DSI Program Description

by

S. D. Johnson

and

A. T. Kohlstaedt

Computer Science Department

Indiana University

Bloomington, Indiana 47405

TECHNICAL REPORT NO. 120

DSI PROGRAM DESCRIPTION

STEVEN D. JOHNSON

AND

ANNE T. KOHLSTAEDT

NOVEMBER, 1981

Research reported herein was supported, in part, by the National Science Foundation under grants number MCS77-22325.

ABSTRACT

The programming system **DSI**, being developed at Indiana University, is a basis for empirical investigations of programming and computer architecture. **DSI**'s development is motivated by a desire to explore in depth the applicative approach to programming. To do so, a pure and robust "computational medium" is needed, and **DSI** represents our model of such a vehicle. The program serves two projects. The first, called Project Alpha, has as an objective establishing a programming system on a conventional host computer, in order to experiment with applicative approaches to large problems. The goal of Project Beta is to specify, build, and measure a prototype applicative list multiprocessor.

This report is a "programmer's introduction" to **DSI**, giving an overall description of the program and reviewing the concepts that influenced its design. Discussion centers on the suspension, a transparent processing entity on which the program's design is based. An informal discussion of suspensions is followed by description of **DSI**'s overall program structure and a machine independent look at its representation of primitive objects.

To give orientation and perspective to the current state of **DSI**'s design, the remainder of the report deals with issues that guided its evolution and gives directions for further research and development. This includes a survey of related outside work as well as extensive references to material published by this department in the area of applicative programming. We conclude with a history of the implementation effort.

TABLE OF CONTENTS

1. Introduction

DSI is a programming system being developed at Indiana University. It is intended to serve as an experimental basis for continuing investigations of programming language semantics and computer architecture. Such investigations are outgrowths of a more general study of applicative programming for systems. Our purpose here is to review the underlying concepts that motivate DSI's development, and to illustrate how these concepts are implemented.

By itself, DSI is just a data-space manager. While its data-space can be manipulated only through a high level programming language, the basis of DSI's design is far removed from high level syntax. For the purpose of analogy, consider DSI's most venerable ancestor, LISP.[33] The term "LISP" actually denotes two entities: a list processing host facility and a language for the interpretation of symbolic expressions. The role of LISP "the language" as a lingua Franca for the functional programming community obscures the contribution of LISP "the list processor". LISP's facility in managing its data-space is what makes it so attractive as a machine model. In DSI the issues of language and architecture are also intertwined. In order to keep them distinct the access language has a name of its own, Daisy.[31] Daisy's interpreter uses DSI to manipulate its program and data structures.

DSI plays a part in two distinct but related research efforts, called Alpha and Beta. The primary goal of the Alpha Project is to produce an applicative programming system around the language Daisy. The Beta Project is a step toward designing a computer architecture for the efficient execution of applicative programs.

Daisy's syntax is described elsewhere.[31,26] It is a functional programming language in which the basic form of expression is a system of mutually recursive equations. Daisy has a call-by-name semantics and its interpreter computes by graph reduction. Since the language is side-effect free, call-by-name is implemented efficiently and transparently using a "suspending CONS".[8] DSI-alpha implements an underlying list processor that supports Daisy's computational mechanism.

The Alpha Project

	abstraction	implementation
control	Daisy semantics	suspending constructors
mechanism	graph reduction	DSI-alpha

At the same time, DSI is a design model for a prototype applicative multiprocessor. Implemented on a conventional host machine, DSI-beta emulates a target architecture that

exploits the non-sequential character of functional programs. Techniques to introduce concurrency have evolved from the "demand driven" approach to computation.[30] Design alternatives are implemented and tested with software in DSI-beta, eventually to be realized in the hardware of a list multiprocessor.

The Beta Project

	abstraction	implementation
control	DSI-beta	uniprocessor emulator
mechanism	demand driven computation	list multiprocessor

Thus DSI is in the intersection of two research efforts, and plays a different role in each. While Daisy's semantics guides DSI-alpha's implementation, its interpreter is merely a program to the underlying system. Similarly, while DSI-beta strives to emulate a multiprocessing "computational medium", it does not yet simulate a fixed architecture. While the differing roles may eventually lead to conflicting objectives, for the present DSI-alpha and DSI-beta are the same program. Below we give a "programmer's introduction" to DSI, presenting its design without delving into the surrounding issues.

Returning to the LISP analogy, we note that one intent of DSI is to do with process what LISP does with data. LISP "the list processor" is a primeval data management system: it "factors out" some of the complexity of data manipulation by reducing structure to an elemental form -- the binary list cell. Similarly, we seek a "lowest common denominator" for the notion of process. The focal point of our discussion is the suspension, a kinetic counterpart to the inert list cell. Where, in our view, data is fixed and immutable, suspensions evolve in the presence of processing.

In managing a data-space enriched with suspensions DSI must manage process as well. One of the tenets of our work is that process is embedded in data. Thus, aspects of program behavior, nondeterminism for example, arise instead as attributes of data. It becomes the responsibility of the underlying system, DSI, to manage events only weakly specified by programs.

At a higher level of discussion we would elect to keep any notion of a processing object hidden. However, here we will concentrate on these transparent entities. The description of DSI begins with a informal discussion of suspensions in Section 2, which also serves to introduce a vocabulary. Section 3 considers DSI's overall design and discusses representation. In Section 4 some extensions are proposed. Section 5 discusses related work. The summary in Section 6 reviews DSI's development history.

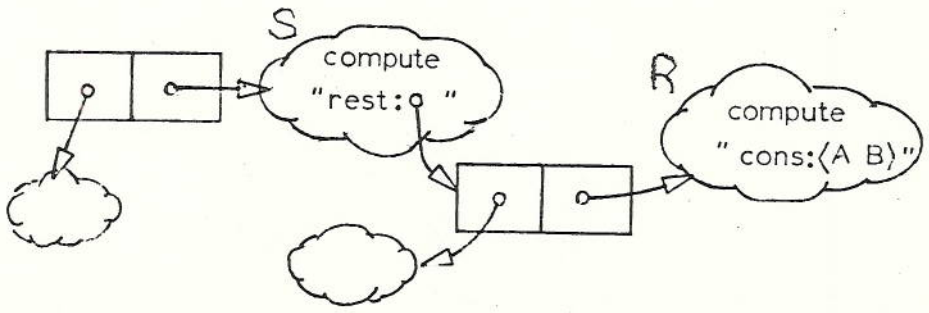
2. Suspensions

Consider the expression E = "<A ! B>". E's value is a list whose head is the value of "A" and whose tail is the value of "B". The evaluation of E involves three sub-computations:

