2

June 1988

Report No. STAN-CS-88-1210

# String-Functional Semantics for Formal Verification of Synchronous Circuits

by

Alexandre Bronstein and Carolyn L. Talcott

Department of Computer Science

Stanford University
Stanford, California 94305

# String-Functional Semantics for Formal Verification of Synchronous Circuits

Alexandre Bronstein & Carolyn L. Talcott

This research was partially supported by Digital Equipment Corp. and by ARPA contract N00039-84-C-0211.

SECURITY CLASSIFICATION OF THIS PAGE

# REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188
Exp Date Jun 30, 1986

1a REPORT SECURITY CLASSIFICATION
unclassified

1b RESTRICTIVE MARKINGS

2a SECURITY CLASSIFICATION AUTHORITY

2b DECLASSIFICATION / DOWNGRADING SCHEDULE

3 DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release:
Distribution Unlimited

4 PERFORMING ORGANIZATION REPORT NUMBER(S)
STAN-CS-88-1210

5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a NAME OF PERFORMING ORGANIZATION
Computer Science Department

6b OFFICE SYMBOL *(If applicable)*

7a. NAME OF MONITORING ORGANIZATION

6c. ADDRESS *(City, State, and ZIP Code)*
Stanford University
Stanford, CA 94305

7b. ADDRESS *(City, State, and ZIP Code)*

8a NAME OF FUNDING / SPONSORING ORGANIZATION
DARPA

8b OFFICE SYMBOL *(If applicable)*

9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c. ADDRESS *(City, State, and ZIP Code)*
1400 Wilson Blvd.
Arlington, VA 22209

10. SOURCE OF FUNDING NUMBERS

| PROGRAM ELEMENT NO. | PROJECT NO. | TASK NO. | WORK UNIT ACCESSION NO |
|---|---|---|---|
| | | | |

11 TITLE *(Include Security Classification)*
String-Functional Semantics for Formal Verification of Synchronous Circuits

12 PERSONAL AUTHOR(S)
Alexandre Bronstein and Carolyn Talcott

13a TYPE OF REPORT

13b TIME COVERED
FROM ________ TO ________

14. DATE OF REPORT *(Year, Month, Day)*
June 1988

15 PAGE COUNT
57

16 SUPPLEMENTARY NOTATION

17 COSATI CODES

| FIELD | GROUP | SUB-GROUP |
|---|---|---|
| | | |
| | | |

18 SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*

19 ABSTRACT *(Continue on reverse if necessary and identify by block number)*

A new functional semantics is proposed for synchronous circuits, as a basis for reasoning formally about that class of hardware systems.

Technically, we define an extensional semantics with monotonic length-preserving functions on finite strings, and an intensional semantics based on functionals on those functions. As support for the semantics we prove the equivalence of the extensional semantics with a simple operational semantics, as well as a characterization of circuits which obey the "every loop is clocked" design rule.

Also, we develop the foundations in complete detail both to increase confidence in the theory, and as a prerequisite to its future mechanization. —

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT
□ UNCLASSIFIED/UNLIMITED □ SAME AS RPT □ DTIC USERS

21 ABSTRACT SECURITY CLASSIFICATION

22a NAME OF RESPONSIBLE INDIVIDUAL

22b. TELEPHONE *(Include Area Code)*

22c OFFICE SYMBOL

83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

# Table of Contents

# List of Figures

# 1. Introduction

## 1.1. Motivation

Hardware design could benefit greatly from a precise computation theory of hardware systems. Current design and validation methods, such as simulation and testing are expensive and unreliable. The call for *formal* methods in hardware design is heard more and more in the hardware community, and not only among theoreticians, but also among practitioners as in [Russell-Kinniment-Chester-McLauchlan 85] (p.189):

> As the designs get bigger this [validation] capability will not be provided by traditional simulators. Formal verification of some other kind will need to be employed, which means that current languages will need to be redesigned to encompass formal techniques.

Formal verification, such as mechanical proof of correctness or transformation-based (inferential) design systems [Burstall-Darlington 77], [Scherlis-Scott 83] , requires a formal underlying semantics, and this is what we mean by a "precise computation theory of hardware systems".

This is not an entirely new concept! Such a formal theory has been around for a long time for a *small* class of hardware systems: combinational circuits. Their semantics are given in terms of Boolean functions, and theoretical applications include equivalences proofs using the Boolean calculus, minimization theorems, and many more advanced theories such as fault-modelling and test-generation. In fact, the Boolean Algebra semantics is ubiquitous in the education of hardware engineers.

Our goal was therefore to find similarly natural and mathematically tractable semantics for more general hardware systems, to serve as a basis for reasoning formally about hardware designs.

## 1.2. Solution proposed

Using functions on finite strings as a basic mathematical object, we have developed the core of a formal theory for a wider class of hardware: synchronous systems/circuits.

The basic ideas and relation to the Boolean function semantics are fairly simple and we have made a special effort to include a detailed, motivated, *informal* explanation in section 3.1 . Technically we build Scott-style domains of strings, and string-functions, and give the extensional semantics of a synchronous circuit in terms of monotonic (with respect to less-defined-than and prefix) and length-preserving string-functions. Note however that in contrast to other work in concurrency theory based on strings, we need only *finite* strings, and use as our primary ordering the pointwise extension of the flat ordering on the base domain, not the prefix ordering. Correspondingly, we solve our fixed point equations in the string-function domain, and not in the string domain. The beginning of a calculus based on these functional extensional semantics is shown among the possible theoretical applications in section 4.1.

In order to reason about synchronous systems in an even more general and powerful manner, we have added a recent idea of software computation theory: intensional semantics . These give a mathematical handle on *how* an algorigthm (or in our case, a circuit) computes its result, as opposed to just *what* the result is, i.e. its extensional semantics. These concepts are studied in great depth in [Talcott 85] and [Moschovakis 83]. They provide a way to compare precisely the objects we are trying to design, and hence provide the relations which will be at the core of future "guaranteed correct" transformation-based design systems [Scherlis-Scott 83]. A very limited taste of such relations is given in section 4.2.

These constitute the main ideas presented in this report. In order to support them however, we have proved a few additional results about our semantics:

- We have given a semantic characterization of synchronous circuits which obey the "Every Loop is

Clocked" design rule, even though our semantics assign a meaning to all circuits (built arbitrarily from primitive components: registers and gates). We have not seen such characterization (in any form) anywhere else in the hardware semantics literature.

- We have defined an *operational* semantics which is extremely simple, and basically a trivial circuit simulation algorigthm, and proved its equivalence to our extensional semantics. We also believe this result to be new in the context of hardware systems, although related operational-denotational equivalence proofs have appeared in the context of dataflow [Faustini 82a] and more clearly [Glasgow-MacEwen 87] within operator nets.
- We have shown how to apply these semantics to Sequential Machines (Mealy Machines [Booth 67], [Hopcroft-Ullman 79]) which are at the core of synchronous circuit design in the engineering community. This allows us to formally state that a certain circuit correctly implements a certain sequential machine.

Finally, since our denotational semantics is based on a *new* domain of string-functions, and since ultimately all claims of design correctness rely on sound underlying mathematics, and since a precise and thorough understanding of the theory is an essential prerequisite to its mechanization (in a theorem-prover), we have taken extreme care to develop the *foundations* in complete detail.

In order to reach the full generality that we needed, such as combinations of functions with arbitrary (and different) number of inputs, without any hand-waving, we found that we had to use some slightly technical tools, such as Moschovakis' induction algebras. Moreover, we isolated two mathematical structures which came up during the process and seemed to present some interest:

- Finite Depth domains, which are generalizations of flat domains, and
- String domains, which are domains generated from a base domain with string operations.

To prevent confusion between these developments and their applications to hardware semantics, and spare less mathematically inclined readers, we have placed them in a separate "Foundations" chapter (chapter 2).

## 1.3. Relation to other work

The original inspiration for this work came from software concurrency theory and the work of [Kahn 74] on semantics of asynchronous communicating processes. The key idea there was to view each node as history- (or string-)functional, the system as a list of string equations, and define the result to be the least solution (or fixed point) of the system, in a domain of infinite strings ordered by the prefix relation. Other people then tried to exhibit operational models for which they could prove the appropriateness of the "Kahn-semantics" [Arnold 81], [Faustini 82a], [Faustini 82b] and references therein.

In our case, we have kept the basic idea of nodes being string-functional, but because of our *synchronous* context, we were able to use a domain of *finite* strings, ordered by a pointwise extension of the flat ordering on the base domain. Also, we made the abstraction to string-*functions* for circuits, which was only implicit in [Kahn 74]. Moreover we view the equations as defining string-functions instead of strings, and correspondingly solve our fixed point system in a functional domain.

Much of the work derived from [Kahn 74] in concurrency theory has gone into trace theory, keeping the history idea, but tossing away the functional abstraction, mainly to deal with limitations of [Kahn 74] in non-deterministic contexts, as pointed out in [Brock-Ackerman 81]. These have been successfully applied to VLSI in [van de Snepscheut 85] and recently in [Dill 88] to asynchronous circuits. However *synchronous* systems do not present any of the difficulties necessitating trace theory. And fundamentally, we believe the functional abstraction to be natural and crucial for the design of large systems, for a rich calculus of synchronous circuits (analogous to the Boolean calculus), and for the intuitive understanding of systems.

Also inspired by the work of Kahn, and trying to apply these ideas to the semantics of hardware, are the works of [Brookes 84] and recently [Kloos 87]:

[Brookes 84] uses infinite strings (viewed as functions on integers) but is fairly informal and based only on one example, which does not have any feedback. His remark concerning the handling of feedback is essentially wrong (or extremely imprecise) since the original state of the registers seems not to be kept in the syntactic object, even though in the presence of feedback, it can affect the final semantics immensely.

[Kloos 87] in contrast is quite formal and thorough, and is very much based on Kahn's idea of functions on infinite strings, with a (slightly modified) prefix ordering due to Broy. This work is the most similar to ours that we have found, and goes a long way towards achieving many of our goals, within a different mathematical environment and for the extensional part only. It is however, much broader in its scope of harware systems it aims to model, and correspondingly, the theory is weaker. Moreover, the algebra of finite strings has many advantages for purposes of mechanizing, such as induction. Also, no proof of equivalence with any operational model or other key property of the semantics is given.

Much other work related to ours falls under the category of "new hardware languages". These have evolved very similarly to software languages: from ad-hoc (assembly) to clearer (high-level) to semantically cleaner (functional) . Just like in software, very few of them really have formal underlying semantics. Two notable exceptions are [Sheeran 83] and [Johnson 83]:

[Sheeran 83] uses FP [Backus 78] as a semantic base, and hence functions on sequences. Aside from an insistence on a variable-free (and hence hardly readable) style, there is a lot of emphasis on algebraic laws, so "philosophically" our work is very related to hers.

[Johnson 83] uses a more standard applicative notation but puts much more emphasis on the language issue than on the semantics. Most of the emphasis is on (informally) transforming recursive descriptions of the algorithm which are not directly implementable in hardware, into other descriptions which are. The semantics only model a special restricted "stylized" kind of circuit (with one "output" line and one "ready" line). The model-theoretic semantics are sketched rapidly, are not very natural (signals are "infinite sequences of instantaneous operations"), and are clearly not the main goal in his work.

Finally, work in mechanical correctness proofs of hardware shares some important goals with us, although we believe that semantics should be thoroughly studied first. The most impressive such result we know so far is [Hunt 85] where two descriptions of a CPU (one of which was isomorphic to the actual hardware) were proved equivalent in the Boyer-Moore system. The semantics however, while quite clear in the combinational logic case, are more fuzzy in the sequential case, where a "stylized" description is used, with no formal justification. One price paid for this is the lack of compositionality, i.e. the unability to combine easily two separate (sequential) specifications into a bigger one. Also along the verification lines, we share a lot "in spirit" with Gordon's work in higher-order logic: [Gordon 85] and related efforts. Technically however we differ significantly. Gordon's semantics are axiomatic: hardware objects are associated with predicates (on functions of time), and systems are "ANDed" together. Besides putting more emphasis on the model-theoretic aspects of our semantics, we have also defined our theory so that hardware systems are describable in just a first-order language. This may simplify automatic derivations, and in any case gives us a greater choice of theorem-provers. Moreover, by studying properties of the algebraic structure (i.e. building a calculus) we can derive system-independent properties.

## 1.4. Notation

We have tried as much as possible to use standard mathematical/logical notation: $\wedge$ , $\vee$ , => , <=> , $\forall$ and $\exists$ are the usual logical symbols. $\omega$ denotes the set of natural numbers (non-negative integers).

We've generalized slightly the tuple projection operator (denoted by subscripting): $(x_1,...,x_n)_i = x_i$ , to take a tuple of positions and return the corresponding sub-tuple of values: $(x_1,...,x_n)_{(i_1,...,i_k)} = \langle x_{i_1},...,x_{i_k})$ .

For our "precise" proofs, we have a semi-formal notation: There are two columns: assertions on the left, and justifications on the right, enclosed in double brackets, which can be mentally read as "because" or "by". Successful completion of the proof is indicated by:

[[]]

often indexed by the name of the theorem it proved. For example:

| | |
|---|---|
| We have $I = V / R$ | [[ Ohm, thm. 1 ]] |
| and $P = V * I$ | [[ definition ]] |
| $\therefore$ $P = V^2 / R$ | |
| and $V = 5.0$ volts | [[ hypothesis ]] |
| and $R \approx 0$ ohm | [[ we've reversed Vcc and Gnd pins ]] |

$[[]]_{\text{Thm. Chip-is-Hot}}$

In general, these proofs are most easily followed by *skipping the individual justifications*, i.e. reading the left column only! Occasionally, if a step appears unclear, then checking the justification is useful.

Other notations for particular structures (such as strings) are defined as concepts are defined. An index of major definitions is given at the end for "random-access" readers. The report itself is "linearly" organized in definition-theorem-proof form, each referring only to concepts previously defined or proved.

# 2. Mathematical Foundations of the Semantics

## 2.1. Basic Theory: CPOs, PCPOs, and Induction Algebras

The domains we consider are chain-complete partially ordered sets. However, since there are some terminology variations across the various authors in the field, we specify here the structures we will use, as well as the main results we'll need about them.

Many of these definitions and results can be found in various places and forms in [Manna 74] chapter 5, [de Bakker 80] chapters 3 and 5, and [Schmidt 86] chapter 6 .

Often however, these concepts (lub, continuity, fixed points) are obscured in standard treatments because they are defined in the specific context in which they are needed, which usually turns out to be a higher-order set where it is hard to visualize things. We have tried to avoid that pitfall here, and have defined each notion in the simplest structure in which it is meaningful.

**Definition 2.1: Partial Order [PO]**
$<P, \subseteq>$ is a Partial Order [PO] <=> $P$ is a set $\wedge$ $\subseteq$ is a binary relation on $P$ which is

- reflexive: $\forall x \in P, x \subseteq x$
- antisymmetric: $\forall x,y \in P, (x \subseteq y \wedge y \subseteq x \Rightarrow x=y)$
- transitive: $\forall x,y\ z \in P, (x \subseteq y \wedge y \subseteq z \Rightarrow x \subseteq z)$

**Definition 2.2: Upper Bound**
Let $<P, \subseteq>$ be a PO, $S$ be a subset of $P$, $y \in P$ is an Upper Bound of $S$ (in $P$) <=> $\forall x \in S, x \subseteq y$

**Definition 2.3: Least Upper Bound [LUB]**
Let $<P, \subseteq>$ be a PO, $S$ be a subset of $P$, $y \in P$ is a Least Upper Bound of $S$ (in $P$) <=> y is an Upper Bound of $S$ $\wedge$ $\forall z \in P$, z Upper Bound of $S$ => $y \subseteq z$

**Definition 2.4: Chain**
Let $<P, \subseteq>$ be a PO, $S$ a subset of $P$, $S$ is a chain <=> $\forall x,y \in S, x \subseteq y \vee y \subseteq x$ (i.e. $\subseteq$ is total in $S$).

Note: we usually refer to chains as indexed by an ordinal I: $(x_i)_{i \in I} \mid \forall i \in I, x_i \subseteq x_{i+1}$ . This does not reduce the generality.

**Definition 2.5: Complete Partial Order [CPO]**
$<P, \subseteq>$ is a Complete Partial Order [CPO] <=> $<P, \subseteq>$ is a PO $\wedge$ every *non-empty* chain in $P$ has a LUB.

**Definition 2.6: Pointed Complete Partial Order [PCPO]**
$<P, \subseteq>$ is a Pointed CPO <=> $<P, \subseteq>$ is a CPO $\wedge$ there is a least element, usually called $\perp$, for $\subseteq$ in $P$ (i.e. the empty chain also has a lub).

The distinction between CPOs and PCPOs is often glossed over, because most domains used in practice are PCPOs ( [Schmidt 86], [Melton-Schmidt 86] make the distinction). In our case, we will deal with structures which are CPOs but not PCPOs, and therefore, we need the more general definitions.

Note that any PCPO is a CPO, and therefore all results true for CPOs apply to PCPOs. Also, an equivalent definition of PCPOs not referring to CPOs can be given, simply by requiring that "every chain has a LUB", but our definition makes the dependency on the empty chain explicit.

**Definition 2.7: Monotonic function on POs**
Let $<P_1, \subseteq_1>$ , $<P_2, \subseteq_2>$ be POs, f a function: $P_1 \rightarrow P_2$ , f is monotonic <=> $\forall$ x,y $\in P_1$ , x $\subseteq_1$ y => f(x) $\subseteq_2$ f(y)

**Definition 2.8: Continuous function on [P]CPOs**
Let $<P_1, \subseteq_1>$ , $<P_2, \subseteq_2>$ be PCPOs [resp. CPOs], f a function: $P_1 \rightarrow P_2$ , f is continuous <=> $\forall$ $(x_i)_{i \in I}$ [resp. non-empty] chain in $P_1$, $(f(x_i))_{i \in I}$ has a lub $\wedge$ f(lub $(x_i)_{i \in I}$) = lub$(f(x_i))_{i \in I}$
where the lubs are taken in the appropriate domains .

By considering a chain of just two elements we immediately get:

**Theorem 2.9: Continuous => Monotonic**
Let $<P_1, \subseteq_1>$ , $<P_2, \subseteq_2>$ be CPOs, and f a function: $P_1 \rightarrow P_2$ , f continuous => f monotonic .

The next two properties are immediate, but ofen useful:

**Theorem 2.10: Composition of monotonic functions**
Let $<P_1, \subseteq_1>$ , $<P_2, \subseteq_2>$ , $<P_3, \subseteq_3>$ be POs. Ler f be a function: $P_1 \rightarrow P_2$ , g be a function: $P_2 \rightarrow P_3$ , f and g are monotonic => $g \circ f : P_1 \rightarrow P_3$ , is monotonic.

**Theorem 2.11: Composition of continous functions**
Let $<P_1, \subseteq_1>$ , $<P_2, \subseteq_2>$ , $<P_3, \subseteq_3>$ be CPOs. Let f be a function: $P_1 \rightarrow P_2$ , g be a function: $P_2 \rightarrow P_3$ , f and g are continuous => $g \circ f : P_1 \rightarrow P_3$ , is continuous.

**Definition 2.12: Fixed Point of a function**
Let $S$ be an arbitrary set, f a unary function on $S$, x $\in S$ is a Fixed Point of f <=> f(x) = x .

Note that the preceding definition is a common mathematical notion, and applicable to any structure, not just CPOs. In Partially Ordered sets, we can additionally define the notion of a Least Fixed Point:

**Definition 2.13: Least Fixed Point [LFP] of a function**
Let $<P, \subseteq>$ be a PO, f a unary function on $P$, x $\in P$ is a Least Fixed Point of f <=> x is a fixed point of f
$\wedge$
$\forall$ y $\in P$ , y fixed point of f => x $\subseteq$ y .

One of the main reasons for using PCPOs as domains is that in these structures, a wide class of functions *have* least fixed points, which moreover can be computed explicitely:

**Theorem 2.14: Kleene**
A continuous function f, on a PCPO $<P, \subseteq>$, has a LFP in $P$ : lub$(f^i(\perp))_{i \in \omega}$ .

**Proof:**
This is an extension of Kleene's 1st Recursion theorem [Kleene 67] . Many proofs of this result exist in the literature, in various forms. One closest to our notation can be found in [Schmidt 86] p. 114 .

[[]]$_{\text{Thm. 2.14}}$

A useful generalization in [Moschovakis 77] extends this result to *families* of PCPOs, and systems of continuous functions on these CPOs. (Moschovakis' results are actually more general and deal with arbitrary induction and big ordinals. We restate them here in the simpler context of continuous induction, and consistently with our notations.)

**Definition 2.15: Induction Algebra**
$<(\boldsymbol{P}_j)_{j \in I}, (\subseteq_j)_{j \in I}, \boldsymbol{F}>$ is an induction algebra $<=>$ $\forall j \in I$, $<\boldsymbol{P}_j, \subseteq_j>$ is a PCPO $\wedge$ $\boldsymbol{F}$ is a set of functions $f: \boldsymbol{P}_{j_1} \times \ldots \times \boldsymbol{P}_{j_f} \rightarrow \boldsymbol{P}_{j_0}$, containing the identity maps, and closed under composition with projections.

By projection we mean a function of the form: $(x_1,\ldots,x_n) \rightarrow x_i$ for some $i \in \{1..n\}$.

By "closed under composition with projections" we mean that if $g \in \boldsymbol{F}$ and f satisfies: $f(x_1,\ldots,x_n) = g(\pi_1(x_1,\ldots,x_n),\ldots,\pi_m(x_1,\ldots,x_n))$ with $\pi_1,\ldots,\pi_m$ given projections, then $f \in \boldsymbol{F}$.

