ORDER-THEORETIC TECHNIQUES FOR NONDETERMINISTIC PROGRAMS*

Michael G. Main

CU-CS-344-86 August 1986

Presented at the Second Workshop on Programming Language Semantics, Manhattan, Kansas

*This research has been supported in part by National Science Foundation grant DCR-8402341.

ORDER-THEORETIC TECHNIQUES FOR NONDETERMINISTIC PROGRAMS*

Michael G. Main
Department of Computer Science
University of Colorado
Boulder, CO 80309 USA
Phone: 303-492-7579

ABSTRACT

This paper is an exposition of the demonic semantics of nondeterministic programs, as recently used in several technical works (*e.g.* [4,17,20]). Particular emphasis is placed on demonstrating the use of the usual order-theoretic proof techniques, such as computational induction. The main example developed is the semantics of nondeterministic recursive programs, but the same techniques apply to nondeterministic recursive program schemes or nondeterministic iterative programs.

*This research has been supported in part by National Science Foundation grant DCR-8402341.

1. MOTIVATION

There are two special cases of the order-theoretic semantics which have been considerably studied on their own:

- *Partial function theory:* Deterministic programs are interpreted as partial functions from an input domain to an output domain. A function is undefined at points where its program fails to terminate.
- *Angelic relational semantics:* Nondeterministic programs are interpreted as binary relations between input and output domains (for example, [7, part I]). Each input element is related to all the possible output elements that it can yield. The possibility of nontermination is ignored. (We hope that an angel will lead the computation away from nonterminating paths.)

Both of these models are useful because they can make use of the techniques of order-theoretic semantics (such as computational induction, fixpoint induction, ...) without employing the full scenario of domain theory. However, neither model is appropriate for total correctness proofs of nondeterministic programs: in the first case, nondeterminism is not allowed, while in the second case nontermination is ignored.

One way to correct this deficiency is to include a new "state" denoted ω. If a computation has the possibility of nontermination on an input x, then x is related to ω by the computation's relation. This is the approach taken by deBakker [3] or Harel & Pratt [8], for example. In both of these cases, *ad hoc* rules are added to indicate when an input is to be related to ω. As a result, the usual order-theoretic proof techniques cannot be immediately applied.

An alternative approach, within order-theoretic semantics, is suggested by Smyth's powerdomain [18] and Plotkin's interpretation of this construction [16,17]. In it's most general setting, Smyth's powerdomain provides semantics for nondeterministic computations over arbitrary CPOs (complete partial orders). The purpose of this paper is to point out the special case of Smyth's powerdomain in relational semantics. The result is a simple relational semantics in which total correctness proofs of nondeterministic programs can be carried out by the usual order-theoretic techniques.

In terms of relational semantics, Smyth's construction provides this principle: each input is related to any output which cannot be ruled out by some *finitely-deep* amount of information about the computation. In practice this means the following:

- If a program has the possibility of nontermination on input x, then x is related to every possible output (including ω). The reason is that any finitely-deep information about the computation paths of the program must leave an unfinished path — hence no possible outputs can be eliminated.
- If a program must always terminate on input x, then x is related to exactly those outputs which are possible via terminating paths. This is the same as the angelic relational semantics.

Notice how this principle makes these two programs equivalent:

PROGRAM 1:

```
i := 0;
while (i = 0) do
    i := 1    or    i := 0
od
```

PROGRAM 2:

```
i := 0;
while (i = 0) do
    i := 0
od
```

The underlying relation for both programs relates every possible input to every possible output — which merely indicates that termination can never be guaranteed. The fact that Program 1 can sometimes terminate is irrelevant here. This has been called *demonic semantics* [4], meaning that if nontermination is possible, then a demon in the system will cause the nonterminating path to be taken. Of course, we don't really expect such a demon to be implemented — but the concept is useful when reasoning about what a program *must* do under all possible nondeterministic executions. (See [1,17,20] for more on this approach).