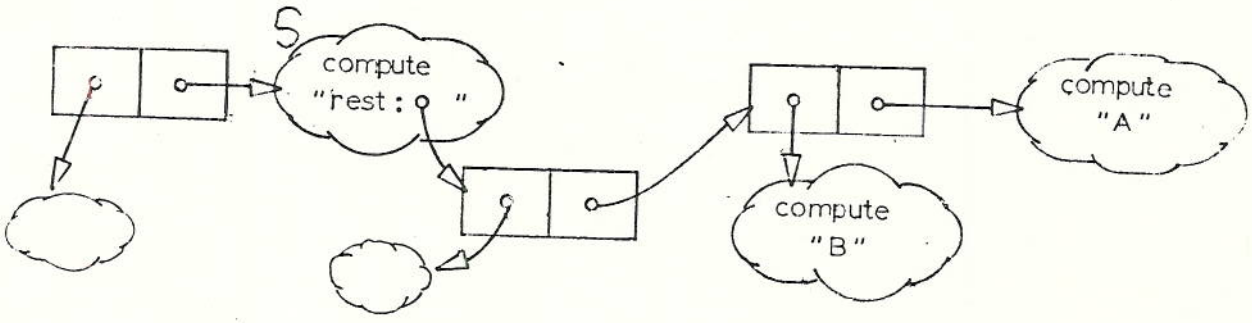
(i) Evaluate "A".
(ii) Evaluate "B".
(iii) Obtain a free list cell and initialize it with the results of (i) and (ii).

The limitations of most computers (the fact that they have only one CPU) dictate that these sub-computations occur in some order. Typically, (i) and (ii) precede (iii), and in any event all three would occur before a value was produced. This is not the case in Daisy however. No order is specified for the sub-computations, and in general the result of (iii) is returned before (i) or (ii) takes place. This behavior is obtained by distinguishing a process's creation from its execution. Computations are created and stored in the data-space.[8] When a particular computation is performed, however, depends on its relation to other computations.

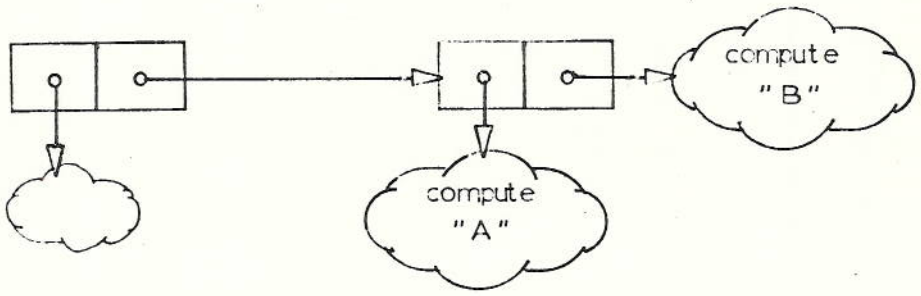
The data-space is a mixture of manifest objects and suspended computations. In pictures we depict the former as boxes and the latter as clouds.



Choose a distinguished process, say **S**, and assume it is in progress. **S**'s computation, "first:L", is a probe, which exposes the suspension **R**. This has the effect of dispatching **R** to converge, or produce a value. (If **R** diverges, or fails to produce a value, so does **S**.) Convergence has a side effect: **R** is replaced by (or replaces itself with) its value.



Should **S** ever become active again, it converges to the result of its probe.



The fundamental attribute of manifest data is its value. A process is ordinarily characterized by its state, which consists of a label (its state of behavior) and an environment (the state of its world). It is sometimes useful to think of a manifest object as a process in the (terminal) "state of being a value". In the original implementation "suspension" was a type, a record consisting of a form to evaluate and an environment. A probe was a coercion* from "type suspended" to "type manifest". In effect, each convergent object went through a length-two sequence of states.[8]

To get an operational basis for indeterminism[12] intermediate states are allowed in the sequence. An iterative

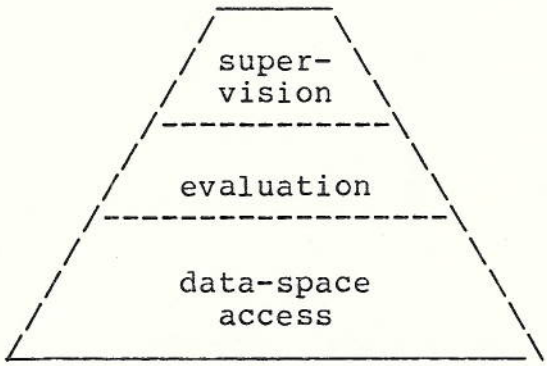
*In the technical sense of reference [35]

form of coercion, called coaxing, is introduced, where the referent suspension need not converge, but only advance its state. A new type of list-object, called a multiset, is added to the set of manifest data types. When multisets are probed their elements are coaxed concurrently until one converges. Some convergent element is promoted, via an in-place sort, to head the list.[5,9,29,27]

It is natural but misleading to describe a suspension's relation to its environment in terms of verbs like "probe" and "coerce". Actually the underlying system is the agent of change. DSI applies processors to processes: the selection of which process to activate is based on many criteria, including the dependencies established by coercions and coaxes. The process dependency relation is referred to as the schedule. In DSI-alpha, where there is only one processor available, the schedule is a data structure; but in a processor-rich system, such as DSI-beta, the distribution of the scheduling responsibility is an obvious goal.

3. Design

DSI's program design is divided into three levels of coding:



At the base level is a collection of pseudo instructions that are used to explore and manipulate the data-space. Processes -- we use the term evaluators synonymously -- are implemented at the middle level. The schedule is maintained at the supervisory level. In this section these levels are discussed individually, starting with the supervisory level. Each discussion begins with an overview, followed by a look at the Beta Project's prototype design, then, where relevant, a description of the Alpha Project's implementation.

3.1. The supervisor

Excluding the garbage collector, the supervisor is the only component of DSI that "knows" about suspensions. Every probe issued by an evaluator is validated, and if the result is manifest the process may proceed. Otherwise, the probing process is interrupted, its state is saved, its suspension is rescheduled, and some pending suspension is activated.

Figure 3.1-1 is a high level diagram of the Beta Project prototype. It is a multiprocessor with a monolithic list store, accessed through a bus. Components on the bus include an operator's console, some number of I/O devices and a set of list processing units (LPUs). The supervisory function is done by an eavesdropping component that monitors all transactions with the store. The eavesdropper intercepts probes of suspensions and schedules their activation. It applies available LPUs to pending processes.

