December 1982 Report No. STAN-CS-82-951

Five Paradigm Shifts in Programming Language Design and Their Realization in Viron, a Dataflow Programming Environment

by

Vaughan Pratt

Department of Computer Science

Stanford University
Stanford, CA 94305

Five Paradigm Shifts in Programming Language Oesign and their Realization in Viron, a Oataf low Programming Environment

Vaughan Pratt
Stanford University

Abstract

We describe five paradigm shifts in programming language design, some old and some relatively new, namely Effect to Entity, Serial to Parallel. Partition Types to Predicate Types. Computable to Definable, and Syntactic Consistency to Semantic Consistency. Wc argue for the adoption of each. Wc exhibit a programming language, Viron, that capitalizes on these shifts.

'This research was supported by NSF grant number MCS82-05451.

1. Background

This is a companion paper to [Pra82]. These two papers started out as one paper, but when a clear division emerged between the language proper and its theoretical foundations it was decided to publish the foundations as a separate and self-contained thcorcticai paper, thereby removing much foundational clutter from the language paper that was not essential on a first reading.

It is possible to read and understand the present paper with only a limited understanding of the foundations. The foundations come into play when implementing the system, writing the manual, convincing oneself of the consistency of the ideas in the present paper, or establishing the soundness and/or completeness of proof systems for proving the correctness of programs in this language. The greater the accuracy with which each of these tasks arc to be performed. the more cioscly must one examine our foundations.

In related work on the language side, our most visible intciiectuai creditors are Hewitt (Actors) [I Iew], Hoare (CSP) [Hoa], and Dennis (VAL) [AI)]. On the foundational side, they arc Kahn and McQueen (Streams) [Kah,KM] and Brock and Ackerman (Scenarios) [BA]. Two notable contributors to the foundations of concurrency arc Petri [Pet] and Miincr [Mil], whose influential work we mention here only to iiiustratc that there is not yet one central generally accepted model of concurrent processes.

'i'his language is the programming language for the Viron computing env ironmcnt, a graphics-intensive user inlcrfacc being developed at Stanford by the author that takes as its starting point the philosophy of the Star environment [Lip] but that goes considerably beyond it in its attention both to semantic foundations and realistic graphics. Wc will call both the interface and its programming language Viron, relying on context to determine which is meant.

This paper deals only with the semantics of Viron. For presentation purposes we will adopt an abstract syntax in the spirit of McCarthy's Miisp. We believe however that such an abstract syntax is too spartan for a comfortable and efficient user interface. Other papers will describe syntaxes appropriate for given applications, including a three-dimensional graphical syntax wcii suited to the Viron interface, as well as a textual syntax that amounts to a formal approximation to English.

As to novelty, the paradigm shifts vary considerably in their individual novelty, with the effect-to-entity and serial-to-parallel paradigm shifts being the least novel. The emphasis of this paper is however not on the paradigms taken alone, but on their harmonious combination in a single programming language.

2. Sumrhary of the Paradigm Shifts

Wc give here a one-paragraph summary of each shift.

From Effect to Entity. I arge objects arc made as mobile as small, so that they can bc easily created, destroyed, and moved around as entities instead of being operated on pieccmcai as a static collection of small objects.

From Serial to Parallel. Constructs intended for parallel computation arc also used to subsume both serial and data constructs, achieving a simplification of the control structures of the language as well as providing a framework for organizing the data structures.

From Partition Types to Predicate Types. The partition view of types partitions the universe into blocks constituting the atomic types, so that every object has a definite type, namely the block containing it. The predicate view considers a type to be just another predicatc, with no a priori notion of *the* type of an object.

From Computable to Definable. Effectiveness is a trnditionai prerequisite for the admission of constructs to programming languages. Weakening this prerequisite to mere definability expands the language's expressivencss thereby improving communication between programmer and computer.

From Syntactic Consistency to Semantic Consistency. Consistency is traditionally enforced by syntactic restriction, e.g. via type structure. A recently dcveiopcd alternative popularized by Dana Scott takes the dual approach of resolving inconsistencies via semantic expansion. as done for models of the untyped λ calculus. Wc argue that the latter approach simplifies language structure.

We now expand on each of these paradigm **shifts.**

3. From Effect to Entity

Traditionally computation has been **viewed,** both by programmers and by mathematicians, as being performed for its effect. The classical basic instruction is the state-modifying assignment statement. The classical model of computation is an automaton (finite-state, pushdown, Turing machine, etc.) with a state or configuration transition function that describes how each state or configuration leads to its **successor.**

This view of computation is gradually being superseded by a more entity or object oriented view. The computing universe is regarded as being populated with entities. The dynamics of the universe is no longer described in terms of the state-to-state transitions of automata but rather in terms of the site-to-site transitions of entities, for which two of the more popular settings are recursive function evaluation and dataflow architecture.

There are several forces acting to bring about this paradigm shift. First, mathematics tends to emphasize entities over effects: consider the classical mathematical model, the algebraic structure, consisting of a set together with operations and relations.

Second, there is the increasing prevalence of parallelism, as nets connect computers into inherently parallel configurations, as cheap microprocessors start showing up several to a system, and as silicon compilers that translate algorithms into VLSI designs become a reality. The sequence-of-states model, well suited to serial computation, rapidly becomes unworkable in the presence of parallelism.

Third, there seems to be a psychological advantage to being able to externalize concepts, that is, to treat them all conceptually as nouns instead of verbs, relations, etc. We speculate that this advantage comes from the simplicity of the "typelessness" resulting from complete externalization. We will raise the issue of typelessness again in more detail in a later section.