The reward for taking this approach is two-fold: First, the demonic relational semantics is easy to specify. Iteration and recursion employ least fixpoints with respect to the superset ordering on relations. Second, since we are in the realm of order-theoretic semantics, the usual proof techniques such as computational induction and fixpoint induction can be employed directly. The remainder of this paper is an expository presentation of these ideas for recursively defined nondeterministic programs (similar to Manna [12], chapter 5).

Notation: The equality operator will generally be denoted by $\equiv$ instead of $=$, (since we reserve the usual equal sign for a weak equality). If $f \subseteq A \times B$ is a binary relation, and $a \in A$, then we use the notation $f(a)$ to denote the set $\{b \in B \mid (a,b) \in f\}$. Similarly for a subset $S \subseteq A$, we use the notation $f(S)$ for $\bigcup_{a \in S} f(a)$. Some care must be taken in relational application. For example, consider the relation $double \subseteq \mathbf{Z} \times \mathbf{Z}$, defined by $double(x) = \{x+x\}$ (where $\mathbf{Z}$ is the set of integers). Here is an *incorrect* derivation:

$$double(\{1,\ 2\}) = \{1,\ 2\}+\{1,\ 2\} = \{1+1,\ 1+2,\ 2+1,\ 2+2\} = \{2,\ 3,\ 4\}$$

The incorrect step is the first step, which applies *double* to its argument $\{1,2\}$. Such a "beta-conversion" can only be carried out if the argument is a singleton. The correct derivation is:

$$double(\{1,\ 2\}) = \{double(1),\ double(2)\} = \{1+1,\ 2+2\} = \{2,\ 4\}$$

This method of application is the "call-time choice" parameter passing rule [9,10].

2. NONDETERMINISTIC RECURSIVE PROGRAMS

Throughout the paper, D is a fixed arbitrary set of values, which includes three special elements *true, false* and ω (called "undefined").

2.1 Nondeterministic programs

We define a nondeterministic n-ary program to be a relation $f \subseteq D^n \times D$, such that

(1) Let $\bar{x} \in D^n$ be a vector where none of the components are ω. Then $f(\bar{x})$ is non-empty, finite and does not contain ω, or $f(\bar{x}) \equiv D$.

(2) Whenever $f(x_1, \cdots, D, \cdots, x_n)$ is not D, then for any $y \in D$:

$$f(x_1, \cdots, y, \cdots, x_n) \equiv f(x_1, \cdots, D, \cdots, x_n).$$

(The y appears at the same position as D in this condition.)

The collection of all such programs is denoted $[D^n \times D]$. The first restriction on $f(\bar{x})$ corresponds to the notion that if $f(\bar{x})$ has an infinite number of nondeterministic possibilities, then it must also have a non-terminating path. Hence (by the principle of demonic semantics), $f(\bar{x}) \equiv D$. (See Dijkstra [6] for more on why infinite nondeterminism causes a nonterminating path.) The second restriction corresponds to the idea that if an argument is D, then either the argument is not used, or the result contains ω (which again results in all of D as output).

We will use this collection together with the usual subset and superset orderings on relations. Notice that every descending chain $f_0 \supseteq f_1 \supseteq f_2 \cdots$ has a greatest lower bound $\bigcap_{i=0}^{\infty} f_i$. The collection also has a greatest element — the complete relation which we ambiguously denote Ω (for any n). In demonic relational semantics, Ω is the program which always has the possibility of nontermination. Also, the collection of programs is closed under the usual relational composition. Here are some example programs:

(1) The program $equal \subseteq D^2 \times D$, where $equal(x,y)$ is

```
if (x is ω) or (y is ω) then D
else if x is identical to y then {true}
else {false}.
```

This will usually be abbreviated as $x{=}y$.

(2) The program $cond \subseteq D^3 \times D$, where $cond(p,x,y)$ is

```
if p is true then {x}
else if p is false then {y}
else D.
```

This will usually be abbreviated as *if p then x else y*.