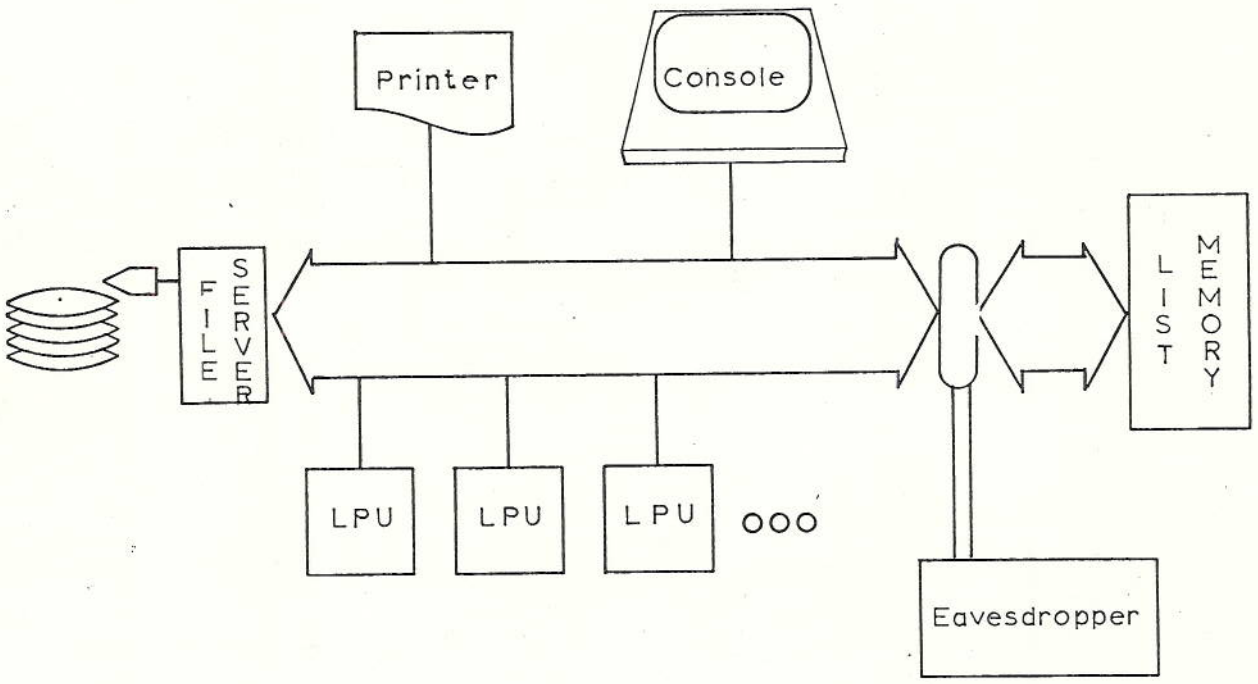
In DSI-alpha, a single processor, the host CPU, does both scheduling and computation. A reentrant routine called the supervisor maintains the schedule in a data structure. The physical representation of the schedule varies, depending on the global behavior being modeled. It now consists of a set of stacks, one for each device of the system (file, printer, keyboard, terminal screen), and a collection of queues, each representing an LPU.

3.2. Evaluators

The central level of DSI consists of a collection of LPU program segments, which perform the higher level functions of the virtual machine:

i) the interpretation of Daisy,
ii) scanning, parsing, and de-parsing Daisy programs,
iii) garbage collection,
iv) device handling.

Evaluators assume a manifest environment and rely on the supervisor to intercede when they attempt to interact.



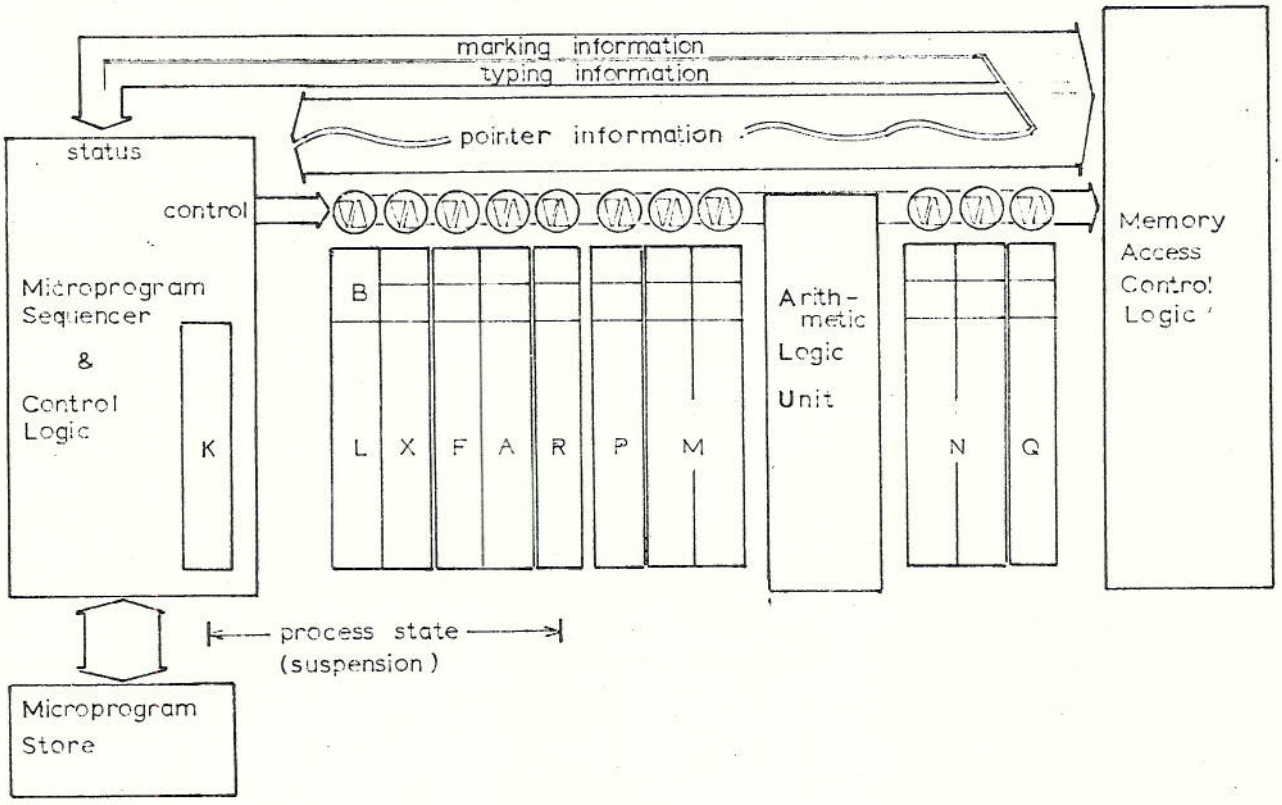
The Beta Prototype

Figure 3.1-1

Their "instruction set" is a collection of predefined transactions with the store. Whether it succeeds or fails, every transaction is designed to make an evaluator's state recoverable. Thus the supervisor may freely interrupt and deactivate an evaluator at any point in its computation.