4. From Serial to Parallel

4.1. Which is the Basic Concept?

One issue here is which is the more central concept, serial or parallel computation? There is an analogous question for determinism versus nondeterminism. For both questions there are ways of formalizing the question to make the answer come out either way. For the latter question the predominant view of automata these days, which is to represent them in terms of constraints on their state transitions, favors nondeterminism. with determinism being merely a special case of nondeterminism. Thus instead of dividing automata into two classes, the deterministic and the nondeterministic, we treat determinism as a property a nondeterministic automaton might or might not have, the linguistic awkwardness of the prefix "non" notwithstanding. This point of view has been firmly supported by the essentially universally accepted formal definition of "automaton" for the past two decades.

Our own intuition about serial versus parallel is that serial is a special case of parallel, rather than vice versa. However unlike the situation with determinism vs. nondeterminism, there has not been a similarly universally accepted formal definition of the notions "serial" and "parallel." This makes it much harder to resolve the question by appeal to a definition.

One can see how hard it is to relate the two notions formally by considering how they relate in current programming languages. Let us consider C (under Unix), Ada, and Hoare's CSP, which are among the better known languages offering parallelism. In each of these languages the basic computational paradigms are serial. Parallelism is introduced by modelling the computing universe as a set of serial computers communicating with each other. (In the case of C under Unix concurrency and inter-process communication, like I/O, is supplied by Unix kernel calls, and is not part of the C language proper. However it is a fine example of a language in current use that in practice does offer concurrency.)

These examples strongly suggest that the relation between the two notions is that serial computation is a necessary prerequisite to defining a notion of parallel computation.

This view of parallelism obviously does not reflect what actually happens inside a machine. Within any "serial" computer one can observe parallelism at many levels in its implementation: in the many electrons that flow through a wire (to go to an absurdly low level). in the many wires making up a bus, and in the many microprocessors that can be found in today's large mainframes, to name just some examples. For whatever reason the programming language of a mainframe is serial, it is not because the hardware itself is serial.

What we would like is a model of computation that not only reflected this ubiquity of parallelism but that at the same time subsumed the notion of serial computation, making it merely a special case of parallel computation. The chief advantage of this would be in simplifying both our theoretical models of computation and our programming languages. A program would then be serial more as an accident than through not using parallel constructs, just as determinism arises more by accident than by avoidance of nondeterministic constructs.

Hewitt [Hew] has advocated just this simplification of programming languages, in the form of his message-passing Actors theory. Although some of the details differ (actors have no output), the underlying rationale appears to be similar.

4.2. Need for a Formal Model

Comparing the situation once again with (non)determinism, there is still one missing ingredient, namely a formal semantics that converts the precedence of parallel over serial from a matter of taste to a mathematical definition. The candidates for a model of parallel computation that we take at all seriously are Petri nets [Pet], Milner's Calculus of Concurrent Systems (C'CS) [Mil], the Kahn-MacQueen model of determinate processes [KM], the Brock-Ackerman Scenarios model [BA], and our own model of processes [Pra82]. Among these models the greatest unity, and the longest history, can be found among [KM], [BA], and [Pra82], which together constitute a monotonically improving sequence of models (in that order). We consider the resulting model to supply exactly the missing ingredient.

The Kahn-MacQueen model defines an n-port process to be an n-ary relation on the set of all histories (sequences of data). This model was developed only for determinate processes, and was suspected by its authors of not being directly usable for nondeterminism. This suspicion was formally confirmed by an enlightening counterexample due to Brock and Ackerman, who also proposed the necessary modification to the model to extend it to nondeterminism [BA]. The modification was to introduce inter-history temporal precedence information. The Brock-Ackerman model was adopted and further extended by the present author [Pra82] to cater for process composition in a satisfactorily formal way, and to support an algebraic view of process composition analogous to the algebraic view of serial-program composition mandated by the structured-programming movement.

Besides the greater economy of subsuming the serial with the parallel, there is also the issue of irrelevant serialization forced in a serial language. Thus x:-a; y: = b is a pair of assignments

whose order must be given even though it is clearly not needed. This issue can bc met piecemeal by adding yet more constructs to the language, e.g. parallel assignment. However starting with an inherently parallel language from the beginning solves this and related problems just as effectively and more generally.

The Viron programming language is noteworthy in having no explicitly serial constructs.

4.3. Data Structures

Auxiliary to the vertical integration of processes into Viron's control structures is its incorporation into the data structures as well. The denotational semantics of processes given in [Pra82] imbues them with the status of object, permitting processes to be thought of as data with the same "charter of rights" [Pop] or "mobility" [Pra79] as integers.

Taking this development one step further, we have chosen to make processes not the organic molecules of our language but rather the elementary particles. That is, every datum, whether of the complexity normally associated with processes, or as simple as a character or an integer, is defined to be a process. In this we are again following Hewitt [Hew], in whose development "everything" is an actor.

The main conceptual obstacle to thinking of an atom as a process is that atoms seem too simple to be thought of in this way. However essentially the same argument was made for millennia excluding zero as a legitimatenumber. Yet today zero is almost universally acknowledged to bc, though less than 1, no less a number than 1.

Of course one might come up with an unconvincing behavior for numbers viewed as processes. Hewitt embeds the knowledge that 3 + 2 = 5 and 3x2 = 6 in the actor that is the number 3, which makes 3 a much more complex process than seems intuitively necessary. The Viron idea of a number, and more generally of any *atom*. as a process is that. although the atom does output something in response to each input, the output is independent of the value of the input and consists of the atom itself.

The main reason for this choice is to fit in with our extensional view of processes, in which two processes with the same behavior must be the same process, If an atom was unresponsive all atoms would collapse to the same atom. A useful fringe benefit of this convention is that, following our straightforward definition of addition, the addition of a number n to an array of numbers results in the addition of n to each of the elements of the array, as will be seen in the account below of Viron.

