, N	
operators, e.g. $\boxed{\texttt{add}}$	
identifiers, I	
$\boxed{\texttt{"}}\; I \;\boxed{\texttt{"}}$	literal quotation
$E_0 \;\boxed{\texttt{:}}\; E_1$	application expression
$\boxed{\texttt{\\}}\; X \;\boxed{\texttt{.}}\; E$	function expression
$\boxed{\texttt{[]}}$ or $\boxed{\texttt{\{\}}}$	Nil
$\boxed{\texttt{[}}\; E_0 \;\boxed{\texttt{!}}\; E_1 \;\boxed{\texttt{]}}$ or $\boxed{\texttt{\{}}\; E_0 \;\boxed{\texttt{!}}\; E_1 \;\boxed{\texttt{\}}}$	list expression
$\boxed{\texttt{rec:[}} X \;\; E_1 \;\; E_2 \boxed{\texttt{]}}$	`rec`-expression

Formal Arguments, $X\colon Arg$

- identifiers, I
- $\boxed{\texttt{[]}}$
- $\boxed{\texttt{[}}\; X_0 \;\boxed{\texttt{!}}\; X_1 \;\boxed{\texttt{]}}$

FIGURE 1 – Daisy Syntax

Semantic Domains

Atm	$(I, N, \mathsf{add}, \mathsf{nil})$	atoms
Err	$(error)$	errons
$Cls = Env \times Arg \times Exp$	$({}_\rho\backslash X\,.\,E)$	closure objects
$Val = Atm + Err + Cls + Lst$	(v)	values
$Lst = Val \times Val$	(p)	lists
$Env = Ide \to Val$	(ρ)	environments

FIGURE 2 – Semantic Types

Evaluation, $\mathcal{V}_{\text{Env}}$: $\mathit{Exp} \to \mathit{Val}$

$$\mathcal{V}_\rho[\![\, N \,]\!] = N \tag{1}$$

$$\mathcal{V}_\rho[\![\, \mathsf{add} \,]\!] = \mathsf{add} \tag{2}$$

$$\mathcal{V}_\rho[\![\, I \,]\!] = \begin{cases} v, & \text{if } v \text{ is assigned to } I; \\ \rho(\, I\,), & \text{otherwise.} \end{cases} \tag{3}$$

$$\mathcal{V}_\rho[\![\, \texttt{"}I\texttt{"} \,]\!] = I \tag{4}$$

$$\mathcal{V}_\rho[\![\, \texttt{[ ]} \,]\!] = \mathcal{V}_\rho[\![\, \texttt{\{ \}} \,]\!] = \mathsf{nil} \tag{5}$$

$$\mathcal{V}_\rho[\![\, \backslash\, X \;.\; E \,]\!] = {}_\rho\backslash X \,.\, E \tag{6}$$

$$\mathcal{V}_\rho[\![\, E_0 : E_1 \,]\!] = \mathcal{A}_\rho \left(\mathcal{V}_\rho[\![\, E_0 \,]\!]\right) \left(\mathcal{V}_\rho[\![\, E_1 \,]\!]\right) \tag{7}$$

$$\mathcal{V}_\rho[\![\, \texttt{[}\, E_0 \;\texttt{!}\; E_1 \,\texttt{]} \,]\!] = \langle \mathcal{V}_\rho[\![\, E_0 \,]\!], \mathcal{V}_\rho[\![\, E_1 \,]\!] \rangle \tag{8}$$

$$\mathcal{V}_\rho[\![\, \texttt{\{}\, E_0 \;\texttt{!}\; \texttt{\{}\, E_1 \;\texttt{!}\; E_2 \,\texttt{\}}\,\texttt{\}} \,]\!] \approx \begin{cases} \mathcal{V}_\rho[\![\, \texttt{[}\, E_0 \;\texttt{!}\; \texttt{\{}\, E_1 \;\texttt{!}\; E_2 \,\texttt{\}}\,\texttt{]} \,]\!], & \text{if } \mathcal{V}_\rho[\![\, E_0 \,]\!] \neq \perp \\ \mathcal{V}_\rho[\![\, \texttt{[}\, E_1 \;\texttt{!}\; \texttt{\{}\, E_0 \;\texttt{!}\; E_2 \,\texttt{\}}\,\texttt{]} \,]\!], & \text{if } \mathcal{V}_\rho[\![\, E_1 \,]\!] \neq \perp \end{cases} \tag{9}$$

$$\mathcal{V}_\rho[\![\, \texttt{rec:[}X \;\; E_1 \;\; E_2\texttt{]} \,]\!] = \mathcal{V}_{\rho'}[\![\, E_2 \,]\!] \text{ where } \left[\begin{array}{l} \rho' = \rho[\, {}^{v}_{X} \,] \\ v = \mathcal{V}_{\rho'}[\![\, E_1 \,]\!] \end{array}\right. \tag{10}$$

Application, $\mathcal{A}_{\text{Env}}$: $\mathit{Val} \to \mathit{Val} \to \mathit{Val}$

$$\mathcal{A}_\rho\, \mathsf{add} \,\langle\, u \;,\; v \rangle = \begin{cases} u + v & \text{if } u \text{ and } v \text{ are numbers,} \\ \mathit{error} & \text{otherwise} \end{cases} \tag{11}$$

$$\mathcal{A}_\rho \left({}_{\rho'}\backslash X \,.\, E\right) v = \mathcal{V}_{\rho''}[\![\, E \,]\!] \text{ where } \rho'' = \rho'[\, {}^{v}_{X} \,] \tag{12}$$

$$\mathcal{A}_\rho \;\perp\; v = \mathcal{A}_\rho\, u \;\perp\; = \;\perp \tag{13}$$

$$\mathcal{A}_\rho\, \mathit{error}\, v = \mathcal{A}_\rho\, u\, \mathit{error} \;\; = \;\; \mathit{error} \tag{14}$$

FIGURE 3 – Direct Semantics

```
                                          |--> COMMENTS
                                          |
WHEN = ^\[[T!Ts] [V!Vs]].                 | Almost Lucid’s ‘asa’
  if:[ T  V  WHEN:[Ts Vs] ]               |___
                                          |
MAPs = \F.                                | Map F over a stream
 rec:[ DO                                 |
       \[V ! Vs]. [F:V ! DO:Vs]           |
       DO                                 |
      ]                                   |___
                                          |
B1 = ^\[V P F R].                         | A sequential schema:
 rec:[ [ State Done Value]                |   B1(v, p, f, r)
       [ [V ! (MAPs:R):State]             |     = WHEN(Done, Value)
         (MAPs:P):State                   |   where
         (MAPs:F):State                   |     State = v ! r*(State)
        ]                                 |      Done = p*(State)
       WHEN:[Done Value]                  |     Value = f*(state)
      ]                                   |___
                                          |
FACTORIAL = ^\N.                          | Factorial as an
 B1:[ [N 1]                               |   instance of B1.
      (\[N M].zero?:N)                    |
      (\[N M].M)                          |
      (\[N M].[dcr:N mpy:[N M])           |
     ]                                    |
                                          |--> COMMENTS
```

FIGURE 4 – Program Example