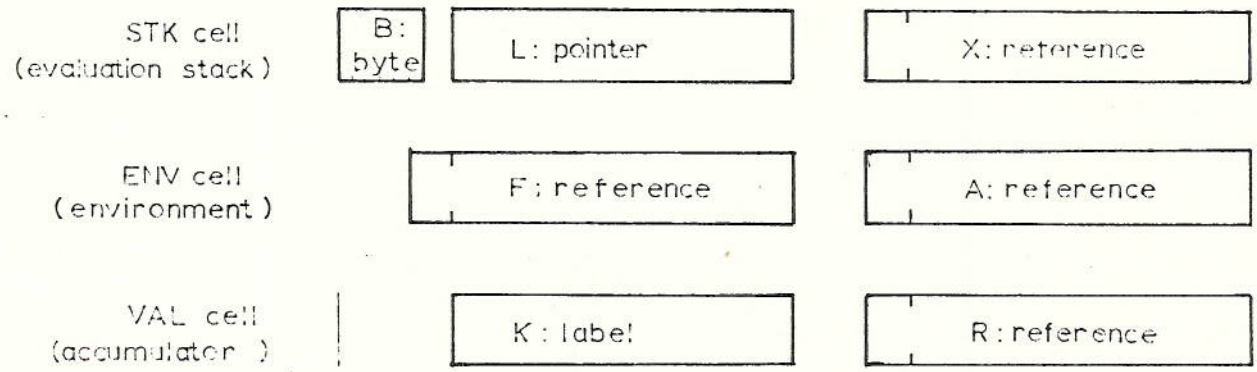
Figure 3.2-1 is a high level diagram of an LPU. In the Beta Project general purpose microprocessors emulate this architecture. Its internal configuration is a collection of registers organized on a bus and controlled by a microcode sequencer. Other components include an interface to the global store and an arithmetic/logical function unit. Of the twelve internal registers, six are for transient storage. The remaining six, including the microprogram index register, constitute the storable state of the processor and coincide with the record structure of a suspension.

Suspensions are held in three contiguous binary list cells. The first, called **STK**, is the topmost cell in a linked evaluation stack. A push on **STK** saves the contents of register B, a small integer, and the reference in register X. **ENV** holds a computational environment that usually consists of a formal part (register F), and an actual part (register A).[28] The **VAL** cell contains the suspension's label, **K**, and an accumulator, register R.



DSI List Processing Unit (LPU)

Figure 3.2-1



DSI's Suspension

A process, then, is an LPU "running" a suspension. Evaluators are written in an intermediate language called betacode that defines the allowable LPU register operations.[24] Each instance of an evaluation is, from its perspective, executing autonomously in a manifest environment.

Figure 3.2-2 shows the betacode for a portion of Daisy's interpreter in flowchart form. Evaluator control is based on register content (e.g. Branch Point C in Figure 3.2-2) or on the type of a referent cell (e.g. Branch Point E). Typical instructions include register transfer, probes, and stack operations.