4.4. The Process Compiler

One might well ask why can't the notion of process be excluded from the programming language proper and made a part of the subroutine library, on the principle that the programming language only need supply a basis from which to extend via the subroutine library. All process-oriented notions in Unix arc supplied in this way, for example. (It should be realized that C was developed by the developers of Unix as part of the Unix effort: thus this expulsion of the notion of process to the library was a consciously made decision in this case, not an accident resulting from an inherited language.)

A plausible motivation for putting the notion in the language is that parallelism is not definable in purely serial terms, much as one might argue that nondeterminism is not definable using only deterministic concepts. f Iowevcr this argument assumes that the library is a true language extension in the sense that all its functions could have been written in the language. 'I'his is actually not the case in the Unix example. which requires non-C assembly code in its system calls in order to access the kernel, which is the source of parallelism in Unix. Thus it is possible to introduce parallelism into the language via the library even if parallelism is not definable using just the basic language, given that the library is permitted to step outside the basic language.

Our actual motivation is that we want to expose parallelism to the optimizing compiler. The state of the art of parallelism forces it to be an interpreted concept, due to its having a purely operational definition, one which admits only literal interpretation of parallel constructs by an interpretive machine. If a more abstract definition of parallelism is given, it becomes possible for an optimizing compiler to choose from a variety of equivalent implementations in compiling a given parallel construct. The dclinition should be maximally abstract: only necessary detail should be retained in the definition.

5. From Partition to Predicate Types

Another paradigm shift has to do with the nature of types. The partition view of types considers "type" to bc a function from the universe onto a partition of that univcrsc; each individual is mapped to the block of that partition containing that individual. Thus type(3) = integer, type(3.14) = real, type([3,1,4]) = list, typc(cos) = real->real, and so on.

In contrast to the partition view, the predicate view of types abandons the attempt to keep types disjoint, and permits each individual to bc of many types. For example 3 may simultaneously bc of type real, integer, positive integer, integer mod 4, mod 5, mod 6, etc. You yourself may simultaneously be a human, a teacher, an American, a Democrat, a Presbyterian, a non-smoker, and so on. There is no such thing in the physical world as TIIE type of an object, although any given context may suggest a particular predicate as being the most appropriate predicate to be called the type of that object in that contcx t.

The partition view can admittedly be made to work in the simple environments that come with today's programming languages. However as the environment gets richer the partition view becomes progressively more intractable. Imagine a programming language in which for every pair i,j of integers with i<j there is a type i..j of integers in the interval from i to j. A pure partition view of types would require that the integer 3 not be one individual but many, one for each interval containing 3. This may seem laughable, yet it is a logical extension of the more readily accepted idea that the real 3.0 is distinct from the integer 3. (It is noteworthy that Pascal adopts a predicate-like approach to its treatment of the range subtype, while remaining partition-oriented elsewhere, thereby avoiding this problem in its more extreme forms.)

The predicate approach to types simplifies this by having only one individual recognizable as 3, common to all intervals containing 3. This individual can even be identified with the individual 3.0 if one wishes to make the integers a subset of the reals, a simplifying view of the integer-real relationship which has much to recommend it.

The predicate view has a certain amount of support from modern mathematical logic. There has been much study of logical theories incorporating various notions of type. Indeed Russell's approach to controlling the logical paradoxes of Frege's theory was to introduce a type hierarchy of the partition kind. However this approach was eventually superseded by the typeless theories of Zermelo-Fraenkel and Bernays-Goedel. Admittedly the Wernays-Goedel theory did go so far as to postulate a two-type hierarchy of sets and classes, but it is noteworthy that the "typeless" (but not predicateless) Zermelo-Fraenkel theoty is the one that today is taken (modulo details) as the formal definition of set theory, which in turn is accepted by many mathematicians as supplying the formal basis for all of mathematics. While Zermelo-Fraenkel set theory may from time to time bc subjected to attacks, it is rarely if ever because of its typelessness.

The entity oriented approach that we wish to explore will be characterized by the "typelessness" of the predicate approach, in that all entities will belong to a single domain. Thus our approach will have the flavor of Lisp's typelessness, though with what we feel is a sounder rationale than has been advanced by the Lisp community to date for typelessness.

6. From Computable to Definable

It is unthinkable today to propose a noneffective model for a computing environment. How would you implement it? It is unimplementable by definition. Nevertheless we feel that this insistence on effectiveness produces inarticulate programmers. We propose to include noneffective concepts in our models to simultaneously enhance the expressive power of and simplify the language.

To begin with, consider the set of even integers and the set of primes. These are objects that are very natural to be able to refer to in a program: certainly in natural language they are referred to all the time. Having these objects in one's domain is only noneffective if one insists on a traditional representation of sets as bit vectors or linked lists of elements. If those two objects were all there were in the domain one bit would serve to represent each.

However suppose we close this tiny domain under Boolean operations. We now want to manipulate Boolean combinations of these two sets. Can this be done effectively? Yes: equality between expressions is decidable since it reduces trivially to the decision problem for two-variable propositional calculus, with the two sets playing the role of the two variables. Four bits suffice (exercise: and are necessary) to represent the sixteen possible Boolean combinations of these two sets.

Now let us go a little further and add a unary operation to the language that adds one to every element in a set. Suddenly we can express infinitely many distinct subsets of the integers, even without the evens. Nevertheless can we still compute in. this language? In particular can we always decide whether two expressions denote the same set? Maybe, maybe not (let us know if you find out), but clearly we cannot continue to add such "reasonable" constructs to the language for long without arriving at a non-effective domain, one in which not even equality is decidable.

R. Popplestone ran into this predicament when drawing up his "charter of rights" for data [Pop], where his notion of datum went beyond just integers and booleans. He wanted every datum, including objects such as arrays and functions, to be assignable to a variable, passable as a parameter, and returnable as the value of a procedure.

However he did not require that it be possible to tell whether two data were equal. More generally, he did not require that procedures behave the same with different representations of the same data. Why? Because equality is undecidable for functions, *inter alia*.