(3) Let $S \subseteq D - \{\omega\}$. Any function $f : S^n \rightarrow D$ can be extended to a program $f \subseteq D^n \times D$, which relates each $\bar{x} \in S^n$ to only $f(\bar{x})$, and relates every other element of D^n to all of D. This is called the demonic extension of f to a relation.

(4) The program $union \subseteq D^2 \times D$, where $\bigcup(x,y)$ is $\{x,y\}$ (if neither x nor y is ω) or D (if $x \equiv \omega$ or $y \equiv \omega$). This is a nondeterministic choice between the two arguments and will usually be abbreviated as x *OR* y.

(5) Here are some properties of the *cond* and *union* programs:

When $fD \equiv D$ then $f(\textit{if } b \textit{ then } x \textit{ else } y) \equiv \textit{if } b \textit{ then } f(x) \textit{ else } f(y)$.
$(\textit{if } p \textit{ then } (x \textit{ OR } z) \textit{ else } (y \textit{ OR } z)) \equiv (\textit{if } p \textit{ then } x \textit{ else } y) \textit{ OR } z$.
$(\textit{if } (p \textit{ OR } q) \textit{ then } x \textit{ else } y) \equiv (\textit{if } p \textit{ then } x \textit{ else } y) \textit{ OR } (\textit{if } q \textit{ then } x \textit{ else } y)$.
$(\textit{if } p \textit{ then } (x \textit{ OR } z) \textit{ else } y) \equiv (\textit{if } p \textit{ then } x \textit{ else } y) \textit{ OR } (\textit{if } p \textit{ then } z \textit{ else } y)$.
$(\textit{if } p \textit{ then } x \textit{ else } (y \textit{ OR } z)) \equiv (\textit{if } p \textit{ then } x \textit{ else } y) \textit{ OR } (\textit{if } p \textit{ then } x \textit{ else } z)$.

2.2 Correctness

To further illustrate the ideas of demonic semantics, this section gives a correctness proof for a simple nondeterministic program. In the demonic setting, the key idea of a correctness proof is that every nondeterministic possibility must be acceptable.

So, suppose $D = \{\omega, true, false, 0, 1, 2, \cdots\}$, and we wish to define a program $f \in [D^2 \times D]$ which gives the quotient of dividing two numbers. We do not care what the value of f is when either of the arguments is not a number, or when the second argument is zero. We state these requirements as a pair of conditions:

- Precondition (A predicate on D^2 which indicates legal inputs):

$$P(x,y) = x \in \{0,1,2,...\} \text{ and } y \in \{1,2,3, \cdots\}.$$

- Postcondition (A predicate on D^3 which indicates what outputs are legal for given inputs):

$$Q(x,y,z) = (z = x \text{ DIV } y).$$

(The value of the postcondition is irrelevant if $P(x,y)$ is false.)

Given these conditions, a program f is correct provided that for all $x,y \in D^2$, and for all $z \in f(x,y)$:

$$P(x,y) \text{ implies } Q(x,y,z).$$

Of course, there are many programs which are correct. The greatest such program is

$$f(x,y) = cond(P(x,y), (x \text{ DIV } y), D),$$

but other programs such as $cond(P(x,y), (x \text{ DIV } y), 0)$ are also correct. In fact any program g such that $f \supseteq g$ is correct.

This is a non-recursive example, so it is not too interesting, but it does illustrate the meaning of a correct program in the demonic semantics, and the importance of the $\supseteq$ order. For recursive programs, the same notion of correctness applies, but the reasoning must involve fixed-point techniques. These techniques are shown in the remainder of the paper.