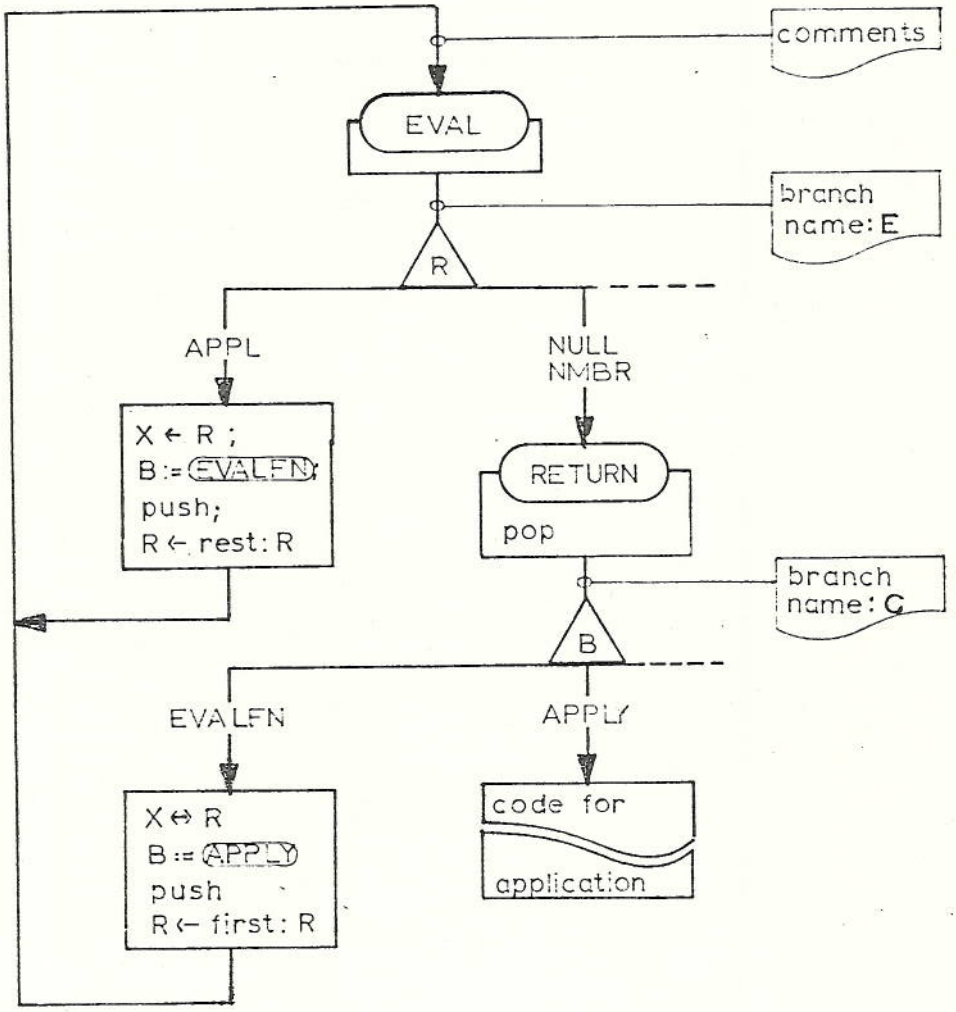


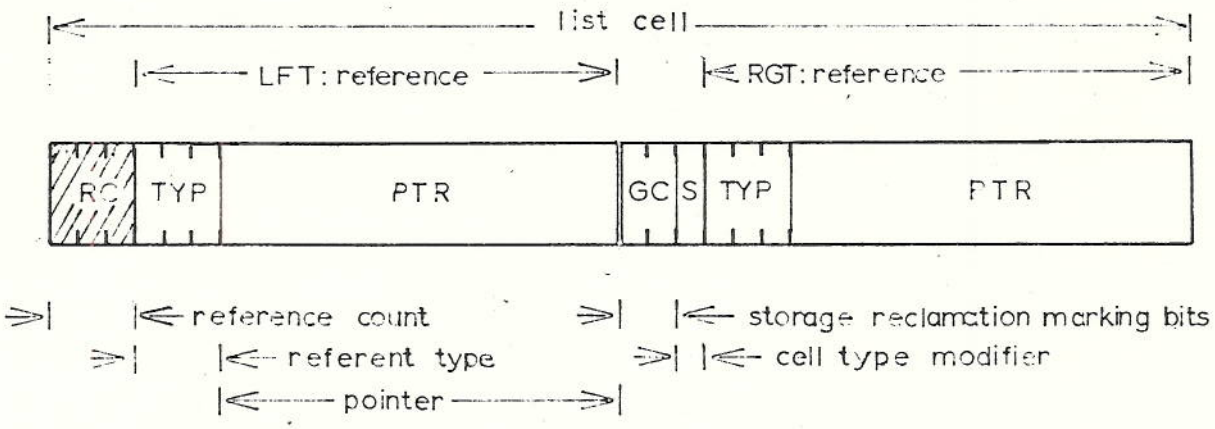
Figure 3.2-2

A Portion of Daisy's
Interpreter in betacode

3.3. Data Space Access

The lowest level of DSI is a collection of procedures, functions, and macros that implement betacode. The instruction set includes fetches, stores, probes, and constructors. It maintains a free space of binary cells and emulates the list-store component of the Beta prototype.

A typical list cell has two primary subfields, each a reference to another cell. The remainder of the record holds some bits of cell typing information and storage reclamation "marks". There is a reserved area for experimentation, especially with reference counts.[11,49]



A reference consists of a pointer together with some typing information. Although placing type indicators in the reference costs some space, it diminishes the number of store accesses required for interpretation.[45] "null", "identifier", "application", and "error" are example ground

types. These indicators are the primary means of control in the LPU, where they index microinstruction fetches.

3.4. Summary

DSI's program structure exhibits two kinds of information hiding. The data-space access routines implement the store as an abstract object, suppressing the implementation dependent details of storage management. The supervisor hides the interference among processing entities, allowing each evaluator to proceed autonomously in its computation. Process interaction is represented by a data structure called the schedule, which can be altered to explore different generic behaviors.

4. Further Research and Development

Clearly, DSI's behavior resembles that of an operating system. Its supervisor organizes a collection of "tasks" around some schedule. The distinction is in the fine grain of process decomposition. On a single-CPU architecture the size of tasks is kept large to reduce the overhead of multitasking. Ultimately, DSI-alpha will supplant its host's operating system. Its elementary operations subsume many of the functions of a general purpose host. While conventional CPUs have hardware that is well suited to the LPU function, it is often the case that privileged operations are needed to implement DSI's betacode efficiently. The host system gets in the way.

Since suspensions are ubiquitous in the data-space their management is a dominant factor in the cost of computation. Even with total control over the CPU, DSI-alpha could not compete with operating systems that maximize the size of their tasks. The payoff comes through multiprocessing, and DSI-beta is a step in that direction. Owing to the transparency of suspensions, we look for a graceful metamorphosis from multitasking to multiprocessing. Since there is no direct notion of process in the access language, the execution of a given program does not depend on processing resources. Moreover, interaction among processes is indirect, taking place through the medium of a manifest context. There is never a communicative "contract" between individual processors to perform subtasks and thus the failure of an individual LPU is not critical.

Further work can be classified by the Alpha/Beta distinction, although in most cases both projects are served by a single well conceived undertaking. This section summarizes those directions in research and development that can be based on the model of computation established in DSI.

4.1. Planning and Management

An orderly approach to advanced design requires a stable development environment. File organization and release procedures have already been established on DSI's current host, a Digital Equipment DEC/10 computer. Our department's newly acquired VAX 11/780 offers improved

management facilities and a transfer of DSI to the VAX is already underway. Once it is complete we can begin the process of incorporating the model at a lower level, beginning with the local optimization of primitive functions. The system will be in constant transition for the foreseeable future, as more of the underlying host mechanisms are subsumed.

Although the Beta prototype is still in the modeling phase, DSI's dual role continues even after design refinements make emulation impractical. Software aids are needed for the assembly, modification, and documentation of the prototype. Ancillary investigations, such as a study of process-memory connection schemes now underway[25,48] should be integrated. It is crucial that the development environment remains flexible enough to accommodate both projects.

4.2. Betacode Assembly

Presently, betacode is translated by hand into the implementation language. An assembly program would be an obvious improvement, but as yet there is neither a fixed source language nor a unique target language for assembly. A betacode assembler must produce target code for at least two machines: the DSI-alpha's VAX host and the Beta prototype's general purpose LPU emulator (not yet selected). The greatest danger in writing this assembler is that it tends to freeze the LPU design. While we cannot defer a precise LPU specification indefinitely, the absence of a

concrete specification allows us to explore alternative architectures without much additional cost. A behavioral model "cast in software" dampens this flexibility. However, once a consensus is reached about details of LPU architecture the situation changes. Standardization becomes necessary and conformity can be enforced through automatic assembly.

While any function of the programming system, an editor for example, could be compiled to betacode and introduced in DSI as an evaluator, we do not intend to make betacode a "source language" for the machine. Daisy should perform well enough for most applications. Thus we do not expect the assembler project to evolve into a compiler project, where Daisy "specifications" are reduced to betacode. We are more interested in program transformation techniques, where clear but inefficient applicative programs are put into a form that may be efficiently interpreted.[47,46]

4.3. Performance Measurement and Improvement

Suspended computation is based on phenomena so far removed from program text that it is tempting to seek general laws governing data-space behavior. Toward this end, work is underway to measure the observable properties of DSI[32] and to compare its spatial behavior with results obtained for a LISP based system.[3] One can make inferences about temporal behavior by looking at process objects, but to measure activity through simulated concurrency takes

enormous resources. The Beta prototype should create enough real concurrency to study its effects. Thus its design must incorporate measurement tools from the start.