We consider Popplestone's charter of rights to be substandard. Under that charter data is not abstract. A programming language should assign abstractness higher priority than effectiveness. This is a logical extension of the programming rule, "Make it work before you make it fast." The extension is to treat the programming language as being primarily a descriptive tool, and only secondarily as a medium for achieving performance or even effectiveness.

Our approach to implementing a noneffective domain is to implement succinctly specified decidable language fragments. The key here is the existence of easily recognized decidable fragments of undecidable languages. We have developed this idea in [Pra80] and [Pra81] for the case of program verification. The idea is not specific to verification however, and can be applied just as readily to execution in a noneffective domain.

Those fragments may grow in size and number as the supply of algorithms improves; all that noneffectiveness does here is to prevent a complete implementation of Viron. The programmer should accept such incompleteness with the same good grace that the mathematician accepts it for his logical tools, which inevitably must be incomplete.

Making the break not only with performance but with effectiveness removes a source of worry from the programmer much as having an undo key reassures the user of a word processor. The programmer can get on with the job without the distraction of whether a given way of saying something will run fast, or even will run at all.

There is a feeling in some programming circles that the burden of performance should be placed on the compiler. This is possible up to a point, although no compiler can assume the full burden, since there are always new algorithms to be discovered. Our position is that exactly this situation holds for effectiveness as well as for performance. A compiler can deal with some of the issues of finding an effective way to execute a program, but no one compiler can discover every such effective way on its own, it must sometimes depend on the programmer. Just as the impossibility of the perfect optimizer does not imply the uselessness of optimizers, so does the impossibility of the perfect automatic programmer not imply the uselessness of compilers that can find effective methods in many cases.

7. From Syntactic to Semantic Consistency

Effectiveness is only one of the inhibitors of articulate expression. The current approaches to controlling inconsistency constitute another. Russell's theory of types was designed to' avoid the inconsistencies Russell and others found in Frege's logical theories. The introduction of a hierarchy of types into the X-calculus serves a similar end.

In contrast to these syntically cautious approaches are the syntactically casual languages of Schoenfmkel [Sch] (combinatory logic) and Church [Chu] (the untyped λ calculus). Here paradoxes of the traditional kind may be obtained at the drop of a hat: for example either language may express the seemingly nonsensical concept of a fixed point of the integer successor function. Yet the languages are more "user-friendly" than ones which introduce typing restrictions aimed at preventing such paradoxes. Are such languages merely syntactic curiosities devoid of referential significance, or can they be considered to actually denote, despite the inconsistencies? Surely they could not denote, or they would not be inconsistent.

Dana Scott has worked out the details of an approach to making semantic sense of paradoxical and hence ostensively meaningless languages, which is to computation as complex numbers are to electrical impedance. The idea is to augment an otherwise normal domain with fuzzy or information-lacking elements. Fuzziness is represented with a partial ordering of the domain in which x dominating y indicates that x has more information than y, which can be rephrased without using the word "information" by saying that y might on closer examination turn out to be x.

A very simple example of the shift from syntactic to semantic consistency is provided by Boolean circuits. A simple syntactic constraint on a circuit that guarantees predictable static behavior is that it be acyclic. This condition may be relaxed with caution to yield more interesting behaviors. However if in the interests of simplicity all conditions on circuits are dropped. we can then connect the output of an inverter (a device realizing the unary Boolean operation of complementation) to its input. This provides a simple physical model of the logical paradox implicit in the equation $x = \sim x$.

Classically a paradox means an inconsistency, which in turn means there is no model of the paradox - the universe should

disappear when we feed the inverter's output back to its input! This actually does happen, at least in the sense that the universe of pure truth values no longer provides an adequate account of the circuit behavior. With the feedback loop the inverter functions like an amplifier. with negative feedback, with its common input and output stabilizing at a voltage somewhere between logical 0 and 1. 'I'his intermediate voltage is not a part of the O-l Boolean universe, but it is a part of a more detailed model that admits invalid or uninformative data in addition to the regular data. Thus if we postulate three values, 0, *, and 1, with 0 and 1 considered maximally informative and * uninformative, and take the response of an inverter to the inputs 0, *, 1 to be respectively 1, *, 0, then we may solve x = ~x with x = *.

The key feature of this simple example is that we have moved from a syntactic to a semantic solution to the problem of paradox. Instead of relying on the absence of cycles or some other syntactic constraint to prevent paradoxes, Scott's approach is instead to expand the universe to account for and hence dispose of paradoxes.

Scott's approach was motivated by just the sort of "user-friendly" syntactic sloppiness that actually arises in real programming languages. such as the ability in Algol 60 to pass as a parameter to the function f any function including f itself. More recently Saul Kripke [Kri] has made a very similar proposal to the philosophical community with a paradox-explaining theory of truth that has been received with remarkable enthusiasm by the philosophical community. Kripke's theory of truth is founded on the existence of fixpoints of monotone functionals in a complete partial order, just as with Scott's theory.

It should be observed that the Scott-Strachey school of mathematical semantics that developed at Oxford has made two distinct contributions to programming semantics: the notion of denotational semantics as a homomorphism from expressions to values, and the notion of the information order as a basis for a fixpoint-of-monotone-functional semantics for resolving paradoxes. Yet little attempt is made by computer scientists to distinguish these two contributions, and the term "denotational semantics" is frequently applied to both of them as a single package, with the implication that the latter is a vital component of the former. In fact one can carry out a very comprehensive program of semantics without any reference to an information ordering. 'I'his is done for example in such program logic schools as algorithmic logic, dynamic logic, and temporal logic, where the semantics is of a homomorphic character but with no dependence on ordered domains. It is also done in [Pra82], the foundations on which the semantics of Viron are built.