2.3 Continuous Relationals

A *relational* is a function $\tau{:}[D^n \times D] \rightarrow [D^n \times D]$. It is *monotonic* provided that it preserves the $\supseteq$ ordering (*i.e.,* $f \supseteq g$ implies $\tau[f] \supseteq \tau[g]$). It is *continuous* provided that it is monotonic and preserves the intersection of any decreasing chain (*i.e.,* if $f_0 \supseteq f_1 \supseteq f_2 \supseteq \cdots$ is a chain, then $\tau[\bigcap_{i=0}^{\infty} f_i]$ is $\bigcap_{i=0}^{\infty} (\tau[f_i])$. The usual fixpoint theorems [11] guarantee that any continuous relational τ has a greatest fixpoint — $\bigcap_{i=0}^{\infty} \tau^i[\Omega]$ (where τ^i is the composition of τ with itself i times). In analogy to [12, Theorem 5-1], we have the following:

THEOREM (Continuous Relationals): *Any relational $\tau[F]$ defined by composition of programs and the relation variable F, is continuous.*

Proof. The proof is by induction on the structure of τ, as in the analogous theorem for continuous functionals. □

Here are some examples of continuous relationals (where p and g are some fixed nondeterministic programs):

$$\tau[F](x,y) = \textit{if } p(x) \textit{ then } y \textit{ else } x \textit{ OR } F(g(x),y)$$

$$\tau[F](x) = \textit{if } p(x) \textit{ then } x \textit{ else } x \textit{ OR } F(g(x))$$

2.4 Recursive definitions

A recursive nondeterministic n-ary program has the form

$$F(x_1, \cdots, x_n) \Leftarrow \tau[F](x_1, \cdots, x_n),$$

where τ is a continuous relational. In demonic semantics, the relation associated with this program is the greatest fixpoint of τ. For a recursive program P of the form $F(x_1, \cdots, x_n) \Leftarrow \tau[F](x_1, \cdots, x_n)$, we will denote the relation for P by $f_P = \bigcap_{i=0}^{\infty} \tau^i[\Omega]$.

3. VERIFICATION TECHNIQUES

The common verification methods developed for order-theoretic semantics can be applied to proofs of properties of f_P. Some of these techniques are demonstrated in this section.

3.1 Stepwise Computational Induction (*e.g.* Manna [12, section 5-3.1]; see also [13,19]).

Let P be the recursive program $f(\bar{x}) \Leftarrow \tau[f](\bar{x})$ To show that some property ψ holds for f_P, we can show three things:

- That $\psi(\Omega)$ holds.
- That whenever $\psi(f)$ holds, then $\psi(\tau(f))$ also holds.
- That ψ is an *admissible predicate.* (A predicate is admissible provided that for every continuous relational τ, if $\psi(\tau^i(\Omega))$ holds for all $i \geq 0$, then $\psi(\bigcap_{i=0}^{\infty}(\tau^i(\Omega)))$ also holds.)

This proof technique is called stepwise computational induction. Considerable effort has gone into showing that certain sorts of predicates are admissible. Most of this effort carries over to demonic semantics; For example, one of the results of Manna, Ness and

Vuillemin [13] generalizes to this lemma:

LEMMA (The admissible predicates lemma). *Let S be a subset of D^n. Every predicate which is of the form $\forall \bar{x} \in S{:}\alpha[F](\bar{x}) \subseteq \beta[F](\bar{x})$, where α and β are continuous relationals, is admissible. And, a finite conjunction of admissible predicates is admissible.* $\square$

EXAMPLE 3-1. Let $p \subseteq D \times D$ be a fixed program, and consider this recursive program:

$$P{:}F(x) \Leftarrow \text{if } p(x) \textit{ then } x \textit{ else } F(\cdots F \cdots)$$

For this example, the "$\cdots F \cdots$" can be filled in with any term formed from F, fixed programs and the argument x. Some simple properties follow from fixpoint properties. For example, if $p(\omega) \equiv D$, then the equality $f_P(D) \equiv D$ is obtained as follows:

$f_P(D) \supseteq f_P(\omega)$	Since $\omega \in D$
$\equiv \textit{if } p(\omega) \textit{ then } \omega \textit{ else } f_P(\cdots f_P \cdots)$	Fixpoint property
$\supseteq \textit{if } \omega \textit{ then } \omega \textit{ else } f_P(\cdots f_P \cdots)$	$p(\omega) \equiv D$
$\equiv D$	Definition of *cond*

A more complex argument (using the fact that p is a program) shows this same equality holds even when $p(\omega) \not\equiv D$.