With respect to Project Alpha, a number of performance improvements are straightforward. Suspensions fit into the register file of contemporary CPUs and could be activated quickly through a register transfer operation. Presently, the activation of suspensions is still a matter of software. Because it happens so often, the interruption of probes should also be at hardware level, probably through some form of forced memory access trap. We know of no high level programming language which allows us both to generate an exception and to capture the program state once this is done. By moving these primitives to assembly level, we expect a performance improvement of about one order of magnitude.

While substantial performance gains can be achieved through lower level coding, a more interesting challenge is the relaxation of DSI's fully demand-driven properties. (In other words, how can we safely make DSI less lazy?) Methods to introduce "partial strictness" range from compilation[39] to global processing heuristics.[6,40] Each addresses some aspect of expected program behavior and assumes some mechanism for efficient implementation. Thus the selection of a specific method has a profound influence on design. Here again, empirical study is needed to establish criteria for appraising methods.

4.4. Devices

The greatest single barrier to cohesive development in DSI is the behavior of conventional I/O devices. Input is almost universally viewed as the driving force of computation. Yet we regard computation as output-driven[10], and are constantly engaged in a battle to make the host architecture behave accordingly. Based on our experience, reversing the predilection for "input causality" requires major involvement with the host operating system. The problem may be eased by DSI's transfer to the VAX, since UNIX "pipes" have some of the properties we seek for file flow.

Ultimately a device is just a process that produces or consumes a stream. The overt behavior of a keyboard or a printer is indistinguishable from that of an LPU. The only difference is that these processes are dedicated to the peripheral devices they serve. Especially in the case of output, they place a "load" on the schedule by autonomously probing its environment.[10]

4.5. Applicative methods

While there are many claims about the benefits of applicative programming, there are few accounts of its practice "in the large". Efforts to improve DSI's performance on a conventional host reflect our desire to establish a laboratory for experimentation with larger applications. Project Alpha must address the breadth of the programming

experience and provide alternatives to imperative techniques. In some areas, error recovery for example, there is no established basis for an applicative approach.[38] In others, such as file organization, the solutions offered are tentative and seem costly.[7]

In the absence of a truly applicative programming vehicle it is difficult to test hypothetical approaches. There is always a side-effect that fixes the problem. We believe that good programmers will invent good methods if they are "constrained" to a purely functional environment.

5. Related Work

DSI is the beneficiary of numerous other investigations into programming language semantics, operating systems research, and multiprocessor design. This section discusses the more prominent influences. Omissions are inevitable; we are continually finding projects, even areas of study, that are applicable to our work.

The foundations of our work differs from others' in a couple of ways. First is the semantic distinction of call-by-value and call-by-name approaches. It seems appropriate to take one of these mechanisms as basic and the other as exceptional. Most choose call-by-value; we have chosen call-by-name. As the prefix "call-by" suggests, the basis for multiprocessing is often assumed to lie in the relationship of a function and its argument. Instead, we view

call-by-need as a symptom of a generic relationship between any process and its environment. It is not yet clear whether anything substantive is gained by this distinction, although it does allow us to concentrate on a computational model independent of language constructs.

Either approach can lead to a spectrum of control methods. At one extreme is an interrupt paradigm: a process is "awakened", or notified of the completion of a subtask, usually through the communication of a result. Alternatively, we adopt a polling paradigm, where the waiting process repeatedly tests its context for the emergence of manifest data. We are striving to minimize, or at least factor out, all assumptions about the communicative aspects of computation. By forcing interaction to occur through contexts, the outcome of a program never depends on the availability of an individual processing component.

5.1. SCHEME [42,44,43,45]

Sussman and Steele's implementation of the LISP dialect SCHEME has had a profound effect on DSI's design. Its influence continues through their recent work and through local SCHEME-based investigations. The most notable representational difference is the suspension's lack of a CLINK, or continuation pointer. In DSI this information is kept in the schedule, a result of the belief that control is not necessarily a matter of direct communication. For the same reason, the SCHEME-frame's accumulator is absent;

suspensions pass their results through their contexts.

5.2. The Model of Grit and Page [17,18,19,40]

Grit and Page have done extensive simulation on a model for general purpose suspension-based multiprocessing. Theirs is a large experimental effort born from and of direct consequence to our work. They have projected performance targets for an architecture similar to the Beta prototype, and have begun a study large scale communication architectures. They introduce (and inhibit) concurrency through function-argument interaction, and have explored the multiset construct. It is not clear to what extent, if any, our perception of the data-space differs from theirs, but it is likely that operational differences exist. Once these differences have been isolated their quantitative results should serve as guideposts to the Beta project design effort.

5.3. Actors [23,22,21,4]

Hewitt's work, and that of his colleagues, addresses a wide range of issues including syntactic constructs, process recovery schemes, and formal foundations for concurrency. So far, the effects on DSI have been indirect. The most straightforward application of the Actor model may be descriptive. For instance, DSI's supervisor is an object that manipulates continuations, and Hewitt's syntactic constructs have great facility in this regard. Clinger's

semantics for Actors[4] will likely direct our semantics for ferns.[9]

5.4. CCS [34,36,37]

Milne and Milner have developed a formal model of communication that manages to deal with state and still appear applicative. Theirs is the most attractive formal description technique we have seen. In defining process interaction as a coincidental message exchange rather than a directed communication, they have rendered the question of causality moot. But this does not solve the problem of cause and effect, and we do not presently know how to proceed gracefully from a formal description of behavior under their model to an implementable operational semantics.

5.5. Clark's empirical study of data-spaces [2,1,3]

Clark's measurements result from a comprehensive empirical study of LISP's data-space. A number of DSI design decisions, the hashing of small numbers for instance, were based on his observations. His approach gives a comparative basis for our own investigation and measurement of DSI's data-space behavior.

5.6. PARLAM [41]

Prini's applicative language PARLAM contains primitives for the scheduling and inspection of suspended objects. In our own terminology, he has elected to dissect the data-

space along different lines than we have, making exposure explicit and context implicit. If we regard PARLAM as a high level language, it reveals the very entities we set out to hide. However, PARLAM does isolate relational operators needed to build a schedule. This set of "dependency primitives" may in fact be minimal with regard to a language like Daisy.