When paradoxes emerge however in response to lax syntax, Scott's information order becomes a key ingredient of a successful semantics.

In the commonest account of Scott's theory (not Scott's own account however). based on complete partial orders (cpo's, partial orders in which every directed set has a sup), the maximal elements of the cpo can be considered the "normal" or "ideal" elements. the objects we consider to normally populate the universe. The other elements are approximations to the ideal elements, in the same sense as intervals with rational endpoints on the real line are approximations to reals. In the cpo account, unlike in Scott's account, there are no overspecified elements containing more information than the ideal elements.

The simple expressions of the language, e.g. the numerals, arithmetic expressions over numerals, etc., are considered to denote ideal elements. However some of the more complex expressions will only denote approximations. In particular the paradoxical expressions are guaranteed to denote approximations: no matter how closely you inspect a paradoxical element you cannot tell what ideal element it should denote. By withholding information in this way, the model prevents you from arriving at a contradiction. For example an expression denoting a fixed point of the successor function will denote an approximation to integers, usually one that approximates all integers (integer "bottom").

The main advantage of Scott's approach is the way it can simplify the language, which no longer needs to be sensitive to inconsistencies. On the other hand it does complicate the model. Yet even here there is an advantage, for the model can be used to permit the relocation of the implementability boundary from syntax to semantics, a novel concept for programming languages but one that we believe can be used to good effect. Let us see how this works.

Normally a system designer chooses an implementable language, and as new needs arise augments the language with additional implementable constructs. With Scottish models it is possible to fix an absurdly over-expressive yet simple language once and for all, and to augment not the language but the interpretation of the language, by increasing the information available to the language interpreter about the interpretations of expressions in the language. (Interpretation I, mapping expressions to domain elements, is considered an augmentation of interpretation J when I dominates J, i.e. I(e) dominates J(c) for all expressions e in the language.)

As a trivial example, one could start out with a semantic function that mapped numerals to integers, and all other expressions to the bottom element of the domain. Although the language might have addition, that function would in effect start out as the everywhere undefined function. Then one could add some set of computable arithmetic functions by raising from bottom to integers the Interpretations of all expressions containing only those functions and numerals, at the same time providing the necessary implementation of this increase. At some point one might raise the interpretations of "set of evens" and "set of primes" to the appropriate sets, also ideal elements. As algorithms for evaluating various linguistic fragments of set theory came to light one could implement them and so raise the interpretations of corresponding expressions. (If desired one might also add heuristics for noneffective fragments, thereby further raising some interpretations, though by ill-characterized amounts for an ill-characterized subset of the language.)

The advantage of putting "language subsetting" in the semantics instead of in the syntax is that it decouples language development from implementability considerations. This in turn makes it possible to make the full language available immediately for development of algorithms without waiting for full implementation support for those algorithms. These would sometimes be noneffective algorithms when they referred to as-yet undefined functions, but they still would serve the useful purpose of specifying problems that could then be rewritten manually in an effective sublanguage.

A language as powerful as this can be built up until it subsumes any given requirements language. From this point of view implementation reduces to translation within the language to achieve a raising of the interpretation (meaning) of the translated expression. The raising happens because, for example, some noneffective function or concept (e.g. quantification) is translated to a more effective form. The definition of correctness of an implementation is that it dominate the expression it was translated from (where the ordering between expressions is just that induced by the ordering on the interpretations of those expressions, i.e. for expressions e and f, e<f when I(e)<I(f)).

If one views an automatic programmer as a function mapping expressions to expressions in this language then the automatic programmer is correct just when it is monotonic.

This one-language view of the relation between requirements and implementation is appealingly simple. Yet it fits naturally into the real world of requirements and implementations, which typically form a hierarchy in which implementations turn into requirements as one programs from top to bottom. The homogeneity of our requirements and implementations simplifies this dual view of requirements/programs by expressing them all in a common language.

8. Lisp as a Benchmark

Lisp is a good benchmark against which to measure progress in language design. Despite its age (approaching the quarter century mark) it still ranks as one of the primary sources of insight into the principles of programming language design.

Lisp, at least pure Lisp, emphasizes entity over effect. Lisp treats its complex data, lists and (to an extent) functions, as objects to be moved around the computing environment with the same mobility as integers, putting demands on the storage management algorithms beyond what suffices for a domain of say integers. Furthermore Lisp emphasizes the homogeneity or typelessness of the predicate approach to typing.

However pure Lisp does not gracefully handle the process-oriented notions of state. memory, coroutine, or concurrency, concepts that are at best feebly captured in a domain of recursively defined functions and functionals on a basis of lists and atonts. It is usual to think of these as only recently being demanded, but we are of the opinion that their need has always been present, and that only the lack of the necessary concepts has prevented the Lisp designers and users from recognizing these needs as process-oriented needs long ago. We believe that the impurities of Lisp - PROG, SETQ, GOTO, RPLACA, RPLACD, etc. - arose in response to such needs, and met them by reverting from the entity paradigm to the effect paradigm, where it was already understood intuitively how to implement process oriented notions. The price for this step backwards was the loss of mathematical meaning for the concepts of Lisp, to the extent that being effect-oriented leads to clumsier definitions than being entity-oriented.

The similarity between pure Lisp and Viron is that both are entity oriented. The difference is that Viron entities are specifically intended to model the notions of state, memory, coroutines, and concurrency.

9. Foundations

As stated in the introduction, this is the second paper of a series whose first paper [Pra82] described the mathematical foundations for a notion of *process*. We repeat here the bare definitions.

There are two views *of* processes, *internal* and *external*. The internal view is the more detailed one, and depends on the notion of a *net* of processes, without regard for what actual data flows between them. All processes have two countable sets of *input* and *output ports*. $I_1, I_2, I_3, \ldots$ and $O_1, O_2, O_3, \ldots$, all but finitely many of which will normally go unused. (This arrangement avoids the encumbrance of a syntactic classification of processes according to their port structure.) The net consists of zero or more disjoint communication *links* each connecting one output port to one input port; each port is connected to at most one link. A net can be studied in its own right, or as a means of implementing a process. in which case certain of its processes are associated with ports of the implemented process.