Stepwise computational induction can be used to show that $f_P f_P = f_P$. The proof is not difficult, but it does demonstrate the extra care that must be taken to handle possible nondeterminism in p. The proof uses the admissible predicate $\psi(F)$ defined by

$$\psi(F){:}\ \ \forall x \in D\,[f_P(F(x)) \equiv F(x)]$$

Base step: We must show $\psi(\Omega)$. For any $x \in D$, $f_P(\Omega(x)) \equiv f_P(D) \equiv D \equiv \Omega(x)$, which asserts $\psi(\Omega)$.

Induction step: We must show $\psi(\tau[f])$ holds whenever $\psi(f)$ holds (where τ is the continuous relational used to define f_P). For any $x \in D$, there are these four subcases:

Case 1: $p(x) \not\subseteq \{true, false\}$

It is easy to show that both $f_P(\tau[f](x))$ and $\tau[f](x)$ are D.

Case 2: $p(x) = \{true\}$

Then $\tau[f](x) \equiv \text{if } p(x) \text{ } then \text{ } x \text{ } else \text{ } f(\cdots f \cdots) \equiv \{x\}$. Hence

$$f_P(\tau[f](x)) \equiv f_P(x) \equiv \text{if } p(x) \textit{ then } x \textit{ else } f_P(\cdots f_P \cdots) \equiv \{x\} \equiv \tau[f](x)$$

Case 3: $p(x) = \{false\}$

Then $\tau[f](x) \equiv \text{if } p(x) \textit{ then } x \textit{ else } f(\cdots f \cdots) \equiv f(\cdots f \cdots)$. Therefore:

$$f_P(\tau[f](x)) \equiv f_P(f(\cdots f \cdots)) \equiv f(\cdots f \cdots) \equiv \tau[f](x)$$

(The second equality follows from the induction hypothesis.)

Case 4: $p(x) = \{true, false\}$

This case is a combination of cases 2 and 3, using the equality $\tau[f](x) \equiv x \textit{ OR } f(\cdots f \cdots)$.

Thus, since ψ is admissible, $\psi(f_P)$ holds, which implies $f_P f_P \equiv f_P$.

EXAMPLE 3-2. Let $p \subseteq D \times D$ and $h \subseteq D \times D$ be programs, and examine these recursive programs:

$$P{:}F(x,y) \Leftarrow (\textit{if } p(x) \textit{ then } y \textit{ else } F(h(x),y)) \textit{ OR } (\textit{if } p(y) \textit{ then } x \textit{ else } F(x,h(y)))$$

$$Q{:}G(x) \Leftarrow \text{if } p(x) \textit{ then } x \textit{ else } (x \textit{ OR } G(h(x)))$$

We will show that $f_P(x,y) \equiv f_Q(x) \cup f_Q(y)$. This equality may seem surprising at first, because it is not true in the usual relational semantics, nor in the extended relational semantics which includes ω together with *ad hoc* rules. It is only true in the demonic semantics, where possible nontermination causes disaster.

The proof is in several parts; the first part of the proof uses the subset $S \equiv \{z \in D \mid \forall i: p(h^i(z)) \not\equiv \{true\}\}$. We will use stepwise computational induction to show that for all $z \in S$: $f_Q(z) \equiv D$. The proof uses this admissible predicate:

$$\psi(G): \forall z \in S\ [G(z) \equiv D].$$

Base step: We must show $\psi(\Omega)$. This is easy since Ω is the largest relation, so $\Omega(x)$ is always D.