6. Summary: DSI's Development History

In an article published in 1976 Friedman and Wise describe how a "suspending CONS" transforms a call-by-value LISP interpreter to call-by-need semantics.[8] The definitional interpreter they present was implemented in LISP, and used to demonstrate the effects of a lazy constructor.[50] While Wise's program, called SLISP, gave a succinct definition, it executed so slowly that even simple examples took minutes to run. Using SLISP as a specification, Cynthia Brown undertook a project to remove one level of interpretation by implementing an interpreter in Pascal.

In the spring of 1976 a seminar was held on the frame model of computation. It was there that the foundations of DSI's design began to take shape. The course was taught by Daniel Friedman, who, through considerable syntactic slight of hand, crafted a single-page definitional interpreter for SCHEME called CODA. CODA has evolved continually since then (it is now called ALONZO) and is still a guiding force in DSI's development.

Brown's implementation of SLISP was running by the summer of 1976, and when she accepted a faculty position, Steve Johnson took over the project. One of his first tasks was to find out why SLISP was still disappointingly slow.

Throughout the 1977 academic year, Friedman and Wise explored further implications of output-driven computation.[5,12,7,13,14,11] A research assistant, Tom Grismer, used SLISP to implement a number of systems level algorithms applicatively.[15] It was clear that CONS was too strong a function to handle problems in real-time programming. A weaker probing operator to do coaxing was rejected in favor of a weaker constructor, frons. The constructor frons was to deliver "call-time-choice" semantics [20] while retaining delayed evaluation. There are numerous ways to obtain this behavior, and the operational semantics of frons is still being discussed.[27]

In order to add frons to SLISP it was necessary to reconsider the role of suspensions in the system. They evolved from being a type and acquired the temporal aspects of state. This was such a fundamental change that it invalidated much of SLISP's design. Multisets were incorporated anyway, and SLISP continued to be used through 1980.[16] But the program proved too unwieldy a tool for deeper research into architecture, and too slow for use in the study of large algorithms. When Johnson left to work at Bell Laboratories, SLISP died from lack of interest.

A year later Anne Kohlstaedt and Casper Martin joined the project. They implemented a language they called Suspense in Pascal, using CODA as a specification. Suspense performed much better than SLISP on deterministic programs, due to a cleaner underlying design. But because it used Pascal's recursion to do scheduling, there was no easy way to capture state. Kohlstaedt and Martin experienced the same difficulties in adding multisets that Johnson did in SLISP.

In the fall of 1979, Johnson returned to replace Martin on the research team. Design was begun on DSI with several specific goals in mind. Multiprocessing would be incorporated at the outset, to avoid the inevitable complications of adding it to a deterministic system. We were determined to establish a development environment that would withstand changes in personnel, and be flexible enough to serve as a foundation for long term research and development. The frontier of suspended computation was to go beyond the language interpreter to include parsing, compiling, device handling, in brief, everything. DSI's primitive operations had to fit existing host architectures well enough to execute much faster than its predecessors. Finally, it was to provide a medium for quantitative study.

By mid-1980, the list processing subsystem, DSI, was running. Betacode began to evolve, and shortly after the 1980 academic year an interpreter for Daisy was implemented.

Over the next year a multiprocessing scheduler was developed, an interface to the file system was added, and about thirty benchmark programs were developed and measured.

At this time, further development has been postponed in order to transfer the system to the Computer Science Department's newly acquired VAX 11/780, operating under UNIX. DSI falls short of our performance goals by about two orders of magnitude. We believe a tenfold increase in speed can be gained by optimization of DSI at the assembly level. More improvement will come through refined scheduling techniques. But to meet and exceed our expectations, DSI's future depends on new architecture.

The "applicative premise" -- the hypothesis that applicative languages are more promising <u>in practice</u> than imperative languages -- is based partly on the belief that applicative programs can compete with their imperative counterparts. Support of this premise entails finding ways to make expressed algorithms conform to the physical constraints of the hardware that executes them. When hardware was the dominant cost in computing this meant allowing the program text to vary, through compilation. Now the situation has changed, software costs dominate, and we are looking for ways to let the hardware vary with the demands of a program. One approach is to remain in a general purpose setting and design extensible machines. This is the direction of the Beta Project. However, at this stage we are not out to

build a faster or a bigger computer, but a measurable one. We do so in order to find ways of appraising the overwhelming number of design decisions needed to build a machine that "runs applicatively".

References

1. Clark, D. W. and Green, C. C., "An empirical study of list structure in LISP," Comm. ACM, Vol. 20, (2) pp. 78-87 (February 1977).

2. Clark, D. W., List Structure: Measurements, Algorithms, and Encodings, Ph.D. dissertation, Carnegie-Mellon Univ., Pittsburgh, PA (August 1976).

3. Clark, D. W., "Measurement of dynamic list structure use in LISP," IEEE Trans. on Software Engineering, Vol. SE-5, (1) pp. 51-59 (Jan. 1979).

4. Clinger, W. D., Foundations of Actor Semantics, Ph. D. dissertation, MIT Mathematics Dept. (May 1981).

5. Friedman, D. P. and Wise, D. S., "Applicative multiprogramming," Technical Report No. 72, Indiana Univ. Computer Science Dept., Bloomington, Indiana (revised: April 1979).

6. Friedman, D. P. and Wise, D. S., "Aspects of applicative programming for parallel processing," IEEE Transactions on Computers, Vol. C-27, (4) pp. 289-296 (April 1978).

7. Friedman, D. P. and Wise, D. S., "Aspects of applicative programming for file systems," Proc. ACM Conf. on Language Design for Reliable Software, ACM SIGPLAN Notices, Vol. 12, pp. 41-55 (March 1977).

8. Friedman, D. P. and Wise, D. S., "CONS should not evaluate its arguments," pp. 257-284 in Automata, Languages and Programming, ed. S. Michaelson and R. Milner, Edinburgh University Press, Edingurgh (1976).

9. Friedman, D. P. and Wise, D. S., "Fancy ferns require little care," Technical Report No. 106, Indiana Univ. Computer Science Dept., Bloomington, Indiana (March 1981).

10. Friedman, D. P. and Wise, D. S., "Output driven interpretation of recursive programs, or writing creates and destroys data structures," Information Processing Letters, Vol. 5, (6) pp. 155-160 (December 1976). Erratum: Information Processing Letters, Vol. 9 (2) p.101 (August 1979).

11. Friedman, D. P. and Wise, D. S., "Reference Counting can manage the circular environments of mutual recursion," Information Processing Letters, Vol. 8, (1) pp. 41-44 (January 1979).

12. Friedman, D. P. and Wise, D. S., "An approach to fair applicative multiprogramming," pp. 203-226 in Semantics of Concurrent Computation, ed. G. Kahn, Springer-Verlag, New York (1979).