**Theorem 2.16: Kleene-Moschovakis**
Let $<(\boldsymbol{P}_j)_{j \in I}, (\subseteq_j)_{j \in I}, \boldsymbol{F}>$ be an induction algebra. Let $(f_1,\ldots,f_n)$ be a system of *continuous* functions in $\boldsymbol{F}$, where $\forall k \in \{1..n\}$, $f_k: \boldsymbol{P}_{j_1} \times \ldots \times \boldsymbol{P}_{j_n} \rightarrow \boldsymbol{P}_{j_k}$, then that system has a LFP in $\boldsymbol{P}_{j_1} \times \ldots \times \boldsymbol{P}_{j_n}$:
$\text{lub}[(f_1,\ldots,f_n)^i(\perp_{j_1},\ldots\perp_{j_n})]_{i \in \omega}$.

**Proof:**
See [Moschovakis 77], Lemmas 2.4 and 2.5. These actually apply to *monotone* functions, and conclude that the system has a fixed point:
$\text{lub}[(f_1,\ldots,f_n)^i(\perp_{j_1},\ldots\perp_{j_n})]_{i \in \kappa}$ with $\kappa$ some "big enough" ordinal.

Since in our case we are restricting ourselves to *continuous* functions, it is clear that $\omega$ is big enough:
We have $f[\text{lub}(f^i(\perp))_{i \in \omega}] = \text{lub}(f^{i+1}(\perp))_{i \in \omega}$ [[ continuity of f ]]
and $(f^{i+1}(\perp))_{i \in \omega} = (f^i(\perp))_{i \in \omega} - \{\perp\}$
$\therefore$ $\text{lub}(f^{i+1}(\perp))_{i \in \omega} = \text{lub}(f^i(\perp))_{i \in \omega}$
$\therefore$ $f[\text{lub}(f^i(\perp))_{i \in \omega}] = \text{lub}(f^i(\perp))_{i \in \omega}$
$\therefore$ $\text{lub}(f^i(\perp))_{i \in \omega}$ is a fixed point. And the same proof obviously carries through to a tuple of functions.

$[[]]_{\text{Thm. 2.16}}$

A few other results which help us build CPOs and PCPOs are enumerated below.

**Theorem 2.17: Product of CPOs**
The cartesian product of CPOs is a CPO (under the induced coordinate-wise ordering), and the lub of a chain of tuples is the tuple of the lubs of the coordinates (i.e. the tupl-ing operation is continuous).

This generalizes immediately to finite product.

**Theorem 2.18: Product of PCPOs**
The cartesian product of PCPOs is a PCPO (under the induced coordinate-wise ordering).

This also generalizes immediately to finite product.

**Theorem 2.19: Disjoint union of CPOs**
The disjoint union of CPOs is a CPO (under the union of the ordering relations).

This generalizes to *arbitrary* unions with the following definition: $\cup(\boldsymbol{P}^i)_{i \in I} = \{x \mid \exists i \in I \mid x \in \boldsymbol{P}^i\}$, where the $\boldsymbol{P}^i$'s are all disjoint.

Note however that the disjoint union of PCPOs is not a PCPO (we need to add a new least element in order to obtain a PCPO). It is common in Scott-style semantics to add that extra element without even mentioning it when dealing with PCPOs. We will *not* do that. We still clearly have that the disjoint union of PCPOs is a CPO, which will be enough for our purposes.

As for Kleene's theorem, proofs for the preceding constructions can be found in [Schmidt 86].

**Definition 2.20: Sub-CPO**
Let $<P, \subseteq>$ be a CPO, $P_1$ is a subset of $P$, $P_1$ is a sub-cpo of $P$ $<=>$ $<P_1, \subseteq_{\text{restricted to } P_1}>$ is a CPO.

Note the following two subtleties about sub-cpos:

- In general, subsets of CPOs are not sub-CPOs (counterexample: $\omega+1$, with subset: $\omega$).
- In general, LUBs (of a single chain) in a CPO and a sub-CPO are not necessarily the same (counterexample: $\omega+2$, sub-cpo: $\omega+2 - \{\omega\}$, chain: $\{0,1,...\}$).

The following notion is not as "standard" but very useful in building "nice" sub-CPOs, and we will use it extensively in the rest of this work:

**Definition 2.21: Strongly Admissible predicate on a CPO**
Let $<P, \subseteq>$ be a CPO. Let $\phi$ be a predicate on elements of $P$. $\phi$ is Strongly Admissible on $P$ $<=>$ $\forall (x_i)_{i \in I}$ non-empty chain in $P$, $(\forall i \in I, \phi(x_i))$ $=>$ $\phi(\text{lub } (x_i)_{i \in I})$ .

In other words,"$\phi$ carries to the lub". Note that this property is closely related to, but slightly stronger than, the notion of "admissible" predicate in computational induction [Manna 74].

**Theorem 2.22: "Nice" Sub-CPOs**
Let $<P, \subseteq>$ be a CPO, let $\phi$ be a strongly admissible predicate on $P$, then $P \cap \phi = \{ x \in P \mid \phi(x) \}$, is a sub-CPO of $P$, *and* the LUBs of chains in both domains are the same.

**Proof:**
Immediate by def. 2.21. I.e. we've defined "Strongly Admissible" to be exactly what we needed for this theorem to be true; the work will be in proving that specific properties we're interested in *are* in fact strongly admissible.

$[[]]_{\text{Thm. 2.22}}$

We now move on to function domains. We can easily extend the ordering of a Partially Ordered set to an ordering on its functions:

**Definition 2.23: Pointwise function ordering**
Let $<P_1, \subseteq_1>$, $<P_2, \subseteq_2>$ be POs, f,g functions: $P_1 \rightarrow P_2$, $f \subseteq_{\text{pointwise}} g$ $<=>$ $\forall x \in P_1$, $f(x) \subseteq_2 g(x)$ .

It is immediate that $\subseteq_{\text{pointwise}}$ is reflexive, antisymmetric and transitive. The subscript "$_{\text{pointwise}}$" is usually dropped since the correct relation can be inferred from context.

Note that this definition immediately applies to functions of arbitrary arity, by considering them as unary functions from the product PO.

**Function domains on CPO**: In the literature, one usually finds a proof that the set of monotonic functions on a CPO is a CPO, or that the set of continuous functions on a CPO is a CPO. However, many more function domains on a CPO can be usefully built, as the next few theorems show.

**Theorem 2.24: $P_1^{P_2}$ is a CPO.**
Let $<P_1, \subseteq_1>$, $<P_2, \subseteq_2>$ be CPOs, the set of all functions from $P_1$ to $P_2$: $P_2^{P_1}$, under the pointwise ordering, is a CPO.

The proof is fairly standard. However, we give it because we will need to refer explicitely to the contruction of the lub of a function-chain in many other occasions.

**Proof:**

Assume [h1] $<P_1, \subseteq_1>$ CPO, [h2] $<P_2, \subseteq_2>$ CPO, and [h3] $(f_i)_{i \in I}$ non-empty chain in $P_2^{P_1}$.

Define (and this is the essence of the proof) $f = \lambda x.\mathrm{lub}(f_i(x))_{i \in I}$, we prove that 1) $f \in P_2^{P_1}$ and 2) f is lub $(f_i)_{i \in I}$.

1) Let $x \in P_1$, arbitrary.

We have $\forall i \in I, f_i \subseteq f_{i+1}$ [[ h3 ]]

$\therefore$ $\forall i \in I, f_i(x) \subseteq_2 f_{i+1}(x)$ [[ def. 2.23 ]]

$\therefore$ $\{ f_i(x), _{i \in I} \}$ is a non-empty chain in $P_2$ [[ def. 2.4 ]]

$\therefore$ $\{ f_i(x), _{i \in I} \}$ has a lub in $P_2$ [[ h2 ]]

and this was done for arbitrary x,

$\therefore$ f is a (well-defined) function from $P_1$ to $P_2$ .

$[[]]_1$

2) Let $i \in I$, arbitrary.

We have $\forall x \in P_1, f_i(x) \subseteq_2 \mathrm{lub}(f_i(x))_{i \in I}$ [[ def. 2.3, LUB => Upper Bound ]]

$\therefore$ $\forall x \in P_2, f_i(x) \subseteq_2 f(x)$ [[ construction of f ]]

$\therefore$ $f_i \subseteq f$ [[ def. 2.23 ]]

and this was done for arbitrary i,

$\therefore$ f is an upper bound of $(f_i)_{i \in I}$. [[ def. 2.2 ]]

Assume [h4] $g \in P_2^{P_1} \mid \forall i \in I, f_i \subseteq g$

Let $x \in P_1$, arbitrary.

We have $\forall i \in I, f_i(x) \subseteq_2 g(x)$ [[ h4, def. 2.23 ]]

$\therefore$ $\mathrm{lub}(f_i(x))_{i \in I} \subseteq_2 g(x)$ [[ def. 2.3 ]]

$\therefore$ $f(x) \subseteq_2 g(x)$ [[ construction of f ]]

and this was done for arbitrary x,

$\therefore$ $f \subseteq g$ [[ def. 2.23 ]]

$\therefore$ $f = \mathrm{lub}\ (f_i)_{i \in I}$

$[[]]_2$

$[[]]_{\text{Thm. 2.24}}$

As an immediate corollary we get:

**Theorem 2.25: $P^{P^n}$ is a CPO.**

Let $<P, \subseteq>$ be a CPO, the set of all functions (of arity n) on $P$: $P^{P^n}$, under the pointwise ordering, is a CPO.

As an immediate application of the preceding theorem (thm. 2.24) and our notion of strongly admissible predicates (thm. 2.22), we get a whole class of function CPOs:

**Theorem 2.26: Function domains on CPOs**

Let $<P_1, \subseteq_1>$, $<P_2, \subseteq_2>$ be CPOs. Let $\phi$ be a strongly admissible predicate on $P_2^{P_1}$, then $P_2^{P_1} \cap \phi = \{ f \in P_2^{P_1} \mid \phi(f) \}$, under the pointwise ordering, is a CPO. And, the LUB of a function-chain in $P_2^{P_1} \cap \phi$ is the same as the LUB in $P_2^{P_1}$ .

**Theorem 2.27: Corollary: Monotonic functions CPO, Continuous functions CPO**
Let $<P_1, \subseteq_1>$, $<P_2, \subseteq_2>$ be CPOs. The following sets of functions, under the pointwise ordering, are CPOs:

- set of all monotonic functions: $[\, P_1 \rightarrow P_2 \,]$,
- set of all continuous functions: $(\, P_1 \rightarrow P_2 \,)$.

**Proof:**

$\phi(f)$ = "f is monotonic" is strongly admissible on $P_2^{P_1}$:

Assume [h1] $(f_i)_{i \in I}$ non-empty chain of monotonic functions from $P_1$ to $P_2$ .

We have $f = \lambda x.\mathrm{lub}(f_i(x))_{i \in I} = \mathrm{lub}\,(f_i)_{i \in I}$ [[ construction of lub of function-chains ]]

Let $x, y \in P_1 \mid x \subseteq_1 y$

We have $\forall\, i \in I,\ f_i(x) \subseteq_2 f_i(y)$ [[ h1, $f_i$ is monotonic ]]

and $\forall\, i \in I,\ f_i(y) \subseteq_2 f(y)$ [[ construction of f ]]

$\therefore$ $\forall\, i \in I,\ f_i(x) \subseteq_2 f(y)$ [[ $\subseteq$ transitive ]]

$\therefore$ $\mathrm{lub}(f_i(x))_{i \in I} \subseteq_2 f(y)$ [[ def. 2.3 ]]

$\therefore$ $f(x) \subseteq_2 f(y)$ [[ construction of f ]]

$\therefore$ f is monotonic.

$[[]]_{\text{monotonic strongly admissible}}$

$\phi(f)$ = "f is continuous" is strongly admissible on $P_2^{P_1}$:

Assume [h2] $(f_i)_{i \in I}$ non-empty chain of continuous functions from $P_1$ to $P_2$ .

We have $f = \lambda x.\mathrm{lub}(f_i(x))_{i \in I} = \mathrm{lub}\,(f_i)_{i \in I}$ [[ construction of lub of function-chains ]]

and we already know that f is monotonic [[ by above proof ]]

Let $(x_j)_{j \in I}$ chain in $P_1$

We have $\forall\, j \in I,\ x_j \subseteq_1 \mathrm{lub}\,(x_j)_{j \in I}$ [[ def. 2.3, LUB => Upper Bound ]]

$\therefore$ $\forall\, j \in I,\ f(x_j) \subseteq_2 f(\mathrm{lub}\,(x_j)_{j \in I})$ [[ f monotonic ]]

$\therefore$ L1: $\mathrm{lub}(f(x_j))_{j \in I} \subseteq_2 f(\mathrm{lub}\,(x_j)_{j \in I})$ [[ def. 2.3 ]]

Let $i \in I$, arbitrary.

We have $f_i \subseteq f$ [[ $f = \mathrm{lub}\,(f_i)_{i \in I}$, LUB => Upper Bound ]]

$\therefore$ $\forall\, j \in I,\ f_i(x_j) \subseteq_2 f(x_j)$ [[ def. 2.23 ]]

and $\forall\, j \in I,\ f(x_j) \subseteq_2 \mathrm{lub}(f(x_j))_{j \in I}$ [[ def. 2.3, LUB => Upper Bound ]]

$\therefore$ $\forall\, j \in I,\ f_i(x_j) \subseteq_2 \mathrm{lub}(f(x_j))_{j \in I}$ [[ $\subseteq$ transitive ]]

$\therefore$ $\mathrm{lub}(f_i(x_j))_{j \in I} \subseteq_2 \mathrm{lub}(f(x_j))_{j \in I}$ [[ def. 2.3 ]]

and $f_i(\mathrm{lub}\,(x_j)_{j \in I}) = \mathrm{lub}(f_i(x_j))_{j \in I}$ [[ h2, $f_i$ continuous ]]

$\therefore$ $f_i(\mathrm{lub}\,(x_j)_{j \in I}) \subseteq_2 \mathrm{lub}(f(x_j))_{j \in I}$

and this was done for arbitrary i,

$\therefore$ $\forall\, i \in I,\ f_i(\mathrm{lub}\,(x_j)_{j \in I}) \subseteq_2 \mathrm{lub}(f(x_j))_{j \in I}$

$\therefore$ $\mathrm{lub}(f_i(\mathrm{lub}\,(x_j)_{j \in I}))_{i \in I} \subseteq_2 \mathrm{lub}(f(x_j))_{j \in I}$ [[ def. 2.3 ]]

and $f(\mathrm{lub}\,(x_j)_{j \in I}) = \mathrm{lub}(f_i(\mathrm{lub}\,(x_j)_{j \in I}))_{i \in I}$ [[ construction of f ]]

$\therefore$ L2: $f(\mathrm{lub}\,(x_j)_{j \in I}) \subseteq_2 \mathrm{lub}(f(x_j))_{j \in I}$

$\therefore$ $f(\mathrm{lub}\,(x_j)_{j \in I}) = \mathrm{lub}(f(x_j))_{j \in I}$ [[ lines L1 and L2 ]]

$\therefore$ f is continuous.

$[[]]_{\text{continuous strongly admissible}}$

$[[]]_{\text{Thm. 2.27}}$

Other strongly admissible functional predicates will appear in the next sections.

This completes our list of (slightly extended) standard notions. We now concentrate on particular classes of domains which will be of essential use later.

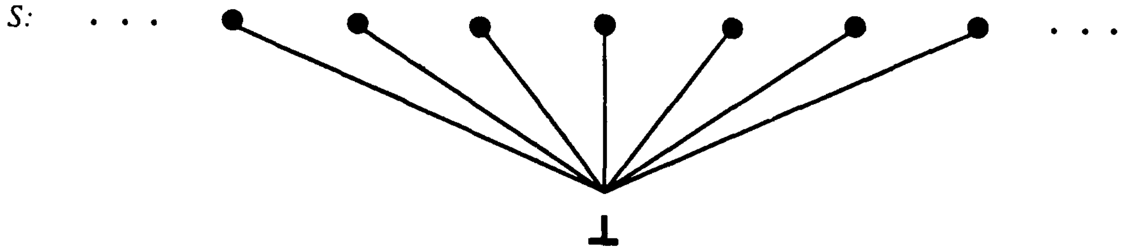
## 2.2. Finite Depth domains

**Definition 2.28: Flat domain**
Let $S$ be an arbitrary set, $S_\perp$ (read "$S$ lifted", or "$S$ bottom") is the PCPO obtained by adding an extra element: $\perp$, and the binary relation: $\subseteq$ defined by: $\forall\, x,y \in S$, $x \subseteq y \iff x = \perp \;\vee\; x = y$.

It is immediate that $\subseteq$ is reflexive, antisymmetric and transitive, and that all $\subseteq$-chains have a lub.

A picture of $S_\perp$ is most convincing:

**Figure 2-1:** Flat domain



Syntactic note about $\perp$ : the character "$\perp$ " has no magical properties! In a different context (such as chapter 3), we will free to use a different "least element" character more appropriate for that context.

An essential property of flat domains is that all chains of distinct elements are *finite*, in fact they are at most of length 2. Many properties of flat domains (such as can be found in [Manna 74], chapter 5) generalize, often more clearly, to arbitrary CPOs which have this "finite depth" property.

Moreover, the domain on which we will base our semantics for synchronous circuits is a finite depth domain. We have therefore isolated this property here, as well as its consequences, so as to distinguish the abstract properties of these domains from the idiosynchrasies of their application to the semantics of synchronous circuits.

**Definition 2.29: Finite Depth domain [FD-CPO]**
Let $<P, \subseteq>$ be a CPO, $<P, \subseteq>$ is of Finite Depth $\iff$ any chain in $P$ is a finite set.

An equivalent way of characterizing FD-CPOs is the "Accumulation" property:

**Theorem 2.30: Accumulation**
Let $<P, \subseteq>$ be a CPO, $<P, \subseteq>$ FD-CPO $\iff \forall\, (x_i)_{i \in I}$ non-empty chain in $P$, $\exists\, i_0 \in \omega \mid \forall\, i \geq i_0,\ x_i = x_{i_0}$ (and therefore also: $\mathrm{lub}(x_i) = x_{i_0}$ ).

In other words, there is a *finite* index, after which the chain is constant. We refer to $i_0$ as the "accumulation point" and $x_{i_0}$ as the "accumulation value" (or "lub").

**Prnof:**
(Should be intuitively clear, given for completeness.)

=>

Assume [h1] $<P, \subseteq>$ FD-CPO, [h2] $(x_i)_{i \in I}$ arbitrary non-empty chain in ***P***, we prove the Accumulation property by contradiction:

Assume that it is false, we have: $\forall i \in \omega, \exists j_i \geq i \mid x_i \subsetneq x_{j_i} \wedge x_i \neq x_{j_i}$

then we extract $X = (x_{j_i})_{i \in \omega}$, which is a chain [[ h2, and subset of a chain is a chain ]]

and X contains an infinite number of (distinct) elements [[ by construction ]]

$\therefore$ X is an infinite chain in ***P***, contradicting h1.

[[]] $_{=>}$

<=

Assume [h1] Accumulation property holds, [h2] $(x_i)_{i \in I}$ arbitrary chain.

We have if $(x_i)_{i \in I}$ is empty, then it is finite [[ trivially ]]

and if $(x_i)_{i \in I}$ is not empty

then $\exists i_0 \in \omega \mid \forall i \geq i_0, x_i = \cdot$ [[ h1 , h2 ]]

$\therefore$ $(x_i)_{i \in I} = \{ (x_i), i = 0 .. i_0 \}$ [[ set extension! ]]

$\therefore$ $(x_i)_{i \in I}$ is a finite set.

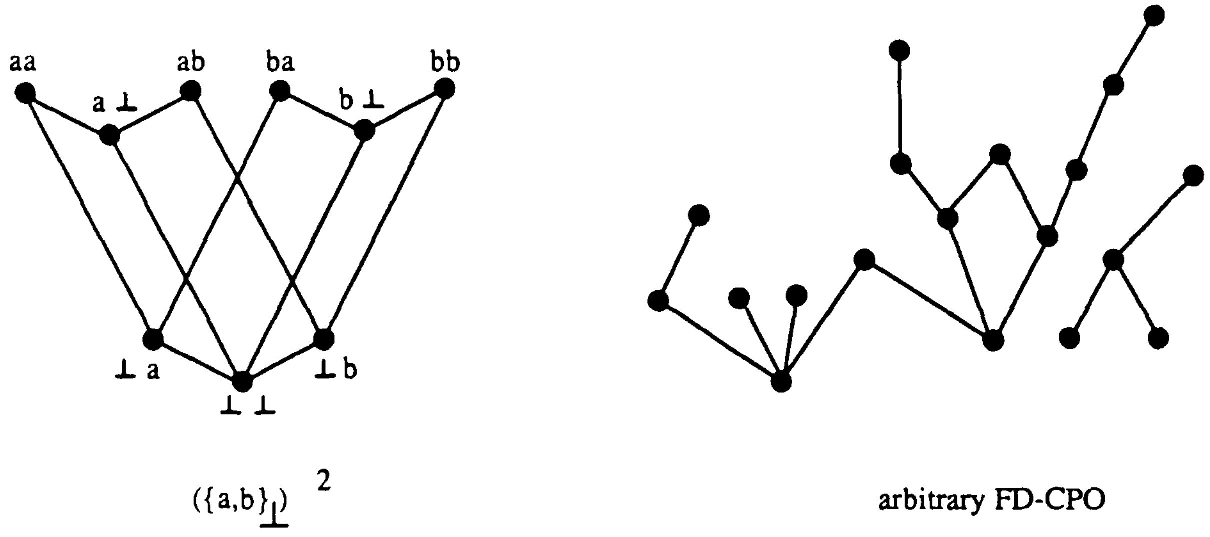
and this was done for an arbitrary chain, so ***P*** is a FD-CPO.