Induction step: Let g be a program where $\psi(g)$ holds. We must show that $\psi(\tau[g])$ also holds (where τ is the continuous relational used to define f_Q). Toward this end, let $z \in S$. Then:

$$\tau[g](z)\ \supseteq\ z \cup g(h(z)) \equiv z \cup D \equiv D.$$

The first inclusion follows from the definition of τ (and $z \in S$); the next equality holds by the induction hypothesis and the fact that at least one element of $h(z)$ is in S.

In a similar manner we can also show that for all $z \in S$ and $x \in D$:

$$f_P(z,x) \equiv D \equiv f_P(x,z).$$

Again, this induction was not difficult, but we can employ it to show the original goal that $f_P(x,y) \equiv f_Q(x) \cup f_Q(y)$, for any $x,y \in D$. Here are the cases:

Case 1: $x \in S$ or $y \in S$

Then $f_P(x,y) \equiv D \equiv f_Q(x) \cup f_Q(y)$ — by the above computational induction.

Case 2: Neither x nor y is in S

Let i be the smallest integer such that $p(h^i(x)) \equiv \{true\}$, and let j be the smallest integer such that $p(h^i(y)) \equiv \{true\}$. The proof is a double induction on i and j; it makes use of the fact that f_P and f_Q are fixed-points of the corresponding equations.

3.2 Fixpoint Induction (*e.g.* Manna [12, page 406]; see also [15]).

Demonic relational semantics gives the meaning of a recursive program as the greatest fixpoint of a continuous relational with respect to the $\subseteq$ order. As a result, properties of greatest fixpoints, such as this, are available for program verification:

LEMMA (Fixpoint induction). *Let τ be a continuous relational and let f_P be the greatest fixpoint of τ. Then for any program g, $\tau[g] \supseteq g$ implies $f_P \supseteq g$.*

EXAMPLE 3-3. Let $p, g \subseteq D \times D$ be programs, and consider these recursive programs:

$$S\colon F(x) \Leftarrow \text{if } p(x) \textit{ then } x \textit{ else } F(g(x))$$

$$T\colon F(x) \Leftarrow \text{if } p(x) \textit{ then } x \textit{ else } F(F(g(x)))$$

$$U\colon F(x) \Leftarrow \text{if } p(x) \textit{ then } x \textit{ else } F(g(x)) \textit{ OR } F(F(g(x)))$$

The three continuous relationals associated with these programs will be denoted σ, τ and υ, while the programs themselves are f_S, f_T and f_U. Note that for any relation f, $\sigma[f] \subseteq \upsilon[f]$ and $\tau[f] \subseteq \upsilon[f]$, which implies $f_S \subseteq f_U$ and $f_T \subseteq f_U$. Fixpoint induction can be used to show the opposite relations $f_S \supseteq f_U$ and $f_T \supseteq f_U$ — which implies that all three programs are equivalent. Here are the fixpoint induction proofs:

$f_S \supseteq f_U$:

From the fixpoint induction lemma we must show that $\sigma[f_U] \supseteq \mathrm{f}_U$. Actually, we can show the stronger condition $\sigma[f_U] \equiv \mathrm{f}_U$, as follows. For any $x \in D$:

$\sigma[f_U](x) \equiv$ *if* $p(x)$ *then* x *else* $f_U(g(x))$ — Definition of σ

$\equiv$ *if* $p(x)$ *then* x *else* — Fixpoint property of f_U

[*if* $p(x)$ *then* x *else* $(f_U(g(x))$ *OR* $f_U(f_U(g(x))))$]

$\equiv$ *if* $p(x)$ *then* x *else* $(f_U(g(x))$ *OR* $f_U(f_U(g(x))))$ — Simplification

$\equiv f_U(x)$ — Fixpoint property of f_U

$f_T \supseteq f_U$:

From the fixpoint induction lemma we must show that $\tau[f_U] \supseteq f_U$. Again, we can show the stronger condition $\tau[f_U] \equiv f_U$. To prove this, note that example 3-1 implies that $f_U f_U \equiv f_U$, therefore, for any $x \in D$:

$\tau[f_U](x) \equiv$ *if* $p(x)$ *then* x *else* $f_U(f_U(g(x)))$ — Definition of τ

$\equiv$ *if* $p(x)$ *then* x *else* $f_U(g(x))$ — Example 3–1

$\equiv \sigma[f_U](x)$ — Definition of σ

$\equiv f_U(x)$ — Previous case

3.3 Structural Induction (*e.g.* Manna [12, page 408]; see also [5]).

Structural induction is a method for proving properties about an n-ary recursive program when the set D^n is equipped with a *well-founded* partial order $\lhd$. The fact that the partial order is well-founded means that there are no infinite descending chains $x_0 \rhd x_1 \rhd x_2 \rhd \cdots$ (where $\rhd$ is the inverse relation to $\lhd$.) For example, the natural numbers are well-founded, with the usual ordering.

LEMMA (Structural Induction). *Let $\lhd$ be a well-founded order on a set S, and let $\psi: S \rightarrow \{true, false\}$ be a predicate on S. Then $\psi(x)$ holds for all $x \in S$ provided that for all $x \in S$, $\psi(x)$ is implied by $\bigwedge_{y \lhd x} \psi(y)$.*

EXAMPLE 3-4. As an application, let S be the set of all finite non-empty strings over the alphabet $\{a,b\}$, and let $D = S \cup \{true, false, \omega\}$. Here are several partial functions on strings:

$simple(x)$ is *true* if x has length one, and *false* otherwise.
$cat(x,y)$ is the catenation of the strings x and y.
$head(x)$ is the one-character string, consisting of the initial character of x.
$tail(x)$ is the one-character string, consisting of the final character of x.
$first(x)$ is x with the final character removed (undefined for strings of length one).
$last(x)$ is x with the initial character removed (undefined for strings of length one).

Using the demonic extensions of these functions, we can define this recursive program:

$$rev(x) \Leftarrow \text{if } simple(x) \text{ } then \text{ } x$$
$$else \text{ } cat(rev(tail(x)), first(x)) \;\cup\; cat(last(x), rev(head(x)))$$

What follows is a proof by structural induction of a property of *rev*. The well-founded order on S is the order $x \lhd y$ iff the length of x is less than the length of y. The proof makes use of these properties of the relations (which are not proved here):

(1) $first(cat(x,y)) = first(x)$.
(2) $first(last(x)) = last(x)$.
(3) If not $simple(x)$ then $tail(x) \lhd x$.
(4) If $simple(x)$ then $first(x) = last(x)$.
(5) If not $simple(y)$ then $last(tail(x)) = last(x)$.

THEOREM: *For any string* $x \in S$, $first(rev(x)) = last(x)$.

Proof. Let $x \in S$ be a string. By the hypothesis of structural induction we may assume that whenever $u \lhd x$ then $first(rev(u))$ is $last(u)$. We must prove that this implies the theorem for x.

If $simple(x)$ holds, then: $first(rev(x)) = first(x) = last(x)$. The first equality is from the definition of *rev*, and the second is from property 4, above.

On the other hand, if $simple(x)$ is *false*, then:

$first(rev(x))$

$\equiv first(cat(rev(tail(x)),first(x)) \cup cat(last(x),rev(head(x))))$	Definition *rev*
$\equiv first(cat(rev(tail(x)),first(x))) \cup first(cat(last(x),rev(head(x))))$	
$\equiv first(rev(tail(x))) \cup first(last(x))$	Property 1
$\equiv first(last(tail(x))) \cup first(last(x))$	Induction & Prop. 3
$\equiv last(tail(x)) \cup last(x)$	Property 2
$\equiv last(x)$	Property 5

$\square$.