In [Pra82] each port-associated process was assumed to use only one of its own ports. One minor improvement we make here is that model is to collect all the port-associated processes of a net into a single process. called the *exterior process of* the net. Port I_j of this process corresponds to port O_j of the implemented process! in the sense that data sent by the net to I_j will appear as output from port O_j of the process implemented by this net. Dually data arriving at port I_j of the implemented processes enters the net of the implementation of that process via port O_j of the net's exterior process.

To ask how the exterior process of net N is implemented is to ask what network the process implemented by N is embedded in. This viewpoint reflects a certain symmetry between the exterior and interior of processes that sharpens the role of the process as network interface.

A link is to be thought of not in the information theoretic sense of a channel having capacity, or affecting its messages, but rather merely as an arbitrary boundary between two processes. A datum flowing between two processes must at some time cross that boundary: this is called an net event. It either happens or does not happen: there is no probability, distortion, delay, or queuing associated with the event. Imperfections in the net must always be associated with processes. A transmission link that accumulates, permutes or distorts messages must be modelled as a process in our nets. The question of whether a link has a finite queue, an infinite queue, or no queue, is translated to the question as to what buffering mechanisms a process provides at each of its input and output ports. This in turn is captured abstractly in the "reliability" of a process - finite buffers will reveal themselves through the possibility of imperfect behavior.

Formally. a *net event* is a link-datum pair, interpreted as the traversing of that link by that datum. A *net trace* is a partially ordered multiset of net events, interpreted as a possible computation, with the order specifying which events necessarily preceded which other events in time. Necessary temporal precedence is a primitive notion in this theory. A *net behavior* is a set of net traces. These three notions, net event, net trace, and net behavior, constitute the internal view of a process.

In the external view, a *process event* is a port-datum pair, a *process trace* is a partially ordered multiset of process events, and a *process behavior* is a set of process traces. (The tight correspondence between the internal and external views of a process should be noted.)

There are two connections to be made between the internal and external views of a process. Network traces need to be consistent with the behavior of the constituent processes of the net, achieved by requiring that the restriction of each net trace to any constituent (i.e. non-exterior) process of the net be a process trace of that process. And the process behavior implemented by a net is obtained as the restriction of the net behavior to the exterior process, with I and O interchanged. In both cases "restriction" involves a renaming of links to ports, selection of the relevant events, and corresponding restriction of the partial order; details are in [Pra82].

We adopt an extensional view of processes, identifying them with their process behavior, just as one identifies a function with its graph (set of ordered pairs). Thus we may abbreviate "process behavior implemented by a net" to "process implemented by a net."

A network[1] of n processes numbered 1 through n defines an n-ary operation mapping each n-tuple of processes to the process implemented by that network having those n processes as constituents. The *net-definable* operations are those operations on processes definable in this way. A *net algebra* is any set of processes closed under the net-definable operations.

10. The Programming Language Viron

The goal of Viron is to be maximally useful with a minimum of machinery.

10.1. At the interface

In the word-object dichotomy, the concept of "language" seems to belong as much to the word as to the object it names. In this paper however we shall play down the syntactic part of Viron, leaving that to other papers. and focus instead on Viron's domain of discourse.

---

[1] There is a distinction made in [Pra82] between simple general nets. We have since decided to consider only simple nets.

In the interests of brevity and readability, and in keeping with the introductory nature of this paper, the description of Viron will remain at an informal level. A more rigorous treatment of the language would entail the use of a formal description language. It is our intent to use Viron to describe itself formally, just as an informal description of ZF set theory may be formalized in the language of ZF. (One reason for not using ZF instead of Viron is that they have quite different inconsistency-avoidance mechanisms. Viron evades inconsistency by being noncommittal, cautiously raising its definitions as far as its algorithms permit, whereas ZF sets itself up with fingers crossed as a fixed target that either is or is not consistent.)

The Viron universe is simply a set of processes, ranging in complexity from simple atoms through functional objects such as application and composition to large and/or complex systems. The Viron user interacts with processes: he manipulates them, watches them, talks to them, listens to them, and discourses on them (with an occasional break for coffee). No -one of these activities is intended to be the dominant one, nor is this list of what one can do with processes intended to be complete.

Abstract programming languages generally start out with one or another basic combining primitive. One popular such primitive is application; the domain of discourse of such a language is called a *combinatory algebra,* and the language itself is characterized as being *applicative.* All other combining operators, or *combinators,* are provided as elements of the combinatoty algebra. Church's h-calculus [Chu] provides a familiar example of a combinatory algebra; the set of proofs of propositional calculus, with modus ponens as the analogue of application, provides another.

The informal interface between Viron and its user takes the place of application in an applicative language. The precise definition of the processes themselves makes it possible to provide a formal definition of any given mode of user interaction on demand. Manipulation of processes may be formalized in terms of whatever combinators are supplied by the manipulation language - composition when processes can be assembled into a net, application when date can be input to processes, etc. Watching a process execute can be described formally in terms of viewing a trace. Talking to a process is the same as inputting data to a process, while listening to one is the converse - output from a process is sent to the user. Discourse on processes characterizes a user-Viron talk-listen loop since all transactions are themselves processes.

The fact that all data and computing agents are processes need not be pointed out to the beginner, who will encounter numbers, lists, functions, and so on well before the general notion of a process makes its appearance. However since this paper is for a more sophisticated audience we can afford to make the basic process representation explicit.