[[]] $_{<=}$

[[]]$_{\text{Thm. 2.30}}$

A few pictorial examples may help:

**Figure 2-2:** Finite depth CPOs



$(\{a,b\}_\perp)^2$

arbitrary FD-CPO

Examples of FD-CPOs abound: It is obvious that any finite CPO is a FD-CPO (and any finite PO is a CPO). It is also clear that FD-CPOs can be obtained as follows.

**Theorem 2.31: Flat domains are FD-CPOs.**

**Proof:**

Immediate.

[[]]$_{\text{Thm. 2.31}}$

**Theorem 2.32: Product of FD-CPOs**

The Cartesian product of FD-CPOs is a FD-CPO.

**Proof:**

Immediate with the Accumulation property, by taking the max of the accumulation points for each coordinate.

[[]]$_{\text{Thm. 2.32}}$

**Theorem 2.33: Disjoint union of FD-CPOs**

The disjoint union of FD-CPOs is a FD-CPO.

**Proof:**

Immediate once you notice that any chain in the disjoint union is necessarily included in one of the original sets.

[[]]$_{\text{Thm. 2.33}}$

Finite Depth has interesting consequences regarding continuity issues, both for functions and functionals:

**Theorem 2.34: Monotonic => Continuous in FD-CPOs**

Let $<\boldsymbol{P}_1, \subseteq_1>$ , $<\boldsymbol{P}_2, \subseteq_2>$ be FD-CPOs, f a function from $\boldsymbol{P}_1$ to $\boldsymbol{P}_2$ , f monotonic => f continuous .

**Proof:**

Should be intuitively clear. Given here for completeness.

Assume [h1] $<\boldsymbol{P}_1, \subseteq_1>$ FD-CPO, [h2] $<\boldsymbol{P}_2, \subseteq_2>$ FD-CPO, [h3] f a monotonic function: $\boldsymbol{P}_1 \rightarrow \boldsymbol{P}_2$ , [h4] $(x_i)_{i \in I}$ non-empty chain in $\boldsymbol{P}_1$ .

We have $\exists\, i_0 \in \omega \mid \forall\, i \geq i_0 ,\, x_i = x_{i_0} = \text{lub}\ (x_i)_{i \in I}$ [[ h1, thm. 2.30 ]]

We have $f(x_i)_{,\, i \in I}$ non-empty chain in $\boldsymbol{P}_2$, [[ h3 and h4 ]]

$\therefore\quad \exists\, i_1 \in \omega \mid \forall\, i \geq i_1 ,\, f(x_i) = f(x_{i_1}) = \text{lub}(f(x_i))_{i \in I}$ [[ h2, thm. 2.30 ]]

Let $j = \max(i_0, i_1)$

We have $x_j = \text{lub}\ (x_i)_{i \in I} \ \wedge\ f(x_j) = \text{lub}(f(x_i))_{i \in I}$

$\therefore\quad f(\text{lub}\ (x_i)_{i \in I}) = \text{lub}(f(x_i))_{i \in I}$

$\therefore$ f is continuous.

[[]]$_{\text{Thm. 2.34}}$

Our result about functionals is a generalization of [Manna 74] theorem 5.1 , which states that functionals (on monotonic functions, of arity n) on a flat domain, defined by composition of monotonic functions (of arity n) and a function variable "F", are continuous.

Besides separating what is true in any CPO from what depends essentially on the finite depth property, we generalize the result in three ways:

- To apply to FD-CPOs instead of just flat domains,
- To allow functions of any arity in the construction of the functional, as long as arities match. This

technicality corrects the fact that the theorem as stated by Manna does not even apply to the functional defining "factorial". .

- To apply to functionals on any *sub-cpo* of the set of monotonic functions (another technicality which we will require in order to apply this result for our purposes in the next section).

The first theorem applies to any CPO, independently of finite depth considerations:

**Theorem 2.35: Continuous functionals on a CPO**
Let $<P, \subseteq>$ be a CPO, if $\tau$ is a functional, on *continuous* functions: ( $P^n \rightarrow P$ ) defined by (arity-correct) composition of *continuous* functions: ( $P^m \rightarrow P$ ) for any $m \in \omega$, and the function variable "F", then $\tau$ is continuous.

Our proof is similar in structure (induction cases) to [Manna 74]'s (partial) proof in the flat domain case, but different in detail since we do not mingle considerations of "finite-depth" (accumulation property).

**Proof:**

The proof is by structural induction on $\tau$. There are 4 cases. In each case we have to check that
$\tau$ is closed (i.e. yields continuous functions when fed a continuous function as input),
$\tau$ is monotonic,
$\tau$ preserves lubs of function-chains.

[Base] case 1: $\tau = \lambda F.g$ , with g continuous function: $P^n \rightarrow P$ .
$\tau$ closed: immediate.
$\tau$ monotonic: immediate    [[ constant fun. (in any PO) is monotonic ]]
$\tau$ preserves lubs of function-chains: immediate    [[ constant fun. (in any CPO) is continuous ]]

[[]]$_{\text{case 1}}$

[Base] case 2: $\tau = \lambda F.F$ .
$\tau$ closed: immediate    [[ Identity is always closed on any set! ]]
$\tau$ monotonic: immediate    [[ Identity (in any PO) is monotonic ]]
$\tau$ preserves lubs of function-chains: immediate    [[ Identity (in any CPO) is continuous ]]

[[]]$_{\text{case 2}}$

[Induction] case 3: $\tau = \lambda F.g_\circ(\tau_1(F),..,\tau_m(F))$ , with g continuous function: $P^m \rightarrow P$ .
$\tau$ closed: immediate    [[ thm. 2.11, induction hyp. on $\tau_1..\tau_m$ ]]
$\tau$ monotonic:
Let $f_1, f_2$ continuous functions: $P^n \rightarrow P \mid f_1 \subseteq f_2$
We have $\forall j \in \{1..m\}, \tau_j(f_1) \subseteq \tau_j(f_2)$    [[ $\tau_j$ monotonic, induction hyp. ]]
$\therefore \quad \forall x \in P^n, \forall j \in \{1..m\}, (\tau_j(f_1))(x) \subseteq (\tau_j(f_2))(x)$    [[ def. 2.23 ]]
$\therefore \quad \forall x \in P^n, g[(\tau_1(f_1))(x),..,(\tau_m(f_1))(x)] \subseteq g[(\tau_1(f_2))(x),..,(\tau_m(f_2))(x)]$
[[ g monotonic, thm. 2.9 ]]
$\therefore \quad \tau(f_1) \subseteq \tau(f_2)$    [[ def. 2.23, definition of $\tau$ ]]
$\tau$ preserves lubs of function-chains:
Let $(f_i)_{i \in I}$ non-empty chain of continuous functions: $P^n \rightarrow P$ .
We have $\forall j \in \{1..m\}, \tau_j(\text{lub}\,(f_i)_{i \in I}) = \text{lub}[\tau_j(f_i)]_{i \in I}$    [[ $\tau_j$ continuous, induction hyp. ]]
$\therefore$ L1: $\forall x \in P^n, \forall j \in \{1..m\}, (\tau_j(\text{lub}\,(f_i)_{i \in I}))(x) = \text{lub}[(\tau_j(f_i))(x)]_{i \in I}$
[[ construction of lub of function-chains ]]

Let $x \in P^n$, arbitrary.
We have $(\tau(\text{lub}\,(f_i)_{i \in I}))(x) = g((\tau_1(\text{lub}\,(f_i)_{i \in I}))(x),..,(\tau_m(\text{lub}\,(f_i)_{i \in I}))(x))$

[[ definition of τ ]]

$\ldots = g(\mathrm{lub}[(\tau_1(f_i))(x)]_{i \in I}, \ldots, \mathrm{lub}[(\tau_m(f_i))(x)]_{i \in I})$ [[ line L1 ]]

$\ldots = \mathrm{lub}[g((\tau_1(f_i))(x), \ldots, (\tau_m(f_i))(x))]_{i \in I}$ [[ g continuous ]]

$\ldots = \mathrm{lub}[(\tau(f_i))(x)]_{i \in I}$ [[ definition of τ ]]

$\ldots = (\mathrm{lub}[\tau(f_i)]_{i \in I})(x)$ [[ construction of lub of function-chains ]]

and this was done for arbitrary x,

$\therefore \quad \tau(\mathrm{lub}\,(f_i)_{i \in I}) = \mathrm{lub}[\tau(f_i)]_{i \in I}$

$[[]]_{\text{case 3}}$

[Induction] case 4: $\tau = \lambda F.F_\circ(\tau_1(F), \ldots, \tau_n(F))$ .

τ closed: immediate [[ thm. 2.11, induction hyp. on $\tau_1 .. \tau_n$ ]]

τ monotonic:

Let $f_1, f_2$ continuous functions on $\boldsymbol{P}^n \mid f_1 \subseteq f_2$

We have $\forall j \in \{1..n\}, \tau_j(f_1) \subseteq \tau_j(f_2)$ [[ $\tau_j$ monotonic, induction hyp. ]]

$\therefore \quad \forall x \in \boldsymbol{P}^n, \forall j \in \{1..n\}, (\tau_j(f_1))(x) \subseteq (\tau_j(f_2))(x)$ [[ def. 2.23 ]]

$\therefore \quad \forall x \in \boldsymbol{P}^n, f_2[(\tau_1(f_1))(x), \ldots, (\tau_n(f_1))(x)] \subseteq f_2[(\tau_1(f_2))(x), \ldots, (\tau_n(f_2))(x)]$

[[ $f_2$ monotonic, thm. 2.9 ]]

and $\forall x \in \boldsymbol{P}^n, f_1[(\tau_1(f_1))(x), \ldots, (\tau_n(f_1))(x)] \subseteq f_2[(\tau_1(f_1))(x), \ldots, (\tau_n(f_1))(x)]$

[[ $f_1 \subseteq f_2$ ]]

$\therefore \quad \forall x \in \boldsymbol{P}^n, f_1[(\tau_1(f_1))(x), \ldots, (\tau_n(f_1))(x)] \subseteq f_2[(\tau_1(f_2))(x), \ldots, (\tau_n(f_2))(x)]$

[[ ⊆ transitive ]]

$\therefore \quad \tau(f_1) \subseteq \tau(f_2)$ [[ def. 2.23, definition of τ ]]

τ preserves lubs of function-chains:

Let $(f_i)_{i \in I}$ non-empty chain of continuous functions on $\boldsymbol{P}^n$.

We have $\forall j \in \{1..n\}, \tau_j(\mathrm{lub}\,(f_i)_{i \in I}) = \mathrm{lub}[\tau_j(f_i)]_{i \in I}$ [[ $\tau_j$ continuous, induction hyp. ]]

$\therefore$ L2: $\forall x \in \boldsymbol{P}^n, \forall j \in \{1..n\}, (\tau_j(\mathrm{lub}\,(f_i)_{i \in I}))(x) = \mathrm{lub}[(\tau_j(f_i))(x)]_{i \in I}$

[[ construction of lub of function-chains ]]

Let $x \in \boldsymbol{P}^n$, arbitrary.

We have $(\tau(\mathrm{lub}\,(f_i)_{i \in I}))(x) = (\mathrm{lub}\,(f_i)_{i \in I})((\tau_1(\mathrm{lub}\,(f_i)_{i \in I}))(x), \ldots, (\tau_n(\mathrm{lub}\,(f_i)_{i \in I}))(x))$

[[ definition of τ ]]

$\ldots = \mathrm{lub}\{f_i((\tau_1(\mathrm{lub}\,(f_i)_{i \in I}))(x), \ldots, (\tau_n(\mathrm{lub}\,(f_i)_{i \in I}))(x))\}_{i \in I}$ [[ construction of lub of function-chains ]]

$\ldots = \mathrm{lub}\{f_i(\mathrm{lub}[(\tau_1(f_i))(x)]_{i \in I}, \ldots, \mathrm{lub}[(\tau_n(f_i))(x)]_{i \in I})\}_{i \in I}$ [[ line L2 ]]

$\ldots = \mathrm{lub}\{\mathrm{lub}[f_i((\tau_1(f_i))(x), \ldots, (\tau_n(f_i))(x))]_{i \in I}\}_{i \in I}$ [[ $f_i$ continuous ]]

$\ldots = \mathrm{lub}[f_i((\tau_1(f_i))(x), \ldots, (\tau_n(f_i))(x))]_{i \in I}$ [[ $\mathrm{lub}_{i \in I}(\mathrm{lub}_{i \in I}(.)) = \mathrm{lub}_{i \in I}(.)$ ]]

$\ldots = \mathrm{lub}[(\tau(f_i))(x)]_{i \in I}$ [[ definition of τ ]]

$\ldots = (\mathrm{lub}[\tau(f_i)]_{i \in I})(x)$ [[ construction of lub of function-chains ]]

and this was done for arbitrary x,

$\therefore \quad \tau(\mathrm{lub}\,(f_i)_{i \in I}) = \mathrm{lub}[\tau(f_i)]_{i \in I}$

$[[]]_{\text{case 4}}$

$[[]]_{\text{Thm. 2.35}}$

Combining thm. 2.34 and thm. 2.35, we immediately get the result for Finite Depth CPOs:

**Theorem 2.36: Continuous functionals on a FD-CPO**

Let $<\boldsymbol{P}, \subseteq>$ be a FD-CPO, if τ is a functional, on *monotonic* functions: $[\boldsymbol{P}^n \rightarrow \boldsymbol{P}]$, defined by composition of *monotonic* functions: $[\boldsymbol{P}^m \rightarrow \boldsymbol{P}]$ for any $m \in \omega$, and the function variable "F", then τ is continuous.

And finally, noting that the proof of thm. 2.35 carries through to functionals defined on a sub-cpo of the set of monotonic functions, as long as we assume that they are closed on that sub-cpo, we get our final result:

**Theorem 2.37: Continuous functionals on a FD-CPO , general version**
Let $<\boldsymbol{P}, \subseteq>$ be a FD-CPO, if $\tau$ is a functional, on any *sub-cpo* of the set of monotonic functions: $[\, \boldsymbol{P}^{n} \rightarrow \boldsymbol{P} \,]$, closed on that sub-cpo, defined by composition of *monotonic* functions: $[\, \boldsymbol{P}^{m} \rightarrow \boldsymbol{P} \,]$ for any $m \in \omega$, and the function variable "F", then $\tau$ is continuous.

$[[]]_{\text{Generalization of [Manna 74] Thm 5.1}}$

Note that this theorem (or thm. 2.36) are not true in arbitrary CPOs, as the following simple counterexample shows:

**Counter-example:**

Let $\boldsymbol{P} = \omega{+}1$, with the standard (ordinal order) $\leq$ , $\boldsymbol{P}$ is a CPO.

Let $g = \lambda x.(\text{if } x = \omega \text{ then } 1 \text{ else } 0)$

We have g monotonic [[ immediate verification ]]

Let $\tau = \lambda F.g{\circ}F$ , $\tau$ is a functional defined by composition of monotonic functions and the function variable "F".

Let $f_i = \lambda x.i$ (i.e. the constant function: i), $\forall\, i \in \omega$ .

We have $\forall\, i \in \omega$ , $f_i$ is monotonic [[ constant functions are monotonic ]]

and $\forall\, i \in \omega$ , $f_i \leq f_{i+1}$ , i.e. $(f_i)_{i \in \omega}$ chain [[ immediate ]]

and $\text{lub}\,(f_i)_{i \in \omega} = \lambda x.\omega$ [[ immediate verification ]]

$\therefore\quad \tau(\text{lub}\,(f_i)_{i \in \omega}) = \lambda x.1$

We have $\forall\, i \in \omega$ , $\tau(f_i) = \lambda x.0$

$\therefore\quad \text{lub}(\tau(f_i))_{i \in \omega} = \lambda x.0$

$\therefore\quad \tau(\text{lub}\,(f_i)_{i \in \omega}) \neq \text{lub}(\tau(f_i))_{i \in \omega}$

$[[]]_{\textit{counter-example}}$

## 2.3. Strings of a domain, and String Induction Algebra

A particular construction on domains which we have found useful in our semantics is the domain of (finite) Strings on a domain. It is also from these domains that we noticed the generalizations from flat domain to finite depth domain.

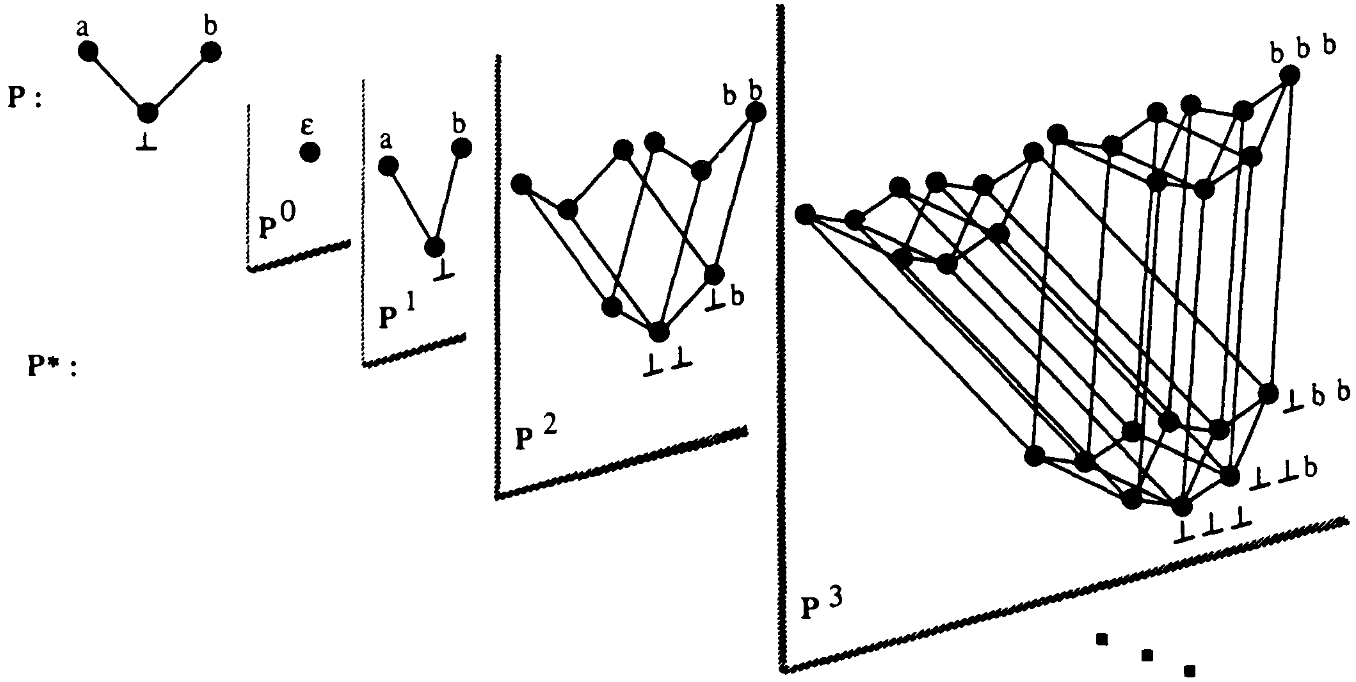
As in the previous section, we study the properties of String domains independently of their application to the semantics of synchronous circuits so as to separate the general from the particular. (This also has the advantage of keeping the overall notation, and hence proofs, simpler.)

**Definition 2.38: Strings of a partial order**
Let $<\boldsymbol{P}, \subseteq>$ be a PO, $\boldsymbol{P}^{*} = \cup\, (\boldsymbol{P}^{i})_{i \in \omega}$, with the *induced* ordering, is a PO (disjoint union of cartesian products of a PO). We call it: Strings of $\boldsymbol{P}$.

Recall that when forming the disjoint union we are *not* adding any new elements (cf thm. 2.19).

Once again, a picture helps.

Figure 2-3: Strings on a flat domain



They key fact about the String construction is that it preserves the "niceness" of the underlying domain, to a great extent:

**Theorem 2.39: Strings on a CPO**
$<P, \subseteq>$ is a CPO => $<P^*, \subseteq>$ is a CPO.

**Proof:**
Immediate by thm. 2.17 and thm. 2.19.

$[[]]_{\text{Thm. 2.39}}$

and most importantly:

**Theorem 2.40: Strings on a FD-CPO**
$<P, \subseteq>$ is a FD-CPO => $<P^*, \subseteq>$ is a FD-CPO.

**Proof:**
Immediate by thm. 2.32 and thm. 2.33.

$[[]]_{\text{Thm. 2.40}}$

Note however that the String construction does *not* preserve "pointedness" (i.e. PCPO). In fact, we have a stronger statement to the contrary:

**Theorem 2.41: Strings do not have a least element**

Let $<\boldsymbol{P}, \subseteq>$ be a PO, $\boldsymbol{P}$ non-empty $\;=>\;$ $<\boldsymbol{P}^*, \subseteq>$ has no least element.

**Proof:**

Assume [h1] $<\boldsymbol{P}, \subseteq>$ PO, [h2] $\boldsymbol{P}$ non-empty

Let $\varepsilon$ be the empty string ( $\in \boldsymbol{P}^*$ )

We have $\forall x \in \boldsymbol{P}^*$, [c1]( $x \subseteq \varepsilon \;=>\; x=\varepsilon$ ) $\wedge$ [c2]( $\varepsilon \subseteq x \;=>\; x=\varepsilon$ )

[[ $\subseteq$ is induced coordinatewise ordering]]

Let $a \in \boldsymbol{P}$ [[ h2 ]]

We have $a \in \boldsymbol{P}^*$ (string of length 1, containing the element a)

Assume $\perp$ least element of $\boldsymbol{P}^*$

then $\perp \subseteq \varepsilon$ and $\perp \subseteq a$

$\therefore$ $\perp = \varepsilon$ [[ c1 and $\perp \subseteq \varepsilon$ ]]

$\therefore$ $\varepsilon \subseteq a$ [[ $\perp \subseteq a$ ]]

$\therefore$ $\varepsilon = a$, which is a contradiction. [[ c2 ]]

$[[]]_{\text{Thm. 2.41}}$

This point was mostly made to bring out the fact that we are not studying the "usual" domain of strings under the prefix ordering (for which ε is a least element), instead we are constructing the String domain of an arbitrary PO, under the *induced ordering*.

The junction with "usual" strings will now be made, but the preceding remark will still be valid for the rest of this work.

We consider the usual (slightly extended) string structure on $\boldsymbol{P}^*$:
$<\boldsymbol{P}^*, \varepsilon, ., |\,|, \leq, ., \text{last}(), \text{abl}(), \text{1st}(), \text{rst}(), \uparrow, \downarrow, \Theta>$

**Definition 2.42: String structure**

- $\varepsilon : \rightarrow \boldsymbol{P}^*$, (constructor) empty string.
- $.$ : Add : $\boldsymbol{P}^* \times \boldsymbol{P} \rightarrow \boldsymbol{P}^*$, (constructor) add a character (to the right).
- $|\,|$ : Length : $\boldsymbol{P}^* \rightarrow \omega$, length of a string. (We assume the integers are included in $\boldsymbol{P}$, or are encodable in it, cf. [Moschovakis 71].)
  Defined by: ( $|\varepsilon| = 0$ ) $\wedge$ ( $|x.u| = |x| + 1$ ) .
- $\leq$ : Prefix : $\boldsymbol{P}^* \times \boldsymbol{P}^* \rightarrow \{T,F\}$, prefix relation on strings.
  Defined by: ( $x \leq \varepsilon \;<=>\; x = \varepsilon$ ) $\wedge$ ( $x \leq y.u \;<=>\; x = y.u \;\vee\; x \leq y$ ) .
- $.$ : Concatenate : $\boldsymbol{P}^* \times \boldsymbol{P}^* \rightarrow \boldsymbol{P}^*$, concatenate two strings. We overload the "." symbol since we will identify characters and strings of length 1. We will also sometimes omit the "." all together, when no confusion can result.
  Defined by: ( $x . \varepsilon = x$ ) $\wedge$ ( $x . (y.u) = (x . y).u$ , where the "." preceding "u" means "Add" ) .
- last() : Last : $\boldsymbol{P}^* \rightarrow \boldsymbol{P}$ (destructor, partial) , last character of a string.
  Defined by: $\text{last}(x.u) = u$ .
- abl() : All-But-Last : $\boldsymbol{P}^* \rightarrow \boldsymbol{P}^*$ (destructor, partial) , all characters of a string but the last one.
  Defined by: $\text{abl}(x.u) = x$ .
- 1st() : First : $\boldsymbol{P}^* \rightarrow \boldsymbol{P}$ (derived destructor, partial) , first character of a string.
  Defined by: $\text{1st}(u.x) = u$ .
- rst() : Rest : $\boldsymbol{P}^* \rightarrow \boldsymbol{P}^*$ (derived destructor, partial) , all characters of a string but the first one.
  Defined by: $\text{rst}(u.x) = x$ .
- $\uparrow$ : "To the power" : $\boldsymbol{P} \times \omega \rightarrow \boldsymbol{P}^*$, make a string by Adding the same character a certain number of

time.
Defined by: $u{\uparrow}^{n} = uu..u$ "n times", or formally: $( u{\uparrow}^{0} = \varepsilon ) \wedge (u{\uparrow}^{n+1} = u{\uparrow}^{n} .u )$ .

- $\downarrow$ : "At index/position" : $\boldsymbol{P}^* \times \omega \rightarrow \boldsymbol{P}$ , extract a character from a string .
  Defined by: Let $n = |x|$ , $x = x{\downarrow}_1 x{\downarrow}_2 ... x{\downarrow}_n$ . We also use $\downarrow$ with 2 arguments to extract substrings: $x{\downarrow}_{i..j}$ denotes the corresponding substring of x if $i \leq j \leq n$ , $\varepsilon$ otherwise. $(x = x{\downarrow}_{1..n})$ . The formal (recursive) definition is messy and uninteresting.

- $\Theta$ : $\Theta$ is to "." (add) in string theory, what $\Sigma$ is to "+" and what $\Pi$ is to " $\times$ " in number theory, i.e. $\Theta_{i=1}^{n} u_i = u_1 u_2 .. u_n$, where $u_i$ is any character expression.
  Formally: $( \Theta_{i=1}^{0} u_i = \varepsilon ) \wedge ( \Theta_{i=1}^{n+1} u_i = (\Theta_{i=1}^{n} u_i) . u_{n+1} )$ .

We also allow ourselves to expand this structure with additional (derived) operations whenever needed.

**Terminology notes:**

There are a few basic string operations which are well-known in the literature: [Landin 65], [Burge 75], [Friedman-Wise 76] and [Manna-Waldinger 85] among many others. However, there are no consistent notations. We have therefore used our own, which we have tried to keep simple, and meaningful relative to the use we will have for them (describing synchronous system semantics).

The notation used for subscripting is taken from [Mason 86] and [Talcott 85]. Even though it is "heavier" than simple subscripting, it allows subscripted string variables by differentiating between x, $x_1$ (strings) and $x{\downarrow}_1$, $x_1{\downarrow}_1$ (characters). [Note: if no confusion can result, i.e. in a context where no subscripted string names are used, then it is reasonable to omit the arrow.]

**Theorem 2.43: Prefix**
There is an equivalent definition of the Prefix relation which we will sometime use: $\forall x,y \in \boldsymbol{P}^* , x \leq y \iff \exists z \in \boldsymbol{P}^* \mid y = x.z$ .

***Proof:***
Immediate induction.

$[[]]_{\text{Thm. 2.43}}$

We now study various function domains on string-CPOs:

Let $<\boldsymbol{P}_1^*, \subseteq_1>$ , $<\boldsymbol{P}_2^*, \subseteq_2>$ be string-CPOs, it is immediate from thm. 2.24 and thm. 2.27 that:

- $\boldsymbol{P}_2^{*\boldsymbol{P}_1^*}$ : all functions from $\boldsymbol{P}_1^*$ to $\boldsymbol{P}_2^*$ ,
- $[ \boldsymbol{P}_1^* \rightarrow \boldsymbol{P}_2^* ]$ : all $\subseteq$-monotonic functions from $\boldsymbol{P}_1^*$ to $\boldsymbol{P}_2^*$ ,
- $( \boldsymbol{P}_1^* \rightarrow \boldsymbol{P}_2^* )$ : all $\subseteq$-continuous functions from $\boldsymbol{P}_1^*$ to $\boldsymbol{P}_2^*$ ,

are CPOs.

There are however other classes of functions which are meaningful only in the string structure, and we are interested in two such classes:

**Definition 2.44: Length-Preserving [LP] function**
Let f be a function: $\boldsymbol{P}_1^* \rightarrow \boldsymbol{P}_2^*$ , f is Length-Preserving [LP] $\iff \forall x \in \boldsymbol{P}_1^* , | f(x) | = | x |$ .

**Definition 2.45: $\leq$-monotonic function**
Let f be a function: $\boldsymbol{P}_1^* \rightarrow \boldsymbol{P}_2^*$ , f is $\leq$-monotonic $\iff \forall x,y \in \boldsymbol{P}_1^* , x \leq y \implies f(x) \leq f(y)$ .

Pronunciation note: $\subseteq$-monotonic can be read "L-monotonic" (short for "less-defined-than-monotonic ). And $\leq$-monotonic can be read "P-monotonic" (for "prefix-monotonic").

**Theorem 2.46: LP preserved by composition**
Let $<\boldsymbol{P}_1{}^*, \subseteq_1>$ , $<\boldsymbol{P}_2{}^*, \subseteq_2>$ and $<\boldsymbol{P}_3{}^*, \subseteq_3>$ be string-CPOs. Let $f : \boldsymbol{P}_1{}^* \rightarrow \boldsymbol{P}_2{}^*$ and $g : \boldsymbol{P}_2{}^* \rightarrow \boldsymbol{P}_3{}^*$ , f and g are LP => $g \circ f : \boldsymbol{P}_1{}^* \rightarrow \boldsymbol{P}_3{}^*$ , is LP .

**Proof:**
Immediate verification.

$[[]]_{\text{Thm. 2.46}}$

**Theorem 2.47: $\leq$-monotonic preserved by composition**
Let $<\boldsymbol{P}_1{}^*, \subseteq_1>$ , $<\boldsymbol{P}_2{}^*, \subseteq_2>$ and $<\boldsymbol{P}_3{}^*, \subseteq_3>$ be string-CPOs. Let $f : \boldsymbol{P}_1{}^* \rightarrow \boldsymbol{P}_2{}^*$ and $g : \boldsymbol{P}_2{}^* \rightarrow \boldsymbol{P}_3{}^*$ , f and g are $\leq$-Monotonic => $g \circ f : \boldsymbol{P}_1{}^* \rightarrow \boldsymbol{P}_3{}^*$ , is $\leq$-Monotonic .

**Proof:**
Immediate verification.

$[[]]_{\text{Thm. 2.47}}$

Both LP and $\leq$-monotonic are in some sense "natural" properties for string of *Finite Depth*-CPOs, as the following theorems indicate.

**Theorem 2.48: LP is strongly admissible on FD-CPOs**
Let $<\boldsymbol{P}_1, \subseteq_1>$ , $<\boldsymbol{P}_2, \subseteq_2>$ be FD-CPOs, " f is LP " is strongly admissible on $\boldsymbol{P}_2{}^{*\boldsymbol{P}_1{}^*}$ .

**Proof:**
Assume [h1] $<\boldsymbol{P}_1, \subseteq_1>$ and $<\boldsymbol{P}_2, \subseteq_2>$ are FD-CPOs, [h2] $(f_i)_{i \in I}$ non-empty chain of LP functions from $\boldsymbol{P}_1{}^*$ to $\boldsymbol{P}_2{}^*$

.

We have $f = \lambda x.\text{lub}(f_i(x))_{i \in I} = \text{lub}\,(f_i)_{i \in I}$ [[ construction of lub of function-chains ]]
Let $x \in \boldsymbol{P}_1{}^*$, arbitrary .
We have $\boldsymbol{P}_2{}^*$ FD-CPO [[ h1, and thm. 2.40 ]]
and $(f_i(x))_{i \in I}$ non-emptychain in $\boldsymbol{P}_2{}^*$ [[ h2 ]]
$\therefore$ $\exists\, i_0 \in \omega \mid \forall\, i \geq i_0 ,\ f\text{-}i(x) = f_{i_0}(x) = \text{lub}(f_i(x))_{i \in I}$ [[ thm. 2.30 ]]
$\therefore$ $f(x) = f_{i_0}(x)$
$\therefore$ $|f(x)| = |f_{i_0}(x)|$
and $|f_{i_0}(x)| = |x|$ [[ $f_{i_0}$ LP, h2 ]]
$\therefore$ $|f(x)| = |x|$
and this was done for arbitrary x,
$\therefore$ f is LP .

$[[]]_{\text{Thm. 2.48}}$

**Theorem 2.49: $\leq$-monotonic is strongly admissible on FD-CPOs**
Let $<\boldsymbol{P}_1, \subseteq_1>$ , $<\boldsymbol{P}_2, \subseteq_2>$ be FD-CPOs, " f is $\leq$-monotonic " is strongly admissible on $\boldsymbol{P}_2{}^{*\boldsymbol{P}_1{}^*}$ .

**Proof:**
Assume [h1] $<\boldsymbol{P}_1, \subseteq_1>$ and $<\boldsymbol{P}_2, \subseteq_2>$ are FD-CPOs, [h2] $(f_i)_{i \in I}$ non-empty chain of $\leq$-monotonic functions from $\boldsymbol{P}_1{}^*$ to $\boldsymbol{P}_2{}^*$ .

We have $f = \lambda x.\mathrm{lub}(f_i(x))_{i \in I} = \mathrm{lub}\,(f_i)_{i \in I}$ [[ construction of lub of function-chains ]]
Let $x,y \in \boldsymbol{P}_1^* \mid [h3]\; x \leq y$ ,
We have $\boldsymbol{P}_2^*$ FD-CPO [[ h1, and thm. 2.40 ]]
and $(f_i(x))_{i \in I}$ , $(f_i(y))_{i \in I}$ non-empty chains in $\boldsymbol{P}_2^*$ [[ h2 ]]
∴ $\exists\, i_0 \in \omega \mid \forall\, i \geq i_0$ , $f\text{-}i(x) = f_{i_0}(x) = \mathrm{lub}(f_i(x))_{i \in I}$ [[ thm. 2.30 ]]
and $\exists\, i_1 \in \omega \mid \forall\, i \geq i_1$ , $f\text{-}i(y) = f_{i_1}(y) = \mathrm{lub}(f_i(y))_{i \in I}$ [[ thm. 2.30 ]]
Let $j = \max(\, i_0 , i_1 \,)$
We have $f(x) = f_j(x)$ and $f(y) = f_j(y)$
and $f_j(x) \leq f_j(y)$ [[ h3, $f_j$ $\leq$-monotonic, h2 ]]
∴ $f(x) \leq f(y)$
∴ f is $\leq$-monotonic .

[[]]$_{\text{Thm. 2.49}}$

It is also obvious that if $\phi_1$ is strongly admissible on $\boldsymbol{P}$ , and $\phi_2$ is strongly admissible on $\boldsymbol{P}$ , then $\phi_1 \wedge \phi_2$ is strongly admissible on $\boldsymbol{P}$.

Therefore we get:

**Theorem 2.50: Function domains on Strings of FD-CPOs**
Let $<\boldsymbol{P}_1, \subseteq_1>$ , $<\boldsymbol{P}_2, \subseteq_2>$ be FD-CPOs, $\boldsymbol{P}_2^{*\boldsymbol{P}_1^*} \cap \phi$, where $\phi$ is any conjunction of

- $\subseteq$-monotonic
- LP
- $\leq$-monotonic

is a CPO, in which the lub of function-chains is unchanged.

**Proof:**
Immediate by thm. 2.22 (sub-CPOs) and thm. 2.27 (for $\subseteq$-monotonic) , thm. 2.48 (for LP) , and thm. 2.49 (for $\leq$-monotonic).

[[]]$_{\text{Thm. 2.50}}$

When trying to extend the notion of Length-Preservation to functions of arity > 1 , we find that the standard cartesian product of string domains is inappropriate. Instead it makes sense to define LP on functions with arguments all of the same length. We therefore define the following product on string domains:

**Definition 2.51: String Cartesian Product**
Let $<\boldsymbol{P}_1^*, \subseteq_1>$ , $<\boldsymbol{P}_2^*, \subseteq_2>$ be string-CPOs, we define their string cartesian product to be: $\boldsymbol{P}_1^* \underline{\times} \boldsymbol{P}_2^* = \{(x,y) \in \boldsymbol{P}_1^* \times \boldsymbol{P}_2^* \mid |x| = |y|\}$ , with the standard (induced) coordinate-wise ordering.

One way to think about this product is: $\boldsymbol{P}_1^* \underline{\times} \boldsymbol{P}_2^* \approx (\boldsymbol{P}_1 \times \boldsymbol{P}_2)^*$ , up to transformations from tuples of strings to strings of tuples and vice-versa. Also, our definition is meaningful in the category of string-domains, as it does not refer to the domains underlying the strings.

Notation: $\boldsymbol{P}^{\underline{n}} = \boldsymbol{P} \underline{\times} \ldots \underline{\times} \boldsymbol{P}$ , n times. And if x denotes an element of $\boldsymbol{P}$ , then $\underline{x}$ will denote an element of $\boldsymbol{P}^{\underline{n}}$ ; the underline, instead of the usual overline, is intended to recall that $\underline{x}$ is a tuple of elements *of equal length*.

We can then immediately generalize the notions of Length-Preservation, $\leq$-monotonicity and $\subseteq$-monotonicity to functions: $\boldsymbol{P}_1^* \underline{\times} \ldots \underline{\times} \boldsymbol{P}_n^* \rightarrow \boldsymbol{P}_0^*$ . thm. 2.50 also immediately generalizes to such functions.

For our purposes in giving semantics to synchronous circuits, we are interested in functions (of various arities) on

$\boldsymbol{P}^*$ which are $\subseteq$-monotonic, $\leq$-monotonic and Length-Preserving and defined by recursive systems of continuous functionals on them. We therefore develop here the String Induction Algebra of a domain $\boldsymbol{P}$ :

**Definition 2.52:** $\boldsymbol{MLP_{P,n}}$

Let $<P, \subseteq>$ be a FD-CPO . $\boldsymbol{MLP_{P,n}}$ is the subset of the set of functions from $\boldsymbol{P}^{*\underline{n}}$ to $\boldsymbol{P}^*$ defined by: $\boldsymbol{MLP_{P,n}} =$ $\boldsymbol{P}^{*P^{*\underline{n}}} \cap (\subseteq\text{-monotonic} \;\wedge\; \leq\text{-monotonic} \;\wedge\;$ Length-Preserving) , together with the standard (induced) pointwise function ordering.

It is an immediate application of Thm. 2.50 that $\boldsymbol{MLP_{P,n}}$ is a CPO, and is a "nice" sub-cpo of the set of monotonic functions. However, by combining all 3 properties, we now get an additional property: Even if $\boldsymbol{P}$ has a least element, $\boldsymbol{P}^{*P^{*\underline{n}}}$ does not have a least element (because no string is less than all others according to the pointwise ordering). However, if $\boldsymbol{P}$ has a least element, then so does $\boldsymbol{MLP_{P,n}}$, as is shown below.

**Theorem 2.53:** $\boldsymbol{MLP_{P,n}}$ **is a PCPO**

Let $<P, \subseteq>$ be a FD-PCPO . $\boldsymbol{MLP_{P,n}}$ is a PCPO with least element: $Q = \lambda\, \underline{x} . \perp \uparrow^{|\underline{x}|}$ , and is a sub-cpo of the set of monotonic functions: $[\, \boldsymbol{P}^n \rightarrow \boldsymbol{P}\, ]$, in which the lub of function-chains is unchanged.

**Proof:**

Let $F \in \boldsymbol{MLP_{P,n}}$ . $\underline{x} \in \boldsymbol{P}^{*\underline{n}}$ arbitrary, let $k = |\underline{x}|$ .

We have $F(\underline{x}) = y\downarrow_{1..k}$ [[ F is LP ]]

and $Q(\underline{x}) = \perp\uparrow^{k}$ [[ definition of Q ]]

$\therefore$ $\forall i \in \{1..k\} , \perp \subseteq y\downarrow_i$ [[ definition of $\perp$ ! ]]

$\therefore$ $\forall i \in \{1..k\} , Q(\underline{x})\downarrow_i \subseteq F(\underline{x})\downarrow_i$

$\therefore$ $Q(\underline{x}) \subseteq F(\underline{x})$ [[ definition of order on strings ]]

and this was done for arbitrary $\underline{x}$ and F,

$\therefore$ Q is least element.

$[[]]_{\text{Thm. 2.53}}$

We can now construct our string induction algebra:

**Theorem 2.54:** $\boldsymbol{MLP_P}$ **Continuous String Induction Algebra**

Let $<P, \subseteq>$ be a FD-PCPO, and let $(F_i)_{i \in I}$ be functions in $\boldsymbol{MLP_{P,n_i}}$ .

Let $\boldsymbol{MLP_P} = < (\boldsymbol{MLP_{P,n}})_{n \in \omega} , \boldsymbol{F}\, [(F_i)_{i \in I}] >$ where $\boldsymbol{F}\, [(F_i)_{i \in I}]$ is the least set of functionals containing:

- the functionals $F_{i\circ} = \lambda\, f.\, F_i \circ f$ , for $i \in I$ . (Or $\lambda\, f_1,\dots,f_{n_i} . (\lambda\, \underline{x} . F_i(f_1(\underline{x}),\dots,f_{n_i}(\underline{x}))$ in the general case.)
- the identity functionals,

and closed under composition with projections, then:

$\boldsymbol{MLP_P}$ is an induction algebra (cf. def. 2.15) and all functionals in $\boldsymbol{F}$ are continuous.

**Proof:**

Domain requirement:

We have $\forall n \in \omega , \boldsymbol{MLP_{P,n}}$ is a PCPO. [[ thm. 2.53 ]]

$[[]]_{\text{domain req.}}$

We still have to prove that all the functionals in $\boldsymbol{F}$ are closed (i.e. really yield a function in $\boldsymbol{MLP_{P,n}}$ for some n) and are continuous.

Closed:

We have $\forall i \in I , F_i \in \boldsymbol{MLP_{P,n_i}}$ . [[ hypothesis ]]

| | | |
|---|---|---|
| and | $\subseteq$-monotonicity, $\leq$-monotonicity and LP are preserved by composition | |
| | | [[ thm. 2.10. thm. 2.47 and thm. 2.46 ]] |
| ∴ | $\forall\, i \in I$. $F_{i^\circ}$ is closed. | |
| and | the identities and projections are closed | [[ immediate ]] |
| ∴ | their compositions are closed. | |

$[[]]_{\text{closed}}$

Continuous: (this is where we use our generalization of [Manna 74] Thm 5.1 : thm. 2.37)

| | | |
|---|---|---|
| We have | $\boldsymbol{P}$ is a FD-PCPO | [[ hypothesis ]] |
| and | $\boldsymbol{MLP}_{\boldsymbol{P},n_i}$ sub-cpo of $[\, \boldsymbol{P}^{n_i} \rightarrow \boldsymbol{P}\, ]$ | [[ thm. 2.53 ]] |
| and | $\forall\, i \in I$, $F_i$ $\subseteq$-monotonic | [[ $F_i \in \boldsymbol{MLP}_{\boldsymbol{P},n_i}$ ]] |
| and | $\forall\, i \in I$, $F_{i^\circ}$ closed | [[ above ]] |
| ∴ | $\forall\, i \in I$, $F_{i^\circ}$ continuous! | [[ thm. 2.37 ]] |
| and | the identities and projections are continuous | [[ immediate ]] |
| ∴ | their compositions are continuous. | [[ thm. 2.11 ]] |

$[[]]_{\text{continous}}$

$[[]]_{\text{Thm. 2.54}}$

# 3. Semantics of Synchronous Circuits

## 3.1. Informal view

The key to our work is to understand what a synchronous circuit is, as a mathematical object. The goal of this section is to guide you through the *evolution* of thoughts which led to the final product, and informally convince you of its appropriateness.

The final product itself is described in exacting precision in the rest of this chapter. In this first section, we have tried to maximize simplicity, and minimize the use of mathematics... We are also assuming *no* prior knowledge of history-functional semantics such as [Kahn 74], [Johnson 84] and [Kloos 87] . More advanced readers should bear with me, or simply skip this informal section.

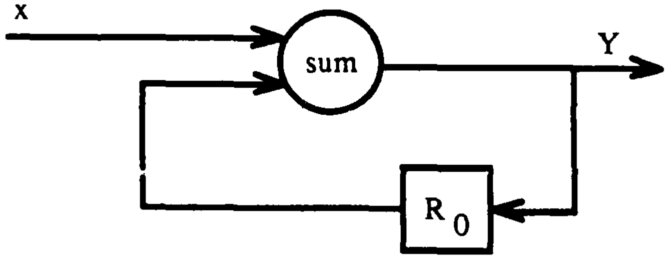
### 3.1.1. First basic intuition (circuit as a black box)

Consider as a start a *combinational* circuit, i.e. a circuit with no memory (no registers and no feedback loops). Assume that the values which can appear on the wire are binary digits (True and False), then we can identify the circuit with a *boolean function*. This is commonly done in all circuit design textbooks. In fact we can easily move from binary digits to natural numbers for example, and identify more general combinational circuits with functions on these numbers.

Abstracting slightly, consider that the values on the wires belong to an arbitrary set: $\Sigma$ , we can identify a combinational circuit with a function from $\Sigma$ to $\Sigma$ .

Once we introduce memory (or state) in the forms of feedback loops, or registers, things are not so simple. For example, consider a running sum sequential circuit (which accumulates the sum of all the inputs it has seen). It is pictured below, with the square representing a register (initialized with 0) and the circle representing an adder.

**Figure 3-1:** Running Sum Circuit



For this example, we have $\Sigma$ = the set of natural numbers. Assume the first number we present is 3, the output is 3. The next number we present is 5, the output is now 8. The next number we present is 5 again, the output is now 13. Clearly, we can no longer identify this circuit as a function on the natural numbers, since it produced a different answer on the same input number.

The solution to this problem is to consider the *sequence* of all inputs, and the *sequence* of outputs; in our case: 3.5.5 → 3.8.13 . If we ever replay the same sequence of inputs (from the start) then we will get the same sequence of outputs.

In other words, a sequential circuit can be identified with a function from se·uences of values in $\Sigma$ to sequences of values in $\Sigma$. These sequences being finite, we refer to them as "strings", and the set of strings on $\Sigma$ is called: $\Sigma^*$.

Note that a combinational circuit identified with a function f: $\Sigma \rightarrow \Sigma$ can be identified in this context as the "memory-less" function: $f^*$ which to the input: a.b.c assigns the output: f(a).f(b).f(c) . (In comparison, the function which corresponds to our register: $R_0$, assigns: 0.a.b to the input string: a.b.c) .

Therefore our conclusion at this point is that any *synchronous circuit* can be identified with a *function* from $\Sigma^*$ to $\Sigma^*$ which we will call a string-function.

However, the string-functions associated with synchronous circuits have two additional (and fundamental) properties:

- Length-Preserving: the length of their output string is always equal to the length of their input string. This is immediate since we find out what our string-function is by looking at all the wires at the end of each clock period say, and tacking these new values onto the history of previous ones for each wire.
- Monotonic: assume that on the input string x, the circuit returned the output string y . Now, assume that we add one more value u to x, making it the string: xu , then the new output string will already start with y, and the circuit will tack on a new value v to y, making the output: yv . The circuit can not "go back in time", change some of the results it had output on input x, and produce a string which does not start with y . This property is exactly monotonicity with respect to the prefix relation: $\leq$ on strings.

So, the essence of our semantics is: ***a synchronous circuit can be identified with a $\leq$-Monotonic, Length-Preserving string-function***.

Abbreviation: we temporarily define MLP= "$\leq$-Monotonic and Length-Preserving".

There are two technicalities we have ignored so far, and which we mention for completeness here:

- If the circuit has many input lines, then the corresponding string-function takes as argument a tuple of strings, all of the same length (for the same reason which led us to the conclusion that the string-function was length-preserving) .
- If the circuit has many output lines, then each output line is identified with an MLP string-function, and the circuit as a whole is identified with a tuple of such functions.

### 3.1.2. Second basic intuition (circuit as a system/network)

We now take a look at how our circuits are built. As far as we are concerned here, synchronous circuits are made from two kinds of elements:

- *Combinational* elements: elements which do not have memory, or state, and which we have associated above with $f^*$ string-functions.
- *Registers/clocked storage elements*: elements which hold values for one clock period (after which they latch in the input presented to them), and which we have associated above with the $R_a$ string-function. (The parameter: a, is the initial value of the register, in the example above it was 0.)

Note that we use the word "register" in a very narrow sense, which is common in the formal hardware specification literature [Leiserson-Saxe 83], [Johnson 84] and [Hunt 85].

Circuits are then built by connecting inputs and outputs of the above components in an almost arbitrary manner.

We say "almost" because for a synchronous circuit, every loop in the connection graph should contain at least one register. Otherwise, we get problems of asynchronous latching, oscillations, etc., i.e. not a correct synchronous circuit; see [Mano 76] and [Mead-Conway 80] for more details. For our semantics, this restriction: "Every-Loop-is-Clocked" [ELC] is not necessary (and we will come back to it in section 3.4), but at this point it is easier to keep thinking in terms of such "good" circuits.

The question is, how do we give meaning (i.e. semantics) to the network, knowing what the individual elements stand for?

If for each element in the circuit we write an equation relating the output to the input(s), then we obtain a new view of our circuit as a system of equations. If there are loops in the circuit, then the system will be recursive.

There is a standard way in semantics to give meaning to a recursive definition, and that is to consider it as an equation in a certain (appropriate) domain, and take a certain (appropriate) solution of this equation as the object being defined by the recursive definition.

This is exactly what we shall do!

Our domain is basically the set strings on $\Sigma$, and the MLP functions on it. Each node is already identified with a certain MLP function ($f^*$ or $R_a$) . A circuit, or system of equations, will be identified with some MLP function which solves that system.

A technicality which we have ignored so far, is that the "appropriate" domains we have mentioned above are *ordered* domains, i.e. there is a notion of an object being "less-defined-than" another. This relation will be denoted by: $\subseteq$ . In our case this notion of $\subseteq$ is very simple: We add to $\Sigma$ one element: ? , which should be read as "unknown". In the $\subseteq$ order, ? is $\subseteq$ all elements of $\Sigma$ , and that's it. The new set is called: $\Sigma_?$ . We then simply extend this order relation to strings (by comparing them one position at a time), and to functions on these strings (also by comparing them point by point). One basic concept of computability in these domains is that the computable functions respect the $\subseteq$ order, i.e. are $\subseteq$-Monotonic.

Pronunciation note: "$\subseteq$-monotonic" can be read "L-monotonic" (short for "less-defined-than-monotonic"); and $\leq$-monotonic can be read "P-monotonic" (for "prefix-monotonic").

We also define the following (permanent) abbreviations to ease everybody's job:
Monotonic= "$\subseteq$-monotonic and $\leq$-monotonic"; and
MLP= "Monotonic and Length-Preserving".

So, in conclusion, ***a synchronous circuit will be identified with an MLP string-function, or a tuple of such functions if there are many output lines.***

### 3.1.3. Extensional versus Intensional view of the world

There is one last subtlety which comes into play in our semantics of synchronous circuits: so far we have always said "a circuit *is identified with* a certain function". What we have really argued however is that "a circuit *computes* a certain function".

So in other words, we have associated a circuit with *what* it computes (a certain function). In doing so, we have abstracted away all information about *how* it computes that function. What we have done is to define an *extensional* semantics of synchronous circuits.

In order to retain more information in our theory, we actually define an *intensional* semantics which identifies a circuit with the functional defined by the system of equations, rather than simply its solution. We can still recover the extensional semantics simply by taking the least fixed point of that functional, and so we end up defining both the intensional and extensional semantics.

This concludes the vague view of things. The remaining sections of this chapter, together with the mathematical preliminaries of chapter 2, are intended to dot all the i's.

## 3.2. Formal Syntax

Formally, we have one basic syntactic object: "SYnchronous System Description" or "SYSD". These are essentially recursive systems of equations, together with a list of which defined functions are the designated output. They correspond very closely to engineer's "net lists". We will define a set of such syntactic objects, i.e. a language: $L_{SD}$.

Note that syntactic entities will be written in `this font`.

**Definition 3.1:** $L_{SD}$

- $L_{\text{char}}$ = countable alphabet with elements denoted by $\mathtt{a}, \mathtt{a}_1, \mathtt{a}_2$ ...
- $L_{\text{char-fun}}$ = countable ranked alphabet (elements have arity) with elements denoted by $\mathtt{f}, \mathtt{f}_1, \mathtt{f}_2$ ...
- $L_{\text{string-fun}} = \{ \mathtt{R}_{\mathtt{a}} \mid \mathtt{a} \in L_{\text{char}} \} \cup \{ \mathtt{f}^{*} \mid \mathtt{f} \in L_{\text{char-fun}} \}$ with elements denoted by $\mathtt{F}, \mathtt{F}_1, \mathtt{F}_2$ ...
- $L_{\text{input-line-var}}$ = countable alphabet with elements denoted by $\mathtt{x}, \mathtt{x}_1, \mathtt{x}_2$ ...
- $L_{\text{non-input-line-var}}$ = countable alphabet with elements denoted by $\mathtt{Y}, \mathtt{Y}_1, \mathtt{Y}_2$ ... $\mathtt{Z}, \mathtt{Z}_1, \mathtt{Z}_2$ ...
- $L_{SD} = \{ (\mathtt{in}, \ \mathtt{sys}, \ \mathtt{out}) \mid$
  `in` = tuple of input-line-vars: $(x_1, ..., x_m)$, also denoted as $\underline{x}$ for short.
  `sys` = system of equations: $\mathtt{Y}_{\mathtt{i}}(\underline{\mathtt{x}}) \leftarrow \mathtt{F}_{\mathtt{i}} \ (..,\mathtt{E}_{\mathtt{j}},..)_{\mathtt{j} \in \{1..\text{arity of } \mathtt{F}_{\mathtt{i}}\}}$, for $\mathtt{i} \in \{1..\mathtt{n}\}$
  with $\mathtt{F}_{\mathtt{i}} \in L_{\text{string-fun}}$ and $\mathtt{E}_{\mathtt{j}}$ = some input $x_k$ or non-input expression $\mathtt{Y}_{\mathtt{k}}(\underline{\mathtt{x}})$ .
  `out` is a tuple of non-input-line-vars among $\mathtt{Y}_1, .., \mathtt{Y}_{\mathtt{n}}$. $\}$
  Elements of $L_{SD}$ are denoted by $\mathtt{S}, \mathtt{S}_1, \mathtt{S}_2$ ...

As syntactic sugar, we will sometimes omit the input variables $(\mathtt{x}_1, .., \mathtt{x}_{\mathtt{k}})$ or $\underline{\mathtt{x}}$ as arguments for $\mathtt{Y}_{\mathtt{i}}$'s in the system, so that $\mathtt{Y}_5 \leftarrow \mathtt{f}^{*}(\mathtt{Y}_3, \mathtt{Y}_1, \mathtt{x}_4)$ will be a legal equation. Note that in this sugared form, our syntax is almost identical to the one used in [Kloos 87] in its "applicative" form. Our reason for not using the sugared form as the primary syntax is that we can view our syntactic objects as restricted expressions in a more general string expression language, and under that angle, we want our expressions to be well-typed.

One weakness of $L_{SD}$ as defined is that it is "flat". It does not allow user-defined string-functions (sub-systems). We did this because treating such objects formally brings semantic complications which are *orthogonal* to the problem at hand: semantics of synchronous concurrent systems. Informally, we treat them as follows:

- Non-recursive string-function definitions, i.e. macros, are simply expanded out.
- Recursive string-function definitions are disallowed. They correspond to non-directly implementable specifications; they are studied in [Johnson 84]. Alternatively they define networks which reconfigure themselves (expand and contract) during execution; see [Glasgow-MacEwen 87] for this view in the context of operator nets.

$L_{SD}$ is a fine language for mathematical and computer treatment. For human interaction however, a graphical language is more appropriate. We will therefore define a second language: $L_{SDGraph}$, of sysd's in graphical form. $L_{SDGraph}$ is isomorphic to $L_{SD}$, and we will give a (trivial) translation function.

**Definition 3.2:** $L_{SDGraph}$
A sysd is a multi-graph (V,E), where vertices are of 2 types:

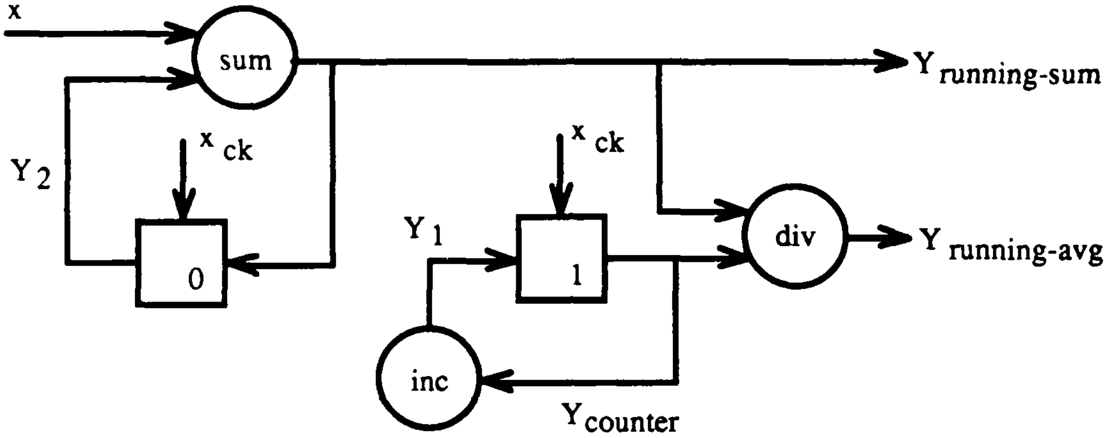
- VCombinational: represented with a circle, and a char-function letter per out-edge. They have n in-degree, and m out-degree, with n,m ≥ 1 .

- VRegister: represented with a square, and a character letter. They have in-degree 2, and out-degree 1.

and where edges have at most 1 From-node, and at least a From-node or a To-node (and usually both). Edges with no From-node are called "Input edges". Some non-input edges are designated as "Output edges".

At this point, an example should help:

**Figure 3-2:** Example: Running Sum/Avg Sysd



Or in sugared $L_{SD}$ :

$Y_{running-sum} \leftarrow \texttt{sum*}(x, Y_2)$
$Y_2 \leftarrow R_0(Y_{running-sum}, x_{ck})$
$Y_{running-avg} \leftarrow \texttt{div*}(Y_{running-sum}, Y_{counter})$
$Y_{counter} \leftarrow R_1(Y_1, x_{ck})$
$Y_1 \leftarrow \texttt{inc*}(Y_{counter})$

In the future, and as commonly done in synchronous circuit design, we will often omit the 2nd input of Registers (the clock input: $x_{ck}$) from graphical or sugared sysd's.

Note: As they stand, elements of $L_{SDGraph}$ are not "classical" mathematical graphs, since an edge here is not just a pair of vertices, but instead, a pair: (0 or 1 vertex,0 or 1 or many vertices). We could reduce these objects to standard graphs simply by introducing additional ("duplicate") vertices, but there is no point in doing so, since we only intend $L_{SDGraph}$ as a front-end (auxiliary) language, and not as a tool for meta-proofs.

**Definition 3.3: Translation:** $L_{SDGraph} \rightarrow L_{SD}$
Let the input edges be: $x_1, \ldots, x_m$, and the non-input edges be: $Y_1, \ldots, Y_n$. Define:
`in` = tuple of input edges.
`sys` =

- For each node in VCombinational, for each out-going edge (out-edge: $Y_i$, char-function letter: $f_i$), add the equation: $Y_i \leftarrow f_i^*(\ldots, E_j, \ldots)$, where $\ldots, E_j, \ldots$ are the incoming edges (either $x_k$'s or $Y_k$'s).

- For each node in VRegisters (out-edge: $Y_i$, character letter: a), add the equation: $Y_i \leftarrow R_a(E_1, E_2)$, where $E_1$ and $E_2$ are the incoming edges.

out = tuple of designated output edges.

## 3.3. Denotational Semantics

The mathematical foundation of our denotational semantics is a String Induction Algebra, of string-functions, and string-functionals. A sysd will be (compositionally) mapped, by [[ ]], into a *string-functional*, or more precisely, a system of functionals. This is in the spirit of [Talcott 85] and [Moschovakis 83], and preserves *intensional* information about the sysd - how it computes - as well as its extensional denotation - what it computes.

Since however, for many of our purposes, we are interested in the *extensional* denotation of the system, we also define an extensional denotation function, $\mu$, which maps a sysd into the tuple of *string-functions* which it computes, and which is the least fixed point of the system of functionals.

**Construction of the String Induction Algebra:**

We have a countable *alphabet:* $\Sigma$, elements of which are denoted by: a, b, c, $a_1$, $b_1$, $c_1$, ... for constants, and u, v, $u_1$, $v_1$, ... for variables. Now we lift the alphabet $\Sigma$, with least element "?": $\Sigma_?$, and get the corresponding $\subseteq$ (flat) order, and we take Strings of $\Sigma_?$: $\Sigma_?^*$, with the induced $\subseteq$ order. Elements of $\Sigma_?^*$ are denoted by: x, y, z, ... for variables, and $\varepsilon$: the empty string, as the only constant.

For reasons explained in 3.1, we are interested in functions on $\Sigma_?^*$ which are $\subseteq$-monotonic, $\leq$-monotonic and Length-Preserving, and which we can define recursively from the following functions:

**Definition 3.4: Primitive string-functions**

- $R_a : (\Sigma_?^*)^{\underline{2}} \rightarrow \Sigma_?^*$ defined by: $R_a(\varepsilon,\varepsilon) = \varepsilon \;\wedge\; R_a(x.u, x_{ck}.v) = a.x$, for $a \in \Sigma$. We call $R_a$ a "register" string-function.
- $f^* : (\Sigma_?^*)^{\underline{n}} \rightarrow \Sigma_?^*$ defined by: $f^*(\varepsilon,\ldots,\varepsilon) = \varepsilon \;\wedge\; f^*(x_1.u_1,\ldots,x_n.u_n) = f^*(x_1,\ldots,x_n) \,.\, f(u_1,\ldots,u_n)$, for $f \in [\Sigma_?^n \rightarrow \Sigma_?]$. We call $f^*$ a "combinational" string-function. It is simply the homomorphic extension of a $\subseteq$-monotonic function on $\Sigma_?$ to strings (of equal length).

Note about Registers: informally, we had treated $R_a$ as a unary function. Formally, we've defined it as a binary function, which ignores its 2nd argument! This is only a semantic subtlety, the reason for it is clear when you consider what happens if you fuse the output of a register with its "main" input. The results of this operation is a perfectly meaningful synchronous circuit, which keeps outputting the same character, *at every clock tick!* In other words, the 2nd argument (the clock) is not entirely ignored. It's just that all its information (its length) is also given by the main input, as long as it exists. Whenever the clock input remains the sole input to the circuit, then it becomes semantically significant.

**Theorem 3.5: $R_a$ and $f^*$ are MLP**
(Recall that MLP= "$\subseteq$-monotonic and $\leq$-monotonic and Length-Preserving".)

**Proof:**
Immediate verification.

$[[]]_{\text{Thm. 3.5}}$

Therefore we can now instantiate the main results of chapter 2, and get the keystone of our denotational semantics: the string induction algebra.

**Theorem 3.6: $MLP_\Sigma$ Continuous Induction Algebra**
The MLP functions on $\Sigma_?^*$, and functionals defined from $R_a$'s and $f^*$'s, form a continuous induction algebra, which we call: $MLP_\Sigma$.

**Proof:**

We have $\Sigma_?$ is a flat CPO [[ by construction ]]
$\therefore$ $\Sigma_?$ is a FD-CPO [[ thm. 2.31 ]]
and $\Sigma_?$ has a least element [[ by construction ]]
$\therefore$ $\Sigma_?$ is a FD-PCPO

The result is now an immediate instantiation of thm. 3.5 and thm. 2.54 where we have slightly abused the terminology in exchange for simplicity...