Of course, *rev* has much stronger properties than this, which can also be proved by structural induction.

4. OTHER NONDETERMINISTIC LANGUAGES

The idea demonstrated here is this: a special case of order-theoretic semantics is obtained by a relational semantics where recursive equations are solved by taking the greatest fixpoint (with respect to the subset ordering). The result is a simple semantics for total correctness of nondeterministic programs — and we can use the usual order-theoretic proof techniques.

The idea has been demonstrated for nondeterministic recursive programs, but the same idea is applicable to programs schemes (such as [2,10,14]) or nondeterministic iterative languages. For example, the meaning of an iterative statement *while b do S od* is the greatest fixpoint of

$$F = if\ \ b\ \ then\ \ ([\![S]\!];F)\ \ else\ \ I,$$

where $[\![S]\!]$ is the meaning of the statement S, and I is the demonic extension of the identity function.

References

(1) S. Abramsky. Experiments, powerdomains and fully abstract models for applicative multiprogramming, in: *Foundations of Computation Theory*, LNCS 158, (Springer-Verlag, 1983), 1-13.

(2) A. Arnold and M. Nivat. Non-deterministic recursive program schemes, in: *Foundations of Computation Theory*, LNCS 56, (Springer-Verlag, 1977), 12-21.

(3) J.W. deBakker. Semantics and termination of nondeterministic recursive programs, in: *Automata, Languages and Programming* (1976), 435-477.

(4) M. Broy. On the Herbrand-Kleene universe for nondeterministic computations, *Theoretical Computer Science* 36 (1985), 1-20.

(5) R.M. Burstall. Proving properties of programs by structural induction, *Comput. J.* 1 (1969), 41-48.

(6) E.W. Dijkstra. *A Discipline of Programming*, (Prentice-Hall, 1976).

(7) D. Harel. First-order dynamic logic, LNCS 68 (Springer-Verlag, 1979).

(8) D. Harel and V.R. Pratt. Nondeterminism in Logic of Programs, in: *Proceedings of the 5th ACM Symposium on Principles of Programming Languages* (1978), 203-213.

(9) M.C.B. Hennessy. The semantics of call-by-value and call-by-name in a nondeterministic environment, *SIAM J. Computing* 9 (1980), 67-84.

(10) M.C.B. Hennessy and E.A. Ashcroft. Parameter-passing mechanisms and nondeterminism, in: *Proceedings of the 9th Annual ACM Symposium on Theory of Computing*, (1977), 306-311.

(11) J.-L. Lassez, V.L. Nguyen and E.A. Sonenberg. Fixed-point theorems and semantics: a folk tale, *IPL* 14 (1982), 112-116.

(12) Z. Manna. *Mathematical Theory of Computation*, (McGraw-Hill, 1974).

(13) Z. Manna, S. Ness and J. Vuillemin. Inductive methods for proving properties of programs, *CACM* 16(1973), 491-502.

(14) M. Nivat. Nondeterministic programs: an algebraic overview, in *Information Processing 80* (S.H. Lavington, Ed.), (North-Holland Publishing Co., 1980), 17-28.

(15) D. Park. Fixpoint induction and proofs of program properties, in: *Machine Intelligence 5*, (American Elsevier, 1970).

(16) G.D. Plotkin. Dijkstra's predicate transformers and Smyth's powerdomains, in: *Abstract Software Specifications*, LNCS 86, (Springer-Verlag, 1980), 527-553.

(17) G.D. Plotkin. *Computer Science Postgraduate Course Notes*, University of Edinburgh, 1980-81.

(18) M. Smyth. Powerdomains, *Journal of Computer and System Sciences* 16 (1978), 23-36.

(19) Denotational semantics: the Scott-Strachey approach to programming language theory, (MIT Press, 1977).

(20) G. Winskel. A note on powerdomains and modality, in: *Foundations of Computation Theory*, LNCS 158, (Springer-Verlag, 1983), 505-514.