The least likely candidates for representation as processes are atomic data such as integers and characters. Somewhat more plausible are functions, which amount to memoryless processes. We have chosen to represent n-ary functions as processes that send one datum to output 1 when one datum has been consumed at each of the first n inputs, the output being the desired function of the consumed inputs.

10.2. Basic Data

Having ensured that functions are processes, to make an atom a process it suffices to make it a function, which we do by defining the atom b to be the constant function b satisfying b(x) = b for all x. (Type circularity is no problem here since we are not using a conventional type hierarchy of functions and functionals.)

Atoms: We take as the atoms of Viron the set Z of integers. (It is tempting to have other atoms such as characters, but the notion of "set of characters" is not sufficiently universal to justify its inclusion in Viron as a primitive.)

Arithmetic functions: the rational functions (addition, subtraction, multiplication, division) are provided. Division is a partial function in the sense that it absorbs its two arguments without response when the divisor is zero.

The arithmetic functions are defined not only on integers but on all functions. (Recall that an integer is an atom and hence a function.) The sum of two functions is coordinatewise addition (on the intersection of the domains of the two functions), and similarly for the other functions. It can be seen that this is consistent with the normal behavior of addition on integers; thus 2 + 3 = 5 either for the usual reason or because as functions the two constant functions whose values are respectively 2 and 3 sum to the constant function whose value is 5. Taking this further, it can be seen that the sum of a list and an integer is the result of adding that integer to the elements of the list, since the integer can be viewed as a constant function, with a domain that is a superset of the domain of the list (an initial segment of the positive integers). This is a happy circumstance, as it coincides with what programmers generally mean by the sum of a list and an integer, e.g. as in APL.

n-dimensional Array: a function with domain a contiguous rectangular subspace of Z^n.

List: a 1-dimensional array starting at 1. (This agrees with the definition of "list" on p.43 of MacLane and Birkhoff [Mac], and makes no attempt to relate lists to pointers. Making the pointer implementation of lists visible to the user, despite its obvious advantages in terms of control, makes the list concept unduly complicated.)

Filter: a restriction of the identity function to a partial function.

Set: a one-input two-output process whose two outputs in effect implement two filters with complementary domains, i.e. a steering mechanism. As such a set is not a function only in that it has two asynchronous outputs (as opposed to one output synchronously yielding a pair).

Predicate: a set.

Record: a function with domain a finite set of symbols.

Memory cell: a process that when sent any value on its second input outputs the most recent value seen on its first. (Cells are defined more formally in [Pra82].) It is a nontrivial example of a process that is not a function.

One may link n processes into a net with the help of various n-ary functions for that purpose. For example there is a binary function Sequence(a,b), which yields a process implemented by a net that connects its input 1 to a's input 1, a's output 1 to b's input 1, and b's output 1 to output 1 of Sequence(a,b). (This is made more formal using the definition of process composition in [Pra82].)

The quaternary function Fork(a,b,c,d) yields a process implemented by a net that connects its input 1 to a's input 1, a's output 1 to b's input 1, a's output 2 to c's input 1, b's output 1 to d's input 1, c's output 1 to d's input 2, and d's output 1 to output 1 of Fork(a,b,c,d).

The ternary function Loop(a,b,c) connects its input 1 to that of a, a's output 1 to b's input 1, b's output 1 to c's input 1, b's output 2 to output 1 of Loop(a,b,c), and c's output 1 to a's input 2.

The process Merge passes all data received on inputs 1 and 2 straight through to output 1, merging them subject to no particular rule.

The above process-combining operations form a useful, though surely incomplete, basis for parallel programming. However they can be seen to easily subsume the conventional serial constructs as well, if we consider "flow of control" in a serial machine to mean the flow of the entire state of the machine

as a single giant datum through a net. Thus "begin a; b end" may be written as Sequence(a,b), "if p then a else b" as Fork(p,a,b,Merge), and "while p do a" as Loop(Merge,p,a), where the predicate p is as defined above (a set, i.e. a pair of filters).

There is no explicit notion of type declaration in Viron. However one can always insert a filter into a data stream to achieve the effect of a declaration, which it does by blocking any object not of that type. If type error reporting is desired, this may be accomplished by using a set instead of a filter and routing the false output to a suitable error handler at run time. Compile time type error reporting amounts to testing at compile time whether it is possible for any errors to reach the error handler. (As usual with compile time computation, such a test may need to be conservative, sometimes predicting errors when none can happen, but never overlooking a possible error.)

Recursion is introduced into our model at the semantic level via the notion of least fixed point. (It is noteworthy that in our semantics the notion of minimality, whether of fixed points or anything else, is not used in the definition of Loop and hence of "while.") Operationally, this becomes the usual substitution of the process definition for the recursive use (invocation) of that process.

The notion of a passing a parameter to a function corresponds in Viron to the notion of inputting data to a process. In this sense only call by value is provided. Call by reference and call by name are avoided as being too unpredictable: it is difficult to prove a program correct when nearby programs hold pointers to objects of that program. Call by need should be treated as an implementation issue. The effects of these parameter-passing disciplines are best handled by passing objects of higher type by value.

11. Impact of the Paradigm Shifts on Viron

We now give a more detailed discussion of the impact of the paradigm shifts on the structure of the language. Much of the impact should already be apparent given the discussion of the paradigm shifts and the nature of the language. Thus this section is just a short Viron-specific supplement to the main discussion of the shifts.

11.1. From Effect to Entity

Viron is in one sense an applicative language. Every communication path is brought out into the open, instead of being hidden by references to shared variables. Applicative languages are normally inherently entity oriented. However in another sense Viron is effect-oriented, in that data entering a process can have an effect on that process. Yet the typical effect is to alter the set and/or arrangement of entities existing inside the process.

Thus Viron is at once entity oriented, like an applicative language. and effect oriented. like an imperative language.

By making every concept an object, and by having processes that can take processes as their input, we get the effect of a language of higher type. This provides a mathematically attractive way of getting expressive power that in other languages either cannot be attained or is strained for with a family of esoteric parameter-passing mechanisms.

11.2. From Partition to Predicate Types

The role of types in conventional programming languages is on the one hand to make clearer to the reader what the program does, and on the other to tell the compiler what data representation and type of operations to use in the translation. In Viron, filters, which arise naturally in Viron as simple process objects, are used to achieve both of these ends. This takes much of the mystery out of types, and at the same time provides a more flexible approach to types in that any Viron-definable predicate may be used as a type.

11.3. From Serial to Parallel

The net-definable operations provide all the needed control structures. This makes Viron an easy language to teach - once the notion of a net is in place, a variety of control structures, whether serial or parallel, can be introduced simply by exhibiting the appropriate net.

In Viron every datum, regardless of its complexity, is a process. Thus adoption of parallelism over serialism permits a uniform treatment of data, whether atomic, structured, or active.

11.4. From Effectiveness to Definability

One result of replacing effectiveness by definability is that it makes sense to think of Viron as a requirements language as well as a programming language. In this respect an implementation of Viron can be viewed as either a compiler/interpreter of Viron or an inference engine. The boundary between execution and inference is not a sharp one, and we feel is best characterized in terms of how much optimization is performed. Code motion, where an operation that the program shows as executing n times is actually only executed once thanks to the optimizer, is clearly an execution-related notion. Induction, where an operation that is shown as executing over all natural numbers is reduced to one step, is clearly part of inferencing. Yet the difference between these two "optimizations" is really only quantitative, if we accept infinity as a quantity. In between, we have in logic the notion of arguing by cases, which is indistinguishable from a program set up to deal with each of those cases.

Another consequence of deemphasizing effectiveness is that it changes the status of lazy evaluation. Normally lazy evaluation is thought of as part of the operational or interpretive semantics of the language, giving it the extra power needed to compute with infinite objects without going into an infinite loop trying to generate the object all at once. In Viron lazy evaluation disappears as a language concept, resurfacing if at all as an implementation concept. The Viron user is not meant to be aware (other than via performance) of whether lazy evaluation or some other method is used to deal with such infinite objects as the set of all primes. It should be possible to interchange such methods and have no effect on the semantics of any Viron program.

11.5. From Syntactic to Semantic Consistency

In Viron it is unnecessary to restrict how expressions may be built up and where data may be sent. One consequence of this is that a single least fixed point operator is possible in Viron, rather than a fixed point operator at each type as would be required in a more traditionally cautious language. Viron is to the untyped λ-calculus as a cautious language would be to the typed X-calculus.

12. Conclusion

Wc have proposed several changes to the way in which we view our programming languages, only some of which are presently advocated by others. These changes are not all obvious ones to make. Nevcrthcless wc believe that they are all changes for the better. We believe our arguments defending them to be sound. Thus the changes certainly should not be rejected without first disposing of our arguments.

13. Bibliography

[AD] Ackerman, W.B. and J.B. Dennis, A Value-Oriented Algorithmic Language, MIT LCS TR-218, June 13, 1979.

[BA] Brock, J.D. and W.B. Ackerman, Scenarios: A Model of Non-Determinate Computation. In Lecture Notes in Computer Science, 107: Formalization of Programming Concepts, J. Diaz and I. Ramos, Eds., Springer-Verlag, New York, 1981.

[Chu] Church, *A., The Calculi of Lambda-conversion.* Princeton University Press, 1941.

[I lew] Hewitt, C. and H.G. Baker, Laws for Communicating Parallel Processes, IFIP 77, 987-992, North-I Iolland, Amsterdam, 1977.

[Hoa] Hoare, C.A.R., Communicating Sequential Processes, CACM, 21. 8, 666-672, August, 1978,

[Kah] Kahn, G., The Semantics of a Simple Language for Parallel Programming, IFIP 74, North-Holland, Amsterdam, 1974.

[KM] Kahn, G. and D.B. MacQueen, Coroutines and Networks of Parallel Processes, IFIP 77, 993-998, North-I Iolland, Amsterdam, 1977.

[Kri] Kripke. S., Outline of a Theory of Truth, J. of Phil., 690-716, 1975.

[Lip] Star Graphics: An Object-Oricntcd Implementation, SIGGRAPH-82 Conference Proceedings, 115-124, ACM, July 1982.

[Mac] MacLane, S., and G. Birkhoff, *Algebra,* Macmillan, NY, 1967.

[Mil] Milner, R., *A Calculus of Communicating Systems,* Springer-Verlag Lecture Notes in Computer Science, 92, 1980.

[Mor] Morris, J.B., Types are Not Sets; 1st Annual ACM Symposium on Principles of Programming Languages, Boston, MA, October 1973.

[Pet] Petri, CA., Introduction to General Net Theory, Springer-Verlag Lecture Notes in Computer Science, 1981.

[Pop] Popplestone, R., The Design Philosophy of POP-II, Machine Intelligence 3, Edinburgh University Press, 1968.

[Pra79] Pratt. V.R.. A Mathematician's View of Lisp, Byte Magazine, August 1979, p. 162

[Pra80] Pratt, V.R.. On Specifying Verifiers, 7th Annual ACM Symposium on Principles of Programming Languages, Jan. 1980.

[Pra81] Pratt, V.R.. Program Logic Without Binding is Decidable, 8th Annual ACM Symposium on Principles of Programming Languages, Jan. 1981.

[Pra82] Pratt. V.R.. On the Composition of Processes, 9th Annual ACM Symposium on Principles of Programming Languages, Albuquerque. NM, Jan. 1982.

[Sch] Schoenfinkel, M, Ueber die Bausteine der Mathematischen Logik, Math. Ann. 92, 305-316, 1924. English translation in *From Frege to Goedel,* Harvard University Press, 1967.