13. Friedman, D. P. and Wise, D. S., "An indeterminate constructor for applicative programming," pp. 243-250 in Seventh Annual Symposium on Principles of Programing Languages, (January 1980).

14. Friedman, D. P. and Wise, D. S., "A note on conditional expressions," Comm. ACM, Vol. 21, (1) pp. 931-933 (November 1978).

15. Grismer, T. M., Solving common programming problems with an applicative programming language, M.S. Thesis, Indiana Univ. Computer Science Dept., Bloomington, Indiana (1980).

16. Grit, D. M., Harwell, J. C., and Page, R. L., "An operating system in an applicative language," Technical Report, Colorado State Univ. (1979).

17. Grit, D. M. and Page, R. L., "Compiling LISP for a multiprocessor system," Technical Report, Colorado State Univ. (September 1979).

18. Grit, D. M. and Page, R. L., "Eager beaver evaluation on the R-ary N-cube," Technical Report, Colorado State Univ. (March 1981).

19. Grit, D. M. and Page, R. L., "Performance of a multiprocessor for applicative programs," Performance 80, (May 1980).

20. Hennesy, M. and Ashcroft, E. A., "Parameter passing mechanisms and nondeterminism," Procedings of the Ninth Annual ACM Symposium on Theory of Computing, pp. 306-311 (May 1977).

21. Hewitt, C. and Baker, H., "Actors and continuous functionals," IFIP Working Conference on Formal Description of Programming Concepts, pp. 16.1-16.21 (August 1977).

22. Hewitt, C. and Baker, H., "Laws for communicating parallel processes," pp. 987-992 in IFIP-77, , Toronto, Canada (August 1977).

23. Hewitt, C., "Viewing control structures as patterns of passing messages," Artifical Intelligence, Vol. 8, pp. 323-363 (1977). Also in Winston and Brown (eds.) Artificial Intelligence: an MIT Perspective, MIT Press, 1979.

24. Johnson, S. D., "Betacode specification," Technical Report, Indiana Univ. Computer Science Dept., Bloomington, Indiana. In progress

25. Johnson, S. D., "Connection Networks for Output-driven List Multiprocessing," Technical Report No. 114, Indiana Univ. Computer Science Dept., Bloomington, Indiana (October 1981).

26. Johnson, S. D., "Daisy programmer's reference card," Technical Report, Indiana Univ. Computer Science Dept., Bloomington, Indiana. In progress

27. Johnson, S. D. and Friedman, D. P., "A semaphore-free promotion strategy for frons," draft report, unpublished (March 13, 1980).

28. Johnson, S. D., "Variable association in Daisy," Technical Report, Indiana Univ. Computer Science Dept., Bloomington, Indiana. In progress

29. Johnson, S. D., An interpretive model for a language based on suspended construction, M.S. Thesis, Indiana Univ. Computer Science Dept., Bloomington, Indiana (1977).

30. Keller, R. M., Lindstrum, T., and Patil, S., "A loosely-coupled applicative multi-processing system," Proc. National Computer Conference, 1979, pp. 613-622 (1979).

31. Kohlstaedt, A. T., "Daisy 1.0 Reference Manual," Technical Report No. 119, Indiana Univ. Computer Science Dept., Bloomington, Indiana (November 1981).

32. Kohlstaedt, A. T., in progess.

33. McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P., and Levin, M. I., Lisp 1.5 Programmer's Manual, The MIT Press, Cambridge, Massachusetts (1973).

34. Milne, G. and Milner, R., "Concurrent processes and their syntax," Journ. ACM, Vol. 26, (2) pp. 209-222 (April 1977).

35. Milne, Robert and Strachey, Christopher, A Theory of Programming Language Semantics, Chapman and Hall, London (1976).

36. Milner, R., "On relating Synchrony and Asynchrony," Technical Report No. CSR-75-80, Univ. of Edinburgh, Edinburgh (1980).

37. Milner, R., "A calculus of communicating systems," Lecture Notes in Computer Science, Vol. 92, Springer-Verlag, (1980).

38. Morris, J. H., "Real programming in functional languages," Technical Report No. CSL-81-11, Xerox Palo Alto Research Center (July 1981).

39. Mycroft, Alan, "The theory and practice of transforming call-by-need into call-by-value," pp. 269-281 in Proceedings of the Fourth International Symposium on Programming, ed. B. Robinet, Springer-Verlag, New York (1980).

40. Page, R. L., Conant, M. G., and Grit, D. M., "If-then-else as a concurrency inhibiter in eager beaver

evaluation of recursive programs," Proc. 1981 ACM Conference on Functional Programming Languages and Computer Architecture, pp. 179-186 (October 1981).

41. Prini, G., "Explicit parrallelism in Lisp-like languages," Conf. Record of the 1980 LISP Conference, pp. 13-18 The Lisp Conference, (February 1980).

42. Steele, G. L. Jr. and Sussman, G. J., "Scheme: an interpreter for extended lambda calculus," Memo 349, MIT Artificial Intelligence Laboratory (December 1975).

43. Steele, G. L. Jr. and Sussman, G. J., "The dream of a lifetime: a lazy variable extent mechanism," Conf. Record of the 1980 LISP Conference, pp. 163-172 The Lisp Conference, (February 1980).

44. Steele, G. L. Jr. and Sussman, G. J., "The revised report on Scheme: a dialect of Lisp," Memo 452, MIT Artificial Intelligence Laboratory (January 1978).

45. Sussman, G. J. and Steele, G. L. Jr., "Design of a LISP-based microprocessor," Comm. ACM, Vol. 23, (11) pp. 628-645 (November 1980).

46. Wand, Mitchell, "Continuation based program transformation strategies," Journ. ACM, Vol. 27, (1) pp. 164-180 (January 1980).

47. Wand, Mitchell and Friedman, D. P., "Compiling lambda expressions using continuations and factorizations," Journ. of Computer Languages, Vol. 3, pp. 241-263 (1978).

48. Wise, D. S., "Compact layouts of banyan/FFT networks," pp. 186-195 in VLSI Systems and Computations, ed. H. T. Kung, B. Sproull, and G. Steele, Computer Science Press, Rockville, Maryland (October 1981).

49. Wise, D. S. and Friedman, D. P., "The one-bit reference count," BIT, Vol. 17, pp. 351-359 (1977).

50. Wise, D. S., "Interpreters for functional programming," pp. 186-195 in Functional Programming and its Application, ed. J. Darlington, P. Henderson and D. Turner, Cambridge University Press, Cambridge (1982).

INDEX OF TERMS

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES