A SHORT NOTE ON "PREDICATIVE PROGRAMMING": GLOBAL AND LOCAL CHAOS

Christian Lengauer

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

TR-84-06 February 1984

ABSTRACT

Recently formal semantics have been developed that identify Pascal-like programs with "implementable" predicates. This is a comment on the treatment of non- or abnormal termination in one such program semantics proposed by E.C.R. Hehner.

1. Introduction

Recently it has been proposed to identify imperative (i.e., Pascal-like) programs with "implementable" predicates (predicates that can be achieved by an automatic mechanism) and therefore embed formal program development in predicate calculus [2, 4]. In this approach, a specification is a predicate about a program's observable behaviour, and a program satisfying the specification is the specification itself, if it is implementable, and any stronger implementable predicate. This short note is a comment on one particular such semantics developed by Hehner [2]. We assume the reader is familiar with this semantics.

Hehner's specifications relate input values of program variables v (denoted $\grave{v}$) to their output values (denoted $\acute{v}$). Hehner's first paper (Part I) is solely concerned with sequential programming constructs. They are defined by predicates stating transformation rules of "initial" values of program variables into "final" values. The second paper (Part II) introduces programming constructs for concurrency: "processes" (communicating program parts) which communicate via "channels" (sequences of input and output values). A channel is a special kind of variable that represents a sequence of "intermediate" values during a computation.

There is a weakest possible specification of the state of a variable x; let us call it $\mathrm{K}(x)$, or "chaos at x". For ordinary variables it is the predicate *true* - nothing may be assumed about the state of the variable; for channel variables it is the assertion that past communications are never modified (see [2] for a formal definition). Hehner uses assertion K:

$$\mathrm{K} \quad =_{\mathrm{df}} \quad \forall x \epsilon v.\ \mathrm{K}(x)$$

where v is the vector of program variables.

The following two sections discuss two alternative treatments of chaos.

2. Global Chaos

"If chaos occurs at one variable it will spread to all others" is the philosophy of **global chaos**. Most of Hehner's semantics follows this philosophy. His definition of assignment is:

$$x{:=}e \quad =_{\mathrm{df}} \quad (\ \mathrm{D}\,\grave{e}\,' \Rightarrow \acute{v}{=}\grave{v}^{\grave{e}}_{\acute{e}}\ )$$

$$\wedge\ (\neg \mathrm{D}\,\grave{e}\,' \Rightarrow \mathrm{K})$$

If the assignment goes wrong, i.e., the initial value of expression e is undefined ($\neg \mathrm{D}\,\grave{e}\,'$), chaos results at **all** variables of vector v. Similarly, in the definition of composition:

$$P;Q \;=_{df}\; ((\neg\forall \acute{v}.\ P{=}K) \Rightarrow (\exists \dot{v}.\ P^{\acute{v}}_{\dot{v}} \wedge Q^{\grave{v}}_{\dot{v}})) \;\wedge\; K$$

chaos in P makes all of P;Q chaos.

Example: Assume two variables x and y.

$$P:\ x{:=}1/0 \;=\; \mathit{true} \qquad (\text{because } \neg D\,\acute{1}/0\,\acute{})$$

$$Q:\ y{:=}0 \;=\; \acute{x}{=}\grave{x} \;\wedge\; \acute{y}{=}0$$

Then $P;Q = Q;P = \mathit{true}$. Composing P and Q spells chaos for both x and y, because of a chaotic assignment of x.

As pointed out in [2], composition is in this semantics not associative. If P;Q spells chaos but neither P nor Q alone do, the associativity

$$(P;Q);R \;=\; P;(Q;R)$$

is violated. Hehner observed (not in [2]) that his particular view of chaos which equates non-termination and termination with an arbitrary result causes the problem. There are other global chaos semantics in which composition is associative. Hehner provides one in an appendix of Part I. It uses a "stop light" variable to separate the two issues of non-termination and termination with an arbitrary result.

3. Local Chaos

"If chaos occurs at one program variable, it will spread only to dependent variables" is the philosophy of **local chaos**. (Variable x depends on variable y if an assignment of x uses y.) An according assignment rule spoils only the target variable:

$$x{:=}e \;=_{df}\; (D\,\grave{e}\,\acute{} \Rightarrow \acute{x}{=}\grave{e})$$

$$\wedge\, (\neg D\,\grave{e}\,\acute{} \Rightarrow K(x))$$

$$\wedge\, \acute{v}_{-x}{=}\grave{v}_{-x}$$

where v_{-x} is vector v with variable x removed. Solely the definition of assignment governs the propagation of chaos. Composition is not concerned with chaos:

$$P;Q \;=_{df}\; \exists \dot{v}.\ P^{\acute{v}}_{\dot{v}} \wedge Q^{\grave{v}}_{\dot{v}}$$

Thus, composition is symmetric and associative.

Example: With programs P and Q of the previous section:

$$P \;=\; \acute{y}{=}\grave{y}$$

$$Q \;=\; \acute{x}{=}\grave{x} \wedge \acute{y}{=}0$$

Then P;Q = Q;P = $\acute{y}$=0. Now, y is spared from chaos since y does not depend on x.

One of Hehner's connectives, independent composition, follows the philosophy of local chaos:

$$P||Q \quad =_{df} \quad P(v_P) \wedge Q(v_Q)$$

Chaos in P does not spread to Q and vice versa. Two programs P and Q may only be composed by || if they are independent. P and Q are independent if they manipulate distinct portions v_P and v_Q of vector v.

4. Mixing Chaos Philosophies

In Hehner's semantics, independent composition is special in several ways:

(a) It is the only connective that does not apply to all, only to independent predicates.

(b) It is the only connective that may introduce unbounded non-determinism.

(c) It is the only connective that does not follow the philosophy of global chaos.

Hehner criticizes (a) and (b); we shall criticize (c):

For our previous programs P and Q, P||Q = $\acute{y}$=0 while P;Q = Q;P = *true* . Ordinarily, the concurrent composition of P and Q (by connective ||) permits or, in Hehner's terminology, "is less determined than" the arbitrary interleaved composition of P and Q (by connectives ;), but not in this semantics. We would like independent composition to be implied by arbitrary interleaved composition. In particular, independent composition should permit ordinary composition. The only justification for connective || is then to simplify the detection of concurrency by making independence explicit.

There are two alternative ways to accomplish this:

(1) Let independent composition follow the philosophy of global chaos, e.g.,

$$P||Q \quad =_{df} \quad ((\neg\forall\acute{v}.\ P{=}K) \wedge (\neg\forall\acute{v}.\ Q{=}K)) \Rightarrow (P(v_P) \wedge Q(v_Q)) \wedge K$$

With this definition, independent composition (P||Q) permits ordinary composition (P;Q and Q;P). To make independent composition permit arbitrary interleaving, we have to worry about chaos down to the level of atomic actions. (Compare Theorem 15 of [2].)

(2) Keep the definition of independent composition and adopt the local chaos semantics described in Sect. 3. Since channel input and output can be expressed as sequences of assignments:

$$\text{input:} \quad c?x \quad = \quad x{:=}c[0];\ c{:=}c[1...]$$

output: $d!e \;=\; d{:=}d^\frown e$

they inherit local chaos semantics from assignment and composition. Essentially, in case of missing input or output, chaos is restricted to the channel in question: c or d, respectively. The semantics of input choice:

$$[a?x \to \mathrm{P} \;[]\; c?y \to \mathrm{Q}]$$

requires a similar change. If input is missing on both channels a and c, chaos occurs at a and c; the states of all other variables are preserved. We leave it to the reader to spell out the according semantic formulas.

Observe that both (1) and (2) make the ordinary composition of independent programs commutative, since independent composition is a special case of ordinary composition and is commutative. This holds, in particular, for independent channel communications. In contrast to [2]:

$$c?x;\; d!1 \;=\; d!1;\; c?x$$

5. Comparison

We have discussed three different semantics:

(a) Global chaos

Global chaos is the "centralized" and weakest of the three semantics. Here, a computation may either succeed completely or fail completely. An occurrence of chaos in just one of many independent components may cause abortion of the entire program. A concurrent computation terminates when all its concurrent components have terminated. Concurrency is identified by independent composition and is implemented by traditional techniques, e.g., may be simulated with arbitrary interleaving.

(b) Local chaos

Local chaos is the "distributed" and strongest of the three semantics. It takes the notion of partial success to the extreme. Chaos only affects dependent parts of a concurrent or sequential computation. Only those parts may abort; independent parts must continue. This requires a data flow analysis. (To make it managable, one might like to discipline the re-assignment of variables [1].) Independent composition becomes obsolete. Its justification was to identify concurrency. Here, concurrency is a by-product of the data flow analysis and may be implemented by lazy evaluation [1]. The rest of the semantics inherits one of the properties of independent composition: the potential for unbounded non-determinism.

(c) Hehner

Hehner's semantics is partly centralized and partly distributed. Program parts (processes) which are themselves composed by centralized semantics

(ordinary composition) are then composed by distributed semantics (independent composition). These two forms of composition are not compatible: independent composition may not be replaced by ordinary composition. Independent composition identifies concurrency, but its implementation still requires data flow analysis: termination of P||Q does not imply termination of both P and Q.

We suggest that the global chaos semantics is appropriate for explicit concurrency, the local chaos semantics for implicit concurrency. The implementation of explicit concurrency should be simpler than that of implicit concurrency. Hehner has a connective for explicit concurrency, but requires implementation techniques for implicit concurrency.

One implication is that with Hehner's interpretation of chaos, where nontermination cannot be distinguished from termination with an arbitrary result, distributed systems need a data flow semantics. Other people also have described distributed systems with data flow semantics. Van Emden [5] uses logic programming: processes are predicates, communication is provided by variable instantiation. Henderson [3] formulates a network of distributed processes as a functional program: processes are functions, communication is provided by parameter passing and value return. Both approaches are referentially transparent (i.e., without side-effects), simplifying data flow analysis. Both approaches implement concurrency by lazy evaluation. Ackerman [1] discusses the representation of communication channels (or "streams") in functional programs.

6. Acknowledgement

This note emerged from discussions with Rick Hehner.

7. References

[1] Ackerman, W.B. Data flow languages. **Computer 15**, 2 (Feb. 1982), 15-25.

[2] Hehner, E.C.R. Predicative programming (Part I and II). **Commun. ACM 27**, 2 (Feb. 1984), 134-151.

[3] Henderson, P. **Functional Programming: Application and Implementation.** Prentice-Hall International, London, 1980, Sect. 8.4.

[4] Hoare, C.A.R. Specifications, programs, and implementations. Technical Monograph PRG-29, Oxford University Computing Laboratory (Computing Research Group), June 1982.

[5] van Emden, M.H., de Lucena Filho, G.J. Predicate logic as a language for parallel programming. In **Logic Programming.** K.L. Clark & S.-A.Tarnlund (Eds.) , APIC Studies in Data Processing No. 16, Academic Press, New York, 1982, p. 189-198.