$[[]]_{\text{Thm. 3.6}}$

We can now define our (intensional) denotational semantics:

**Definition 3.7: Intensional Denotational Semantics: $[\![\ ]\!]$**
Let $S \in L_{SD}$, $S = (\texttt{in}, \texttt{sys}, \texttt{out})$ with non-input lines $Y_i$, $i \in \{1..n\}$, and input lines $x_j$, $j \in \{1..m\}$:

- $L_{SD}$ : $[\![ S ]\!] = (\text{in}, [\![ \texttt{sys} ]\!], \text{out})$ ; $[\![ \texttt{sys} ]\!]$ will be called $\tau_S$ . $\tau_S = (\tau_1,..,\tau_n)$ where $\tau_i = \lambda(Y_1,..,Y_n).[\ \lambda(\underline{x}).\ [\![ \texttt{F}_i ]\!] (..,E_j,..)\ ]$ for equation: $\texttt{Y}_i \leftarrow \texttt{F}_i(\texttt{..}, \texttt{E}_j, \texttt{..})$
- $L_{\text{string-fun}}$ : $[\![ \texttt{R}_a ]\!] = R_{[\![ a ]\!]}$ and $[\![ \texttt{f*} ]\!] = [\![ \texttt{f} ]\!]^*$ .
- $L_{\text{char-fun}}$ : $[\![ \texttt{f} ]\!]$ = some operation on $\Sigma$, naturally extended to $\Sigma_?$ .
- $L_{\text{char}}$ : $[\![ \text{a} ]\!]$ = some character in $\Sigma$ .

Formally, our semantics is parametrized by an algebra $\Sigma$ with some fixed set of constants and operations.

And the (derived) extensional semantics:

**Definition 3.8: Extensional Denotational Semantics: $\mu$**
Let $S \in L_{SD}$, $S = (\texttt{in}, \texttt{sys}, \texttt{out})$ and $[\![ \texttt{sys} ]\!] = \tau_S = (\tau_1,..,\tau_n)$ . We define the extensional semantics of $S$ as the least fixed point of its intensional semantics, i.e. a tuple of string-functions, from which we keep only the selected output lines: $\mu(S) = \text{LFP}(\tau_1,..,\tau_n)_{\text{out}}$ .

To justify this definition: we have $MLP_\Sigma$ is a continuous induction algebra (thm. 3.6) therefore (thm. 2.16), the system $(\tau_i)_{i \in \{1..n\}}$ has a Least Fixed Point in $MLP_\Sigma$: $\text{lub}[(\tau_1,..,\tau_n)^j(Q,..,Q)]_{j \in \omega}$ . (Recall that $Q = \lambda\ \underline{x}\ .\ ?\uparrow^{|\underline{x}|}$ .)

Just to add a touch of concreteness to these definitions, we continue with the example presented in section 3.2, in figure 3-2.

Assuming we've selected the lines: $\texttt{Y}_{\texttt{running-sum}}$ and $\texttt{Y}_{\texttt{running-avg}}$, then its extensional semantics is a pair of string-functions (where the characters are numbers):
$(\ \lambda\ x\ x_{ck}\ .\ \Theta_{i=1}^{|x|}\ (\Sigma_{j=1}^{i}\ x\downarrow_j)\ ,\ \lambda\ x\ x_{ck}\ .\ \Theta_{i=1}^{|x|}\ ((\Sigma_{j=1}^{i}\ x\downarrow_j)\ /\ i\ )\ )$ .

Its intensional semantics is the system of functionals which would be described exactly like the sysd in recursive form (except for the font).

## 3.4. Mathematical characterization of "Every-Loop-is-Clocked"

It is one of the most basic facts of synchronous circuit design that some "building rule" has to be observed: every loop in the circuit should contain a clocked storage element, or more tersely: Every Loop is Clocked [ELC] . Our semantics gives a meaning (assigns string-functions) to all circuits, including those with "illegal" connections. Intuitively however, there is a distinction between "good" synchronous circuits and others.

The goal of this section is to formalize this intuition, i.e. find a mathematical property enjoyed by the "legal" circuits, and prove that the extensional semantics of ELC sysds have that property.

In order to carry this out precisely, we need to define several simple concepts about synchronous circuits:

**Definition 3.9: Predecessor**
Let $S$ be a sysd, with non-input lines: $Y_i$, $i \in \{1..n\}$ , $Y_k$ is a predecessor of $Y_i$ $<=>$ $Y_i \leftarrow F_i(...,Y_k,...)$ ,
i.e. $Y_k$ appears as one of the arguments for $Y_i$.

**Definition 3.10: Path**
Let $S$ be a sysd. A path is a sequence $P = (Z_1,...,Z_p)$ such that Z's are non-input lines in $S$ and $Z_j$ is a predecessor of $Z_{j+1}$ , $\forall j \in \{1..p-1\}$ .

We denote the set of Paths of a sysd $S$ by: Paths($S$) .

**Definition 3.11: Loop**
Let $P = (Z_1,...,Z_p) \in$ Paths($S$) , Loop(P) $<=>$ $Z_p = Z_1$ .

**Definition 3.12: Register-line, Combinational-line**
Let $S$ be a sysd, with non-input lines: $Y_i$ , and equations: $Y_i \leftarrow F_i(...)$ $i \in \{1..n\}$ ,

- $Y_i$ is a Register-line $<=>$ $F_i = R_a$, for some a .
- $Y_i$ is a Combinational-line $<=>$ $F_i = f^*$, for some f .

**Definition 3.13: Path is Clocked**
Let $P = (Z_1,...Z_p) \in$ Paths($S$) , Clocked(P) $<=>$ $\exists j \in \{1..p\} \mid Z_j$ is a Register-line .

Note: the set of all non-clocked paths is the set of all combinational paths through the sysd. It could be totally ordered by appropriately defined weights (delays) on combinational nodes. Its max weight element would then be the "critical path".

**Definition 3.14: Every-Loop-is-Clocked [ELC]**
Let $S$ be a sysd. ELC($S$) $<=>$ $\forall P \in$ Paths($S$) , Loop(P) $=>$ Clocked(P) .

The fact which is informally known in the engineering community, but which I have never seen formally mentioned in any form in the "theoretical" literature is then:

**Theorem 3.15: ELC => Total on $\Sigma^*$**
Let $S$ be a sysd, ELC($S$) $=>$ $\mu(S)$ is total on $\Sigma^*$ .
And more generally: ELC($S$) $=>$ LFP($\tau_S$) is total on $\Sigma^*$ , i.e. the results applies to all the lines of the circuit, not just the ones selected for output.

Important note: all functions we've dealt with so far were "total" functions, but on $\Sigma_?^*$ . The additional property of being total on $\Sigma^*$ means that if the input is in $\Sigma^*$ (i.e. has no ? in it) then so does the output. This is *not* enjoyed in general by arbitrary functions on $\Sigma_?^*$ .

The proof rests on two observations about iterations of Kleene's algorithm in $\boldsymbol{MLP_\Sigma}$ . "Kleene's algorithm" is simply the constructive method used to reach the Least Fixed Point of a continuous functional in Kleene's theorem (thm. 2.14), as the least upper bound of a chain built by iterating the functional starting with the least element of the PCPO.

Informally the proof goes as follows. On any sysd, for an input $\in \Sigma^*$ (i.e. with no ? in it):

- At each Kleene iteration (applied to the input), all values (on all lines) have a particular shape: some "real" (non-?) characters, followed by some ?'s , and each iteration "pushes" the ?'s a little further to the right (or leaves the value unchanged).
- If the algorithm stabilizes with some line still having ?'s in it, then we can "climb back" from that line and extract a loop of combinational-lines (i.e. a non-clocked loop).

More precisely:

**Definition 3.16: K-view**
Let S = (in, sys, out) be an arbitrary Sysd, $\underline{x}$ an arbitrary input. Let $\tau_S = [\![ \text{ sys } ]\!] = \tau_1,..,\tau_n$ .

Define $K^j = (\tau_1,...,\tau_n)^j(Q,...,Q)(\underline{x}) = (K^j{}_1,...,K^j{}_n)$ . Figuratively, $K^j$ is the "view" of the values on all the lines of S, after the j'th iteration of Kleene's algorithm. For example, $K^0 = ( ?\uparrow^{|\underline{x}|} ,..., ?\uparrow^{|\underline{x}|} )$ .

The first observation is expressed in the following lemma:

**Theorem 3.17: K-view shape**
Let $S \in L_{SD}$ , with non-input lines $Y_i$, $i \in \{1..n\}$ and m input lines. Let $\underline{x} \in (\Sigma^*)^{\underline{m}}$ , $\forall j \in \omega$ , $\forall i \in \{1..n\}$ , $\exists\, \rho_{j,i} \in \{0..|\underline{x}|\} \mid K^j{}_i = c{\downarrow}_{1..\rho_{j,i}} . ?\uparrow^{|\underline{x}|-\rho_{j,i}}$ with $c{\downarrow}_{1..\rho_{j,i}} \in \Sigma^*$ , i.e. informally: $K^j{}_i = c_1..c_{\rho_{j,i}}.??..?$ with c's $\neq$ ? .
**Proof:**
Assume [h1] $\underline{x} \in (\Sigma^*)^{\underline{m}}$ . We induct on j (i.e. on Kleene iterations) with predicate:
$\forall i \in \{1..n\} , \exists\, \rho_{j,i} \in \{0..|\underline{x}|\} \mid K^j{}_i = c{\downarrow}_{1..\rho_{j,i}} . ?\uparrow^{|\underline{x}|-\rho_{j,i}}$

Base case: immediate [[ take $\rho_{0,i} = 0$ , $\forall$ i ]]

$[[]]_{\textbf{base case}}$

Induction step: (assume ok for j). Let i arbitrary $\in \{1..n\}$

If $Y_i$ is a register-line: $Y_i \leftarrow R_a(Y_k)$ , then:
We have $K^{j+1}{}_i = a . K^j{}_k{\downarrow}_{1..|x|-1}$ [[def. Kleene's algorithm]]
$\therefore$ $K^{j+1}{}_i = a . c{\downarrow}_{1..\rho_{j,k}} . ?\uparrow^{|\underline{x}|-\rho_{j,k}-1}$ [[ induction hyp., instantiating general i to k ]]
i.e. we have added a non-? character on the left, and chopped off a ? (if any) from the right.
$\therefore$ $K^{j+1}{}_i$ is "of the right shape" $\wedge$ $\rho_{j+1,i} = \text{if } \rho_{j,k} = |\underline{x}| \text{ then } |\underline{x}| \text{ else } 1+\rho_{j,k}$

If $Y_i \leftarrow R_a(x_k)$ , then:
We have $K^{j+1}{}_i = a . \underline{x}_k{\downarrow}_{1..|\underline{x}|-1}$ [[def. Kleene's algorithm]]
$\therefore$ there are no ? in $K^{j+1}{}_i$ [[ $\underline{x}_k \in \Sigma^*$ by h1, a $\neq$ ? by definition, 3.7 ]]
$\therefore$ $K^{j+1}{}_i$ is "of the right shape" $\wedge$ $\rho_{j+1,i} = |\underline{x}|$

If $Y_i$ is a combinational-line: $Y_i \leftarrow f^*(..,Y_k \text{ or } x_k,..)$ , then:
We have $..,K^j{}_k,..$ are "of the right shape" [[ induction hyp. ]]
and all $\underline{x}_k$'s have no ? in them [[ hypothesis h1 ]]

and $K^{j+1}{}_{i} = f^{*}(\ldots K^{j}{}_{k} \text{ or } \underline{x}_{k}, \ldots)$ [[def. Kleene's algorithm]]  
and f is a naturally extended function : $(\Sigma^{*})^{\underline{n}} \rightarrow \Sigma$ [[ by definition. 3.7 ]]  
$\therefore$ Consider any position: $pos \in \{1..|\underline{x}|\}$ :  
We have $K^{j+1}{}_{i}\downarrow_{pos} = f(\ldots K^{j}{}_{k}\downarrow_{pos} \text{ or } \underline{x}_{k}\downarrow_{pos}, \ldots)$ [[def. f* , 3.4]]  
and $\underline{x}_{k}\downarrow_{pos} \neq ?$ therefore:  
if for all predecessors, $K^{j}{}_{k}\downarrow_{pos} \neq ?$ then $K^{j+1}{}_{i}\downarrow_{pos} \neq ?$  
if for some predecessor, $K^{j}{}_{k}\downarrow_{pos} = ?$ then $K^{j+1}{}_{i}\downarrow_{pos} = ?$  
$\therefore$ $K^{j+1}{}_{i}$ is "of the right shape" $\wedge$ $\rho_{j+1,i} = \min\{\rho_{j,k}, Y_{k}$ predecessors of $Y_{i}\}$ or $|\underline{x}|$ if all the arguments are input-lines.

$[[]]_{\text{induction step}}$

$[[]]_{\text{Thm. 3.17}}$

The second observation becomes the proof (by contradiction) of the ELC theorem:

**Proof:**

Let $s \in \mathcal{L}_{SD}$, with non-input lines $Y_{i}$, $i \in \{1..n\}$ and m input lines.  
Assume :  
[h1] $\underline{x} \in (\Sigma^{*})^{\underline{m}}$  
[h2] $\exists\, j \in \omega \mid K^{j+1} = K^{j}$, i.e. the algorithm is stable at the j'th iteration.  
[h3] $\exists\, i_0 \in \{1..n\} \mid \rho_{j,i_0} < |\underline{x}|$, i.e. there is still at least one ? in $K^{j}{}_{i_0}$ .

We now extract a predecessor of $Y_{i_0}$ which also has some ? left in it:

if $Y_{i_0}$ is a register-line, then its argument can not be an input line because inputs are assumed to have no ? in them and hence $K^{j,i_0}$ would have no ? in it, $\forall j > 0$ .  
$\therefore$ $Y_{i_0} \leftarrow R_{a}(Y_{i_1})$  
We have $\rho_{j+1,i_0} = \text{if } \rho_{j,i_1} = |\underline{x}| \text{ then } |\underline{x}| \text{ else } 1+\rho_{j,i_1}$ [[ proof of Shape lemma ]]  
and $\rho_{j+1,i_0} = \rho_{j,i_0}$ [[ hypothesis h2 ]]  
and $\rho_{j,i_0} < |\underline{x}|$ [[ hypothesis h3 ]]  
$\therefore$ $\rho_{j,i_1} < |\underline{x}|$ mainly, and also: $\rho_{j,i_1} > \rho_{j,i_0}$ .

if $Y_{i_0}$ is a combinational-line: $Y_{i_0} \leftarrow f^{*}(\ldots, Y_{k} \text{ or } x_{k}, \ldots)$ . Again, because inputs have no ? in them and $K^{j,i_0}$ contains some ? , at least some arguments must be non-input lines.  
$\therefore$ $\rho_{j+1,i_0} = \min\{\rho_{j,k}, Y_{k}$ predecessors of $Y_{i_0}\}$ [[ proof of Shape lemma ]]  
Let $i_1$ be some predecessor yielding the minimum $\rho$,  
then $\rho_{j,i_1} = \rho_{j+1,i_0}$  
and $\rho_{j+1,i_0} = \rho_{j,i_0}$ [[ hypothesis h2 ]]  
and $\rho_{j,i_0} < |\underline{x}|$ [[ hypothesis h3 ]]  
$\therefore$ $\rho_{j,i_1} < |\underline{x}|$ mainly, and also: $\rho_{j,i_1} = \rho_{j,i_0}$ .

By this process we've extracted a predecessor of $Y_{i_0}$ : $Y_{i_1}$ such that $\rho_{j,i_1} < |\underline{x}|$ , which was the hypothesis we had on $i_0$ therefore we can reiterate this process.

Remark: From the construction above we also get:  
[r1] in either case, $\rho_{j,i_1} \geq \rho_{j,i_0}$  
[r2] $\rho_{j,i_1} = \rho_{j,i_0}$ <=> $Y_{i_0}$ is a combinational-line.

We now build a path by starting with $P = (Y_{i_0})$, and:

- If $Y_{i_j}$ does not already appear in P, we add it to P, and reiterate. Since there are *finitely* many lines in S, we must eventually hit the other case:
- If $Y_{i_j}$ does appear in P, we add it to P and stop: we have now obtained a path which contains a loop!

More precisely, at the end of this (finite) process we have: $P = (Y_{i_0}, Y_{i_1}, \ldots, Y_{i_q}, Y_{i_{q+1}}, \ldots, Y_{i_q})$ for some q. Extract the loop $L = (Y_{i_q}, Y_{i_{q+1}}, \ldots, Y_{i_q})$.

From [r1], we know that the ρ's are weakly increasing along L. And they must be equal at both ends (because L is a loop), therefore they are constant along L. From [r2], the ρ's can only be constant if the lines are combinational-lines.

$\therefore$ L is a loop of combinational-lines in the sysd S

Therefore, the contrapositive is that if S has no combinational loops, i.e. ELC(S), and if the input $\underline{x}$ has no ? in it, and if Kleene's algorithm terminates at the j'th iteration then:

$\forall\, i \in \{1..n\}\ \rho_{j,i} = |\underline{x}|$, i.e. $K^j_i \in \Sigma^*$

and $K^j = LFP(\tau_S)(\underline{x})$ [[ by def. of K-view, and Kleene's thm. ]]

$\therefore$ $LFP(\tau_S)(\underline{x}) \in (\Sigma^*)^{\underline{n}}$

$[[]]_{\text{Thm. 3.15}}$

## 3.5. Operational semantics and Equivalence with (extensional) Denotational semantics

An operational semantics is a different way to assign meaning to a circuit with a more "dynamic" or algorithmic flavor than the denotational semantics. It usually refers to concepts such as state and transition steps, and iterativel computes the outputs from the inputs and the circuit. This is in contrast to the (extensional) denotational semantics which are considered more "static", just stating what the outputs should be (least fixed points of a system of equations) without explicitely constructing them. This however, is only a question of taste since Kleene's theorem for reaching the LFP is constructive and easily implementable.

Proving the equivalence of an operational semantics/algorithm and the (extensional) denotational semantics can be seen under two angles:

- as an additional justification for the denotational semantics, if the operational semantics is "intuitively right",
- or as a proof of correctness of the algorithm, if one believes first in the denotational aspect of the computation.

In this work, our goal is the first angle. We therefore have to pick an operational semantics which is as "intuitively right" as possible to people who would be skeptical of our denotational semantics. To that end, we will give *two* operational semantics, both based on states, and character by character operation, but with a slight distinction:

- The 1st one uses a "big" state: the history of all values seen on all lines, and is therefore a little "abstract". We will refer to it as our "operational semantics".
- The 2nd one uses a more practical state: the current value held in all registers, and is essentially the simplest simulation algorithm for synchronous circuits [Russell-Kinniment-Chester-McLauchlan 85], and hence, quite "concrete". We will refer to it as our "simulation semantics".

And we will prove equivalence with the (extensional) denotational semantics for both of them.

**Definition 3.18: Informal Operational Semantics**
For a given ELC circuit S with non-input lines $Y_i$, $i \in \{1..n\}$, and input lines $x_j$, $j \in \{1..m\}$, we define the

state s = $(s_Y,s_x)$ to be the history of all characters seen on each line.

We define a "next-output" function $\delta_S$ which takes the state $(s_Y,s_x)$ and an input character (for each input line) and returns an output character (for each non-input line) as follows:

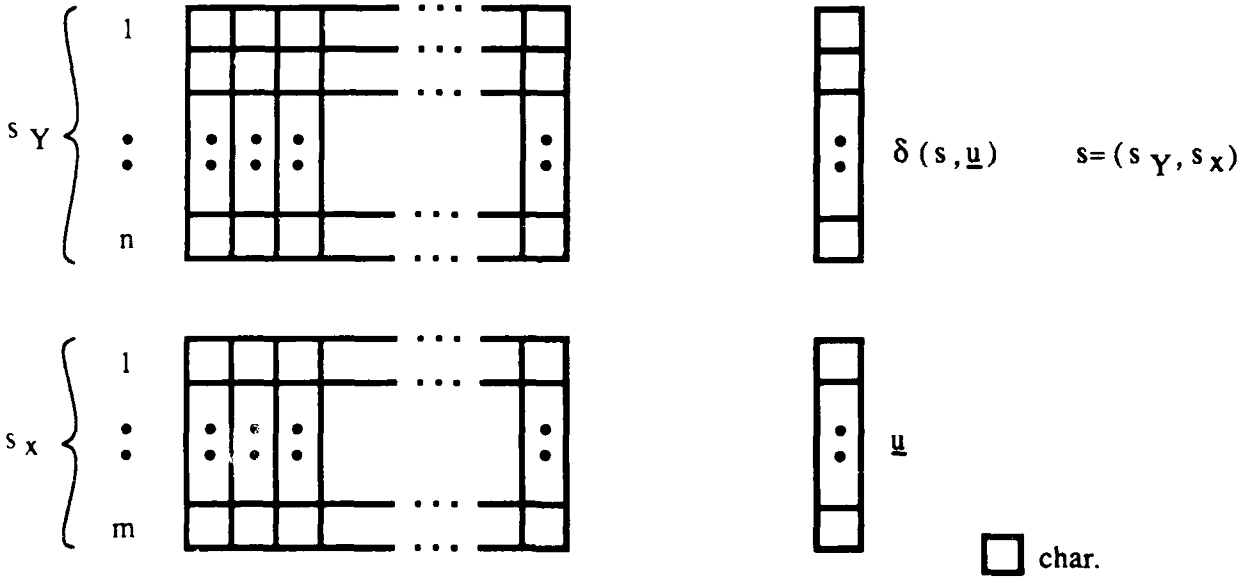
- Case: Register-line $Y_i \leftarrow R_a(Y_k)$ : Return the LAST character which appeared on $Y_k$ so far, because that's the character which is currently being held in the register. We can get that character from the state: $s_{Y_k}$. If there was none, i.e. we are in the initial condition, then return "a".
  If the argument is an input line, lookup the value in $s_x$ instead of $s_Y$.
- Case: Combinational-line $Y_i \leftarrow f^*(..,Y_k,..)$ : Recursively compute the next-output for the predecessor lines and apply f to them.
  If some argument is an input line, then take the current input character for that line.

We also define a "next-state" function $\gamma_S$ which simply tacks on the new character produced by $\delta_S$ to the current state. (And for the input part of the state, tacks on the new input values.)

Then we extend both of these functions to handle *strings* of inputs by iterating the character by character functions, while starting in the initial, empty, state. This yields the "complete-output" function $\Delta_S$ and the "final-state" function $\Gamma_S$.

Pictorially, the set-up looks like this:

**Figure 3-3:** Operational Semantics



Notes:

- The function $\delta_S$ is recursive in an unusual way in the combinational case: it calls itself on all the predecessors of the current line. But since we assume that all loops are clocked (ELC circuit) then these recursive calls will eventually hit a Register-line or an input-line and terminate. We will justify this formally below.
- The 2nd input to "$R_a$" equations was not mentioned because the operational semantics ignores it. (The clock beat is in some sense hardwired in the string recursion.) More precisely, the equivalence theorem is true no matter what line is plugged into the 2nd argument of Registers. However the operational

model matches the *reality* of physical registers only if $x_{ck}$ is indeed connected to their clock pin (and if other physical considerations such as timing, electrical issues, etc... are also correct).

- To lighten up our notations the S subscript will be omitted from here on. Also, we will make use of an "or respectively" notation, to express definitions which are very similar in two symmetric cases (argument is a non-input-line, or input-line). This will be clear with the examples below.

**Definition 3.19: Formal Operational Semantics**
Let $S \in L_{SD}$, with non-input lines $Y_i$, $i \in \{1..n\}$ and input lines $x_j$, $j \in \{1..m\}$, and ELC(S).

Let $s_Y \in (\Sigma_?{}^*)^{\underline{n}}$, $s_x \in (\Sigma_?{}^*)^{\underline{m}}$, $\underline{x} \in (\Sigma_?{}^*)^{\underline{m}}$, $\underline{u} \in (\Sigma_?)^{\underline{m}}$, $\underline{v} \in (\Sigma_?)^{\underline{m}}$.

Define $\delta(s_Y, s_x, \underline{v}) \in (\Sigma_?)^{\underline{n}}$ by: for $i \in \{1..n\}$,

- if $Y_i \leftarrow R_a(Y_k \text{ or } x_k)$ then $\delta(s_Y, s_x, \underline{v})_i = \text{if } s_{Y_k \text{ or } x_k} = \varepsilon \text{ then } a \text{ else } \text{last}(s_{Y_k} \text{ or } s_{x_k})$
- if $Y_i \leftarrow f^*(\ldots Y_k \text{ or } x_k, \ldots)$ then $\delta(s_Y, s_x, \underline{v})_i = f(\ldots, \delta(s_Y, s_x, \underline{v})_k \text{ or } \underline{v}_k, \ldots)$

Define $\gamma(s_Y, s_x, \underline{v}) = (\, s_Y . \delta(s_Y, s_x, \underline{v}) \,,\, s_x . \underline{v}\,)$

And the string-extended functions are defined by recursion on the input string:

$\Delta(\underline{\varepsilon}) = \underline{\varepsilon}$ and $\Delta(\underline{x}.\underline{u}) = \Delta(\underline{x}) \,.\, \delta(\Gamma(\underline{x}), \underline{u})$

$\Gamma(\underline{\varepsilon}) = \underline{\varepsilon}.\underline{\varepsilon}$ and $\Gamma(\underline{x}.\underline{u}) = \gamma(\Gamma(\underline{x}), \underline{u})$

It should be obvious from the state set-up (or the defining equations) that the "complete output" and the "final state" are essentially the same, and that therefore the defining equation for $\Delta$ can be simplified, by replacing $\Gamma$ by $\Delta$. More precisely:

**Theorem 3.20: $\Delta$ simplification**
$\forall\, \underline{x}$ in $(\Sigma_?{}^*)^{\underline{m}}$, $\underline{u} \in (\Sigma_?)^{\underline{m}}$, $\Gamma(\underline{x}) = (\, \Delta(\underline{x}), \underline{x}\,)$ and therefore $\Delta(\underline{x}.\underline{u}) = \Delta(\underline{x}) \,.\, \delta(\Delta(\underline{x}), \underline{x}, \underline{u})$

The first equality is proved by a simple structural induction on $\underline{x}$ ; the second is then a trivial substitution into the definition of $\Delta$.

**Proof:**

Case $\underline{\varepsilon}$ :

We have $\Delta(\underline{\varepsilon}) = \underline{\varepsilon}$ [[ def. 3.19 ]]

and $\Gamma(\underline{\varepsilon}) = \underline{\varepsilon}.\underline{\varepsilon}$ [[ def. 3.19 ]]

$\therefore$ $\Gamma(\underline{\varepsilon}) = (\, \Delta(\underline{\varepsilon}), \underline{\varepsilon}\,)$

$[[]]_{\underline{\varepsilon}}$

Case $\underline{x}.\underline{u}$:

We have $\Gamma(\underline{x}.\underline{u}) = \gamma(\Gamma(\underline{x}), \underline{u})$ [[ def. 3.19 , expanding $\Gamma$ ]]

and $\Gamma(\underline{x}) = (\, \Delta(\underline{x}), \underline{x}\,)$ [[ induction hypothesis ]]

$\therefore$ $\Gamma(\underline{x}.\underline{u}) = \gamma(\Delta(\underline{x}), \underline{x}, \underline{u})$

$\therefore$ $\Gamma(\underline{x}.\underline{u}) = (\, \Delta(\underline{x}).\delta(\Delta(\underline{x}), \underline{x}, \underline{u}) \,,\, \underline{x}.\underline{u})$ [[ def. 3.19 , expanding $\gamma$ ]]

$\therefore$ $\Gamma(\underline{x}.\underline{u}) = (\, \Delta(\underline{x}).\delta(\Gamma(\underline{x}), \underline{u}) \,,\, \underline{x}.\underline{u})$ [[ simplifying $\Delta(\underline{x}), \underline{x}$ w/ induction hyp. ]]

and $\Delta(\underline{x}.\underline{u}) = \Delta(\underline{x}).\delta(\Gamma(\underline{x}), \underline{u})$ [[ def. 3.19 , expanding $\Delta$ ]]

$\therefore$ $\Gamma(\underline{x}.\underline{u}) = (\, \Delta(\underline{x}.\underline{u}) \,,\, \underline{x}.\underline{u}\,)$

$[[]]_{\underline{x}.\underline{u}}$

$[[]]_{\text{Thm. 3.20}}$

**Remark: Totality of the functions $\delta, \gamma, \Delta, \Gamma$**

- $\Delta$, $\Gamma$ and $\gamma$ are primitive recursive in $\delta$; i.e. assuming $\delta$ is total, their totality is simply a structural induction on $\underline{x}$ (i.e. well-founded induction on the $\leq$ (prefix) relation in $\Sigma_?{}^*$ .
- $\delta$ is more unusual: it recurses on its "line" argument (noted as a subscript) in the Combinational line case. I.e. it calls itself back on the predecessor lines of the current combinational line.
  This corresponds to well-founded induction on the predecessor ordering of the circuit "cut" at each Register, i.e. where all Register-lines are considered as sources together with the input lines. Clearly if the circuit is ELC, then all loops have at least a Register-line, and when these loops are "cut" at the Register, the resulting directed graph is *acyclic*, and hence the "R-cut-predecessor" relation is well-founded.
  Therefore the proof of totality for $\delta$ is simply a well-founded induction with the R-cut-predecessor relation on its line argument.

The main reason for all this set-up is of course:

**Theorem 3.21: Operational-Denotational Equivalence**
Let `S = (in, sys, out)` be an ELC sysd (with m inputs), we have: $\forall\, \underline{x} \in (\Sigma^*)^{\underline{m}}$, $\Delta_S(\underline{x})_{out} = \mu(S)(\underline{x})$ .

Or in other words: for all "true" synchronous circuits and inputs, the operational and denotational semantics agree.

The key idea of the proof is that the "complete-output" function $\Delta$ is a fixed point of $\tau_S$ (the functional system denoted by S) , and also of course that it is in the right domain: $\boldsymbol{MLP}_\Sigma$. The inequality $\mu(..) \subseteq \Delta(..)$ is then an immediate consequence of the fact that any fixed point is at least as defined as the *least* fixed point. The ELC-characterization of the previous section gives us that for an ELC circuit and input with no ? in it, the denotational semantics returns strings with no ? in them, i.e. *maximal* strings *under* $\subseteq$ , and this yields the equality.

**Proof:**
Let S be an ELC sysd with lines $Y_i$, $i \in \{1..n\}$ and input lines $x_j$, $j \in \{1..m\}$ .

We want to prove: $MLP(\Delta) \;\wedge\; \tau_S(\Delta) = \Delta$ , which is equivalent to the coujunction of:
[LP]: $\forall\, \underline{x} \in (\Sigma_?{}^*)^{\underline{m}}$ , $|\Delta(\underline{x})| = |\underline{x}|$
[$\leq$-Mon]: $\forall\, \underline{x}, \underline{x}' \in (\Sigma_?{}^*)^{\underline{m}}$ , $\underline{x} \leq \underline{x}' \;\Rightarrow\; \Delta(\underline{x}) \leq \Delta(\underline{x}')$
[$\subseteq$-Mon]: $\forall\, \underline{x}, \underline{x}' \in (\Sigma_?{}^*)^{\underline{m}}$ , $\underline{x} \subseteq \underline{x}' \;\Rightarrow\; \Delta(\underline{x}) \subseteq \Delta(\underline{x}')$
[Fixed-Point]: $\forall\, \underline{x} \in (\Sigma_?{}^*)^{\underline{m}}$ , $\forall\, i \in \{1..n\}$ , $[\,\tau_i(\Delta)\,](\underline{x}) = \Delta(\underline{x})_i$ , where the left-hand-side is simply the expansion of the $Y_i$ definition, substituting: $\Delta(\underline{x})_k$ for: $Y_k(\underline{x})$ .

[LP] is clear from the definition of $\Delta$, since for empty input we return the empty string, and for each additional input character, we concatenate one extra character. Formally, [LP] is a trivial (and hence skipped) structural induction on $\underline{x}$ .

$$[[]]_{LP}$$

[$\leq$-Mon] is similarly easy, since to compute $\Delta(\underline{x}.\underline{u})$ we take $\Delta(\underline{x})$ and append "something" (a character). Therefore $\Delta(\underline{x}) \leq \Delta(\underline{x}.\underline{u})$ . And since $\underline{x} \leq \underline{x}' \iff \exists\, \underline{z} \mid \underline{x}' = \underline{x} . \underline{z}$ , a trivial structural induction on $\underline{z}$ yields [$\leq$-Mon] as originally stated.

$$[[]]_{\leq\text{-Mon}}$$

For [$\subseteq$-Mon] we first prove that $\delta$ is $\subseteq$-Monotonic (in its string arguments), which requires a well-founded induction on the R-cut-predecessor relation on the line argument, corresponding to $\delta$'s recursive definition. Once this is done we can prove that $\Delta$ is $\subseteq$-Monotonic by a simple structural induction on $\underline{x}$ .

$\delta$ is $\subseteq$-Monotonic:

Let $\underline{y},\underline{y}' \in (\Sigma_?^*)^{\underline{n}}$ , $\underline{x},\underline{x}' \in (\Sigma_?^*)^{\underline{m}}$ , $\underline{v},\underline{v}' \in (\Sigma_?)^{\underline{m}}$ .
Assume $\underline{y} \subseteq \underline{y}' \wedge \underline{x} \subseteq \underline{x}' \wedge \underline{v} \subseteq \underline{v}'$ .
Let $i \in \{1..n\}$ arbitrary ,

If $Y_i$ is a register-line: $Y_i \leftarrow R_a(Y_k)$ then:
We have $\delta(\underline{y},\underline{x},\underline{v})_i =$ if $\underline{y}_k = \varepsilon$ then a else last$(\underline{y}_k)$ [[ def. $\delta$, 3.19 ]]
and $\delta(\underline{y}',\underline{x}',\underline{v}')_i =$ if $\underline{y}'_k = \varepsilon$ then a else last$(\underline{y}'_k)$ [[ def. $\delta$, 3.19 ]]
and $\underline{y}_k = \varepsilon$ <=> $\underline{y}'_k = \varepsilon$ [[ $\underline{y} \subseteq \underline{y}'$ hyp. and def. $\subseteq$ , 2.38 ]]
and $a \subseteq a$ [[ def. $\subseteq$ , 2.38 ]]
and last$(\underline{y}_k) \subseteq$ last$(\underline{y}'_k)$ [[ $\underline{y} \subseteq \underline{y}'$ hyp. and last() $\subseteq$-Monotonic ]]
$\therefore$ $\delta(\underline{y},\underline{x},\underline{v})_i \subseteq \delta(\underline{y}',\underline{x}',\underline{v}')_i$

If $Y_i$ is a register-line: $Y_i \leftarrow R_a(x_k)$ then:
exactly the same reasoning as above with x instead of y yields:
$\therefore$ $\delta(\underline{y},\underline{x},\underline{v})_i \subseteq \delta(\underline{y}',\underline{x}',\underline{v}')_i$

If $Y_i$ is a combinational-line: $Y_i \leftarrow f^*(..,Y_k \text{ or } x_k,..)$ then:
We have $\delta(\underline{y},\underline{x},\underline{v})_i = f(..,\delta(\underline{y},\underline{x},\underline{v})_k \text{ or } \underline{v}_k,..)$ [[ def. $\delta$, 3.19 ]]
and $\delta(\underline{y}',\underline{x}',\underline{v}')_i = f(..,\delta(\underline{y}',\underline{x}',\underline{v}')_k \text{ or } \underline{v}'_k,..)$ [[ def. $\delta$, 3.19 ]]
and $\delta(\underline{y},\underline{x},\underline{v})_k \subseteq \delta(\underline{y}',\underline{x}',\underline{v}')_k$ [[ induction hyp.: $k <_{\text{R-cut-predecessor}} i$ ]]
and $\underline{v}_k \subseteq \underline{v}'_k$ [[ $\underline{v} \subseteq \underline{v}'$ hyp. ]]
and $f$ $\subseteq$-Monotonic [[ def. of the meaning of a Sysd, 3.7 ]]
$\therefore$ $f(..,\delta(\underline{y}',\underline{x}',\underline{v}')_k \text{ or } \underline{v}'_k,..) \subseteq f(..,\delta(\underline{y}',\underline{x}',\underline{v}')_k \text{ or } \underline{v}'_k,..)$
$\therefore$ $\delta(\underline{y},\underline{x},\underline{v})_i \subseteq \delta(\underline{y}',\underline{x}',\underline{v}')_i$

[[]] $_{\delta \subseteq\text{-Monotonic}}$

Now we prove [ $\subseteq$-Mon ] by structural induction on $\underline{x}$ :

Case $\underline{\varepsilon}$: Let $\underline{x}'$ arbitrary | $\underline{x} \subseteq \underline{x}'$ ,
We have $\underline{\varepsilon} \subseteq \underline{x}'$ => $\underline{\varepsilon} = \underline{x}'$ [[ def. $\subseteq$ , 2.38 ]]
and $\underline{\varepsilon} = \underline{x}'$ => $\Delta(\underline{\varepsilon}) = \Delta(\underline{x}')$ => $\Delta(\underline{\varepsilon}) \subseteq \Delta(\underline{x}')$

[[]] $_{\subseteq\text{-Mon},\underline{\varepsilon}}$

Case $(\underline{x}.\underline{u})$: Let $\underline{x}'.\underline{u}'$ arbitrary | $\underline{x}.\underline{u} \subseteq \underline{x}'.\underline{u}'$ ,
note: $\underline{x}.\underline{u} \subseteq \underline{y}$ => $|\underline{x}.\underline{u}| = |\underline{y}|$ => $\exists\, \underline{x}',\underline{u}' \mid \underline{y} = \underline{x}'.\underline{u}' \wedge \underline{x} \subseteq \underline{x}' \wedge \underline{u} \subseteq \underline{u}'$
[[ def. $\subseteq$, 2.38 ]] ]]
We have $\Delta(\underline{x}.\underline{u}) = \Delta(\underline{x}) . \delta(\Delta(\underline{x}),\underline{x},\underline{u})$ [[ simplified $\Delta$, thm. 3.20 ]]
and $\Delta(\underline{x}'.\underline{u}') = \Delta(\underline{x}') . \delta(\Delta(\underline{x}'),\underline{x}',\underline{u}')$ [[ simplified $\Delta$, thm. 3.20 ]]
and $\Delta(\underline{x}) \subseteq \Delta(\underline{x}')$ [[ induction hypothesis, $\underline{x} \subseteq \underline{x}'$ ]]
$\therefore$ $\delta(\Delta(\underline{x}),\underline{x},\underline{u}) \subseteq \delta(\Delta(\underline{x}'),\underline{x}',\underline{u}')$ [[ $\delta$ $\subseteq$-Monotonic, $\underline{x} \subseteq \underline{x}'$ , $\underline{u} \subseteq \underline{u}'$ ]]
$\therefore$ $\Delta(\underline{x}.\underline{u}) \subseteq \Delta(\underline{x}'.\underline{u}')$

[[]] $_{\subseteq\text{-Mon},\underline{x}.\underline{u}}$
[[]] $_{\subseteq\text{-Mon}}$

We finally prove the main result: [Fixed-Point] , by structural induction on $\underline{x}$, combined with much equation pushing...

Case $(\underline{\varepsilon})$: let $i \in \{1..n\}$ arbitrary.

We have $\Delta(\underline{\varepsilon})_i = \varepsilon$ [[ def. $\Delta$, 3.19 ]]

and $f^*(\underline{\varepsilon}) = \varepsilon \wedge R_a(\varepsilon) = \varepsilon$ [[ def. $f^*$, $R_a$, 3.4 ]]

$\therefore \quad [\ \tau_i(\Delta)\ ](\underline{\varepsilon}) = \varepsilon = \Delta(\underline{\varepsilon})$

$[[]]_{\text{Fixed-Point},\underline{\varepsilon}}$

Case $(\underline{x}.\underline{u})$: let $i \in \{1..n\}$ arbitrary.

We have $\Delta(\underline{x}.\underline{u})_i = \Delta(\underline{x})_i\ .\ \delta(\Delta(\underline{x}),\underline{x},u)_i$ [[ simplified $\Delta$, thm. 3.20 ]]

If $Y_i$ is a register-line: $Y_i \leftarrow R_a(Y_k \text{ or } x_k)$ then:

We have $\delta(\Delta(\underline{x}),\underline{x},u)_i = [\text{ if } \Delta(\underline{x})_k \text{ or } \underline{x}_k = \varepsilon \text{ then a else last}(\Delta(\underline{x})_k \text{ or } \underline{x}_k)\ ]$ [[ def. $\delta$, 3.19 ]]

$\therefore$ L1: $\Delta(\underline{x}.\underline{u})_i = \Delta(\underline{x})_i\ .\ [\text{ if } \Delta(\underline{x})_k \text{ or } \underline{x}_k = \varepsilon \text{ then a else last}(\Delta(\underline{x})_k \text{ or } \underline{x}_k)\ ]$

and $\Delta(\underline{x})_i = [\ \tau_i(\Delta)\ ](\underline{x})$ [[induction hypothesis ]]

$\therefore \quad \Delta(\underline{x})_i = R_a(\Delta(\underline{x})_k \text{ or } \underline{x}_k)$ [[ expanding def. $\tau_i$ ]]

$\therefore \quad \Delta(\underline{x})_i = [\text{ if } \Delta(\underline{x})_k \text{ or } \underline{x}_k = \varepsilon \text{ then } \varepsilon \text{ else a . abl}(\Delta(\underline{x})_k \text{ or } \underline{x}_k)\ ]$ [[ expanding $R_a$ ]]

$\therefore \quad \Delta(\underline{x}.\underline{u})_i = [\text{ if } \Delta(\underline{x})_k \text{ or } \underline{x}_k = \varepsilon \text{ then } \varepsilon \text{ . a else a . abl}(\Delta(\underline{x})_k \text{ or } \underline{x}_k)\text{ . last}(\Delta(\underline{x})_k \text{ or } \underline{x}_k)\ ]$ [[ replacing $\Delta(\underline{x})_i$ in line L1 ]]

$\therefore \quad \Delta(\underline{x}.\underline{u})_i = [\text{ if } \Delta(\underline{x})_k = \varepsilon \text{ then a else a . }(\Delta(\underline{x})_k \text{ or } \underline{x}_k)\ ]$ [[ simplifying abl().last() ]]

$\therefore$ L2: $\Delta(\underline{x}.\underline{u})_i = \text{a . }(\Delta(\underline{x})_k \text{ or } \underline{x}_k)$ [[ simplifying if expression ]]

We have $[\ \tau_i(\Delta)\ ](\underline{x}.\underline{u}) = R_a(\Delta(\underline{x}.\underline{u})_k \text{ or } \underline{x}_k.\underline{u}_k)$ [[ expanding def. $\tau_i$ ]]

$\therefore \quad [\ \tau_i(\Delta)\ ](\underline{x}.\underline{u}) = R_a[\ \Delta(\underline{x})_k\ .\ \delta(\Delta(\underline{x}),\underline{x},u)_k \text{ or } \underline{x}_k.\underline{u}_k\ ]$ [[ expanding $\Delta(\underline{x}.\underline{u})$ , thm. 3.20 ]]

$\therefore \quad [\ \tau_i(\Delta)\ ](\underline{x}.\underline{u}) = \text{a . }(\Delta(\underline{x})_k \text{ or } \underline{x}_k)$ [[ expanding $R_a$, $\delta$(..) and $\underline{u}_k$ are characters. ]]

$\therefore \quad [\ \tau_i(\Delta)\ ](\underline{x}.\underline{u}) = \Delta(\underline{x}.\underline{u})_i$ [[ *matching with line L2* ]]

$[[]]_{\text{Fixed-Point},\underline{x}.\underline{u},\text{Register}}$

If $Y_i$ is a combinational-line: $Y_i \leftarrow f^*(..,Y_k \text{ or } x_k,..)$ then:

We have $\delta(\Delta(\underline{x}),\underline{x},u)_i = f\,(..\text{last}(\Delta(\underline{x}.\underline{u})_k \text{ or } \underline{x}_k.\underline{u}_k),..)$ [[ def. $\delta$, 3.19 ]]

$\therefore$ L3: $\Delta(\underline{x}.\underline{u})_i = \Delta(\underline{x})_i\ .\ f\,(..\text{last}(\Delta(\underline{x}.\underline{u})_k \text{ or } \underline{x}_k.\underline{u}_k),..)$

and $\Delta(\underline{x})_i = [\ \tau_i(\Delta)\ ](\underline{x})$ [[ induction hypothesis ]]

$\therefore \quad \Delta(\underline{x})_i = f^*(..,\Delta(\underline{x})_k \text{ or } \underline{x}_k,..)$ [[ expanding def. $\tau_i$ ]]

$\therefore \quad \Delta(\underline{x}.\underline{u})_i = f^*(..,\Delta(\underline{x})_k \text{ or } \underline{x}_k,..)\ .\ f\,(..\text{last}(\Delta(\underline{x}.\underline{u})_k \text{ or } \underline{x}_k.\underline{u}_k),..)$ [[ combining with line L3 ]]

$\therefore \quad \Delta(\underline{x}.\underline{u})_i = f^*(..,\Delta(\underline{x})_k.\text{last}(\Delta(\underline{x}.\underline{u})_k) \text{ or } \underline{x}_k.\text{last}(\underline{x}_k.\underline{u}_k),..)$ [[ def. $f^*$ ]]

$\therefore$ L4: $\Delta(\underline{x}.\underline{u})_i = f^*(..,\Delta(\underline{x})_k.\text{last}(\Delta(\underline{x}.\underline{u})_k) \text{ or } \underline{x}_k.\underline{u}_k,..)$ [[ simplifying $\underline{x}_k.\text{last}(\underline{x}_k.\underline{u}_k)$ ]]

and $\Delta(\underline{x}.\underline{u})_k = \Delta(\underline{x})_k\ .\ \delta(\Delta(\underline{x}),\underline{x},u)_k$ [[ thm. 3.20 ]]

$\therefore \quad \Delta(\underline{x}.\underline{u})_k = \Delta(\underline{x})_k\ .\ \text{last}(\Delta(\underline{x}.\underline{u})_k)$ [[ $\delta$(...) is a character! ]]

$\therefore \quad \Delta(\underline{x}.\underline{u})_i = f^*(..,\Delta(\underline{x}.\underline{u})_k \text{ or } \underline{x}_k.\underline{u}_k,..)$ [[ substituting into L4 ]]

and $[\ \tau_i(\Delta)\ ](\underline{x}.\underline{u}) = f^*(..,\Delta(\underline{x}.\underline{u})_k \text{ or } \underline{x}_k.\underline{u}_k,..)$ [[ expanding def. $\tau_i$ ]]

$\therefore \quad [\ \tau_i(\Delta)\ ](\underline{x}.\underline{u}) = \Delta(\underline{x}.\underline{u})_i$

$[[]]_{\text{Fixed-Point},\underline{x}.\underline{u},\text{Combinational}}$

$[[]]_{\text{Fixed-Point},\underline{x}.\underline{u}}$

$[[]]_{\text{Fixed-Point}}$

From all this we know that $\Delta$ is a fixed point of $\tau_S$ and $\Delta \in \boldsymbol{MLP}_\Sigma$ .

$\therefore \quad \text{LFP}(\tau_S) \subseteq \Delta$ [[ LFP is Least! , def. 2.13 ]]

$\therefore \quad \forall\ \underline{x} \in (\Sigma_?^*)^{\underline{m}}\ ,\ \text{LFP}(\tau_S)(\underline{x}) \subseteq \Delta(\underline{x})$ [[ def. pointwise order, 2.23 ]]

From the previous section (section 3.4) and ELC(S) hypothesis :

We have LFP($\tau_S$) total on $\Sigma^*$ [[ ELC thm., 3.15 ]]

$\therefore$ $\forall\ \underline{x} \in (\Sigma^*)^{\underline{m}}$, LFP($\tau_S$)($\underline{x}$) $\in (\Sigma^*)^{\underline{n}}$

and strings with no ? in them are maximal under $\subseteq$ [[ def. $\subseteq$ coordinatewise ]]

$\therefore$ $\forall\ \underline{x} \in (\Sigma^*)^{\underline{m}}$, LFP($\tau_S$)($\underline{x}$) is maximal under $\subseteq$

Combining those 2 results, we get:

$\therefore$ $\forall\ \underline{x} \in (\Sigma^*)^{\underline{m}}$, LFP($\tau_S$)($\underline{x}$) = $\Delta(\underline{x})$

and of course the equality still holds if we project some lines (out) from the tuple:

and $\mu(S) = \mathrm{LFP}(\tau_S)_{out}$ [[ def. 3.8 ]]

$\therefore$ $\forall\ \underline{x} \in (\Sigma^*)^{\underline{m}}$, $\Delta(\underline{x})_{out} = \mu(S)(\underline{x})$ .

$[[]]_{\text{Thm. } 3.21}$

We now move on to our simulation semantics. We will define it both informally and formally, and then prove its equivalence with the operational semantics (and therefore also to the extensional denotational semantics).

**Definition 3.22: Informal Simulation Semantics**

The main difference with the operational semantics is that now the state simply contains the current value stored in each register. We call it $s_R$ and it is indexed by the (Register) line number.

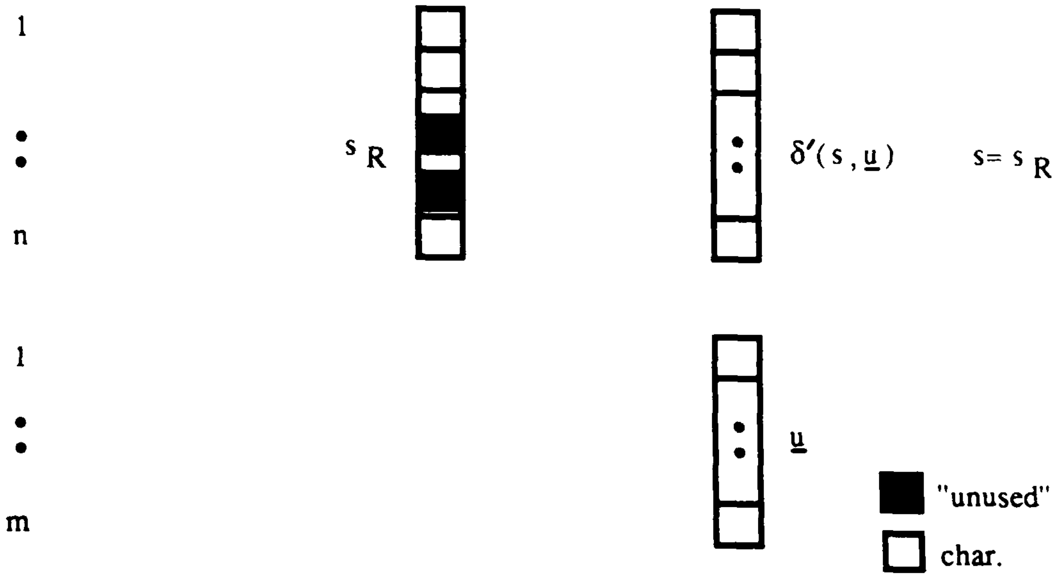
The new "next-output" function $\delta'_S$ differs from the old one in the Register case only and simply returns the character in $s_{R_i}$ for Register-line $Y_i$.

The new "next-state" function $\gamma'_S$ updates $s_R$ by storing in it the character just output by $\delta'_S$ for its predecessor line (or the input character if the argument is an input-line).

The extensions of these functions to handle strings of inputs are done just as in the previous case, by iterating the character by character functions. One detail is different however: the initial state is taken from S, i.e. if S contains the equation $Y_i \leftarrow R_a(Y_k)$ then the initial state has $s_{Rinitial_i} = a$ .

Pictorially, the set-up looks like this:

**Figure 3-4:** Simulation Semantics



As before, the S subscript will be omitted. Note also that we define $s_R$ to be an array of length n, indexed by the line number i, when in fact we only use array slots corresponding to Register-lines. This is just for ease of notation. The other entries can be thought of as "unspecified" or containing an "unused" character, and are irrelevant to the proof.

**Definition 3.23: Formal Simulation Semantics**
Let $S \in L_{SD}$, with non-input lines $Y_i$, $i \in \{1..n\}$ and input lines $x_j$, $j \in \{1..m\}$, and ELC(S).

Let $s_R \in (\Sigma_?)^{\underline{n}}$, $\underline{v} \in (\Sigma_?)^{\underline{m}}$.

Define $\delta'(s_R,\underline{v}) \in (\Sigma_?)^{\underline{n}} \mid \forall i \in \{1..n\}$

- if $Y_i \leftarrow R_a(Y_k \text{ or } x_k)$ then $\delta'(s_R,\underline{v})_i = s_{R_i}$
- if $Y_i \leftarrow f^*(..,Y_k \text{ or } x_k,..)$ then $\delta'(s_R,\underline{v})_i = f(..,\delta'(s_R,\underline{v})_k \text{ or } \underline{v}_k,..)$

Define $\gamma'(s_R,\underline{v}) \mid \forall i \in \{1..n\}$

- if $Y_i \leftarrow R_a(Y_k \text{ or } x_k)$ then $\gamma'(s_R,\underline{v})_i = \delta'(s_R,\underline{v})_k \text{ or } \underline{v}_k$

And the string-extended functions are defined by recursion on the input string:

$\Delta'(\underline{\varepsilon}) = \underline{\varepsilon}$ and $\Delta'(\underline{x}.\underline{u}) = \Delta'(\underline{x}) \, . \, \delta'(\Gamma'(\underline{x}),\underline{u})$

$\Gamma'(\underline{\varepsilon})_i = S_{Rinitial_i} =$ if $Y_i \leftarrow R_a(Y_k \text{ or } x_k)$ then a  and $\Gamma'(\underline{x}.\underline{u}) = \gamma'(\Gamma'(\underline{x}),\underline{u})$

The justification for the totality of these functions is the same as for the operational semantics. The key result is:

**Theorem 3.24: Simulation-Operational Equivalence**
Let S be an ELC sysd (with *m* inputs), we have: $\forall \underline{x} \in (\Sigma_?{}^*)^{\underline{m}}$, $\Delta'_S(\underline{x}) = \Delta_S(\underline{x})$ .

Or in other words: the two operational semantics agree.

The proof proceeds in two steps:

1. A "small state is appropriate" lemma, which makes explicit the fact that the value currently kept in the register is the same as the last character seen on the predecessor line, and which is proved by structural induction on the input string .
2. An inductive proof of equality between $\Delta$ and $\Delta'$. The main subtlety here is to find an induction which proceeds in the same manner as $\Delta$ or $\Delta'$ recurses, i.e. a combination of structural recursion on the input, and R-cut-predecessor recursion on the lines. To achieve that we define $<_{lex}$ : the lexicographic combination of the prefix ordering on strings, and the R-cut-predecessor ordering on the lines of an ELC circuit, and use well-founded induction on $<_{lex}$ .

Once these steps have been identified, what remains is tedious equation pushing...

[State-Lemma]: $\forall\ \underline{x} \in (\Sigma_\gamma{}^*)^{\underline{m}}$, $\forall\ i \in \{1..n\}$ , if $Y_i \leftarrow R_a(Y_k \text{ or } x_k)$ then
$\Gamma'(\underline{x})_i = \text{if } (\Delta'(\underline{x})_k \text{ or } \underline{x}_k) = \varepsilon \text{ then a else last}(\Delta'(\underline{x})_k \text{ or } \underline{x}_k)$

This is proved by a simple structural induction on $\underline{x}$ :

Case $\underline{\varepsilon}$ :

Let $\quad i \in \{1..n\} \mid \text{if } Y_i \leftarrow R_a(Y_k \text{ or } x_k)$

then $\quad \Gamma'(\underline{\varepsilon})_i = a$ $\qquad$ [[ def. $\Gamma'$, 3.23 ]]

and $\quad \Delta'(\underline{\varepsilon}) = \underline{\varepsilon}$ $\qquad$ [[ def. $\Delta'$, 3.23 ]]

$\therefore \quad \Gamma'(\varepsilon)_i = \text{if } \underline{\varepsilon} = \underline{\varepsilon} \text{ then a else } \ldots$

$[[]]_{\text{State-Lemma},\underline{\varepsilon}}$

Case $\underline{x}.\underline{u}$:

Let $\quad i \in \{1..n\} \mid \text{if } Y_i \leftarrow R_a(Y_k \text{ or } x_k)$

then $\quad \Gamma'(\underline{x}.\underline{u})_i = \gamma'(\Gamma'(\underline{x}),\underline{u})$ $\qquad$ [[ def. 3.23, expanding $\Gamma'$ ]]

$\therefore$ L1: $\Gamma'(\underline{x}.\underline{u})_i = \delta'(\Gamma'(\underline{x}),\underline{u})_k \text{ or } \underline{u}_k$ $\qquad$ [[ def. 3.23, expanding $\gamma'$ ]]

and $\quad \Delta'(\underline{x}.\underline{u})_k = \Delta'(\underline{x})_k.\delta'(\Gamma'(\underline{x}),\underline{u})_k$ $\qquad$ [[ def. 3.23, expanding $\Delta'$ ]]

$\therefore \quad \text{last}(\Delta'(\underline{x}.\underline{u})_k) = \delta'(\Gamma'(\underline{x}),\underline{u})_k \ \wedge\ \Delta'(\underline{x}.\underline{u})_k \neq \varepsilon$

$\therefore \quad \Gamma'(\underline{x}.\underline{u})_i = \text{last}(\Delta'(\underline{x}.\underline{u})_k) \text{ or } \underline{u}_k$ $\qquad$ [[ replacing in L1 ]]

and $\quad \underline{u}_k = \text{last}(\underline{x}_k.\underline{u}_k) \ \wedge\ \underline{x}_k.\underline{u}_k \neq \varepsilon$

$\therefore \quad \Gamma'(\underline{x}.\underline{u})_i = \text{last}(\Delta'(\underline{x}.\underline{u})_k \text{ or } \underline{x}_k.\underline{u}_k)$

$\therefore \quad \Gamma'(\underline{x}.\underline{u})_i = \text{if } (\Delta'(\underline{x}.\underline{u})_k \text{ or } \underline{x}_k.\underline{u}_k) = \varepsilon \text{ then} \ldots \text{ else last}(\Delta'(\underline{x}.\underline{u})_k \text{ or } \underline{x}_k.\underline{u}_k)$

$[[]]_{\text{State-Lemma},\underline{x}.\underline{u}}$

$[[]]_{\text{State-Lemma}}$

We now prove the final equivalence: $\forall\ \underline{x} \in (\Sigma_\gamma{}^*)^{\underline{m}}$, $\forall\ i \in \{1..n\}$ , $\Delta'(\underline{x})_i = \Delta(\underline{x})_i$ , by well-founded induction on $<_{lex}(\underline{x},i)$ :

Case $(\underline{\varepsilon},i)$:

We have $\quad \Delta(\underline{\varepsilon})_i = \varepsilon = \Delta'(\underline{\varepsilon})_i$ $\qquad$ [[ def. $\Delta$, 3.19 and def. $\Delta'$, 3.23 ]]

$[[]]_{\underline{\varepsilon},i}$

Case $(\underline{x}.\underline{u},i)$:

We have $\quad \Delta(\underline{x}.\underline{u})_i = \Delta(\underline{x})_i.\delta(\Delta(\underline{x}),\underline{x},u)_i$ $\qquad$ [[ expanding $\Delta$, thm. 3.20 ]]

and $\quad \Delta'(\underline{x}.\underline{u})_i = \Delta'(\underline{x})_i.\delta'(\Gamma'(\underline{x}),\underline{u})_i$ $\qquad$ [[ def. $\Delta'$, 3.23 ]]

and $\quad \Delta(\underline{x})_i = \Delta'(\underline{x})_i$ $\qquad$ [[ $(\underline{x},i) <_{lex} (\underline{x}.\underline{u},i)$, induction hyp. ]]

$\therefore \quad$ only $\delta(\Delta(\underline{x}),\underline{x},u)_i = \delta'(\Gamma'(\underline{x}),\underline{u})_i$ remains to be proved.

if $Y_i \leftarrow R_a(Y_k$ or $x_k)$ then

We have $\delta(\Delta(\underline{x}).\underline{x}.\underline{u})_i = $ if $(\Delta(\underline{x})_k$ or $\underline{x}_k) = \varepsilon$ then a else last$(\Delta(\underline{x})_k$ or $\underline{x}_k)$

[[ def. $\delta$, 3.19 ]]

and $\delta'(\Gamma'(\underline{x}).\underline{u})_i = \Gamma'(\underline{x})_i = $ if $(\Delta'(\underline{x})_k$ or $\underline{x}_k) = \varepsilon$ then a else last$(\Delta'(\underline{x})_k$ or $\underline{x}_k)$

[[ def. $\delta'$, 3.23 and State-Lemma ]]

and $\Delta(\underline{x})_k = \Delta'(\underline{x})_k$ [[ $(\underline{x}.k) <_{lex} (\underline{x}.\underline{u}.k)$, induction hyp. ]]

$\therefore$ $\delta(\Delta(\underline{x}).\underline{x}.u)_i = \delta'(\Gamma'(\underline{x}).\underline{u})_i$

$[[]]_{\underline{x}.\underline{u}.i.\text{Register}}$

If $Y_i \leftarrow f^*(\ldots Y_k$ or $x_k,\ldots)$ then

We have $\delta(\Delta(\underline{x}).\underline{x}.\underline{u})_i = f(\ldots\delta(\Delta(\underline{x}).\underline{x}.\underline{u})_k$ or $\underline{u}_k,\ldots)$ [[ def. $\delta$, 3.19 ]]

and $\delta'(\Gamma'(\underline{x}).\underline{u})_i = f(\ldots,\delta'(\Gamma'(\underline{x}).\underline{u})_k$ or $\underline{u}_k,\ldots)$ [[ def. $\delta'$, 3.23 ]]

and $\Delta(\underline{x}.\underline{u})_k = \Delta'(\underline{x}.\underline{u})_k$ [[ $(\underline{x}.\underline{u}.k) <_{lex} (\underline{x}.\underline{u}.i)$, induction hyp. ]]

and $\Delta(\underline{x}.\underline{u})_k = \Delta(\underline{x})_k.\delta(\Delta(\underline{x}).\underline{x}.u)_k$ [[ expanding $\Delta$, thm. 3.20 ]]

and $\Delta'(\underline{x}.\underline{u})_k = \Delta'(\underline{x})_k.\delta'(\Gamma'(\underline{x}).\underline{u})_k$ [[ def. $\Delta'$, 3.23 ]]

$\therefore$ $\delta(\Delta(\underline{x}).\underline{x}.u)_k = \delta'(\Gamma'(\underline{x}).\underline{u})_k$

$\therefore$ $f(\ldots\delta(\Delta(\underline{x}).\underline{x}.\underline{u})_k$ or $\underline{u}_k,\ldots) = f(\ldots,\delta'(\Gamma'(\underline{x}).\underline{u})_k$ or $\underline{u}_k,\ldots)$

$\therefore$ $\delta(\Delta(\underline{x}).\underline{x}.u)_i = \delta'(\Gamma'(\underline{x}).\underline{u})_i$

$[[]]_{\underline{x}.\underline{u}.i.\text{Combinational}}$

$[[]]_{\underline{x}.\underline{u}.i}$

$[[]]_{\text{Thm. 3.24}}$

# 4. Theoretical Applications of the Semantics

## 4.1. The MLP-calculus

In this section we develop the theory of MLP string-functions, in order to provide some basic tools for the theoretical and practical manipulations of sysd's. The following list of theorems only includes those which we have found useful in our current investigations of mechanical SYSD equivalence proofs. It is only intended as the beginning of a calculus.

**Theorem 4.1: Composition of $f^*$ 's**
Let f,g be character-functions, $(f \circ g)^* = f^* \circ g^*$ .

**Proof:**
Immediate

$[[]]_{\text{Thm. } 4.1}$

The following property is an essential characteristic of combinational functions (which will often be used in mechanical proofs of equivalence of sysd's):

**Theorem 4.2: Combinational-Concatenation Commutativity [CCC]**
Let $f^* : (\Sigma_?^*)^{\underline{n}} \rightarrow \Sigma_?^*$ , $\forall\, \underline{x}, \underline{y} \in (\Sigma_?^*)^{\underline{n}}$ , $f^*(\underline{x} . \underline{y}) = f^*(\underline{x}) . f^*(\underline{y})$ .

**Proof:**
$f^*$ was defined as the homomorphic extension of a character-function f to strings (of same length), therefore this property is immediate.

$[[]]_{\text{Thm. } 4.2}$

We now define the "extended register" function: $R_z$ . Intuitively, $R_z$ outputs z first, and then x, up to a total number of characters equal to the number of characters in the input. The else clause consists of the (uninteresting) case where the input is of smaller length than z.

**Definition 4.3: $R_z$**
Let $z{\downarrow}_{1..k} \in \Sigma_?^*$, define $R_z: \Sigma_?^* \rightarrow \Sigma_?^*$ by: $R_z(x{\downarrow}_{1..n})$ = if $n > k$ then $z{\downarrow}_{1..k}x{\downarrow}_{1..n-k}$ else $z{\downarrow}_{1..n}$

It is immediate that $R_z$ is MLP.

Note that we are abusing the notation slightly in the case where z=a, since the extended $R_a$ is unary, and the original $R_a$ is binary. The confusion is harmless, since the binary $R_a$ ignores its second input ($x_{ck}$), so all algebraic properties of one will carry to the other. In the rest of this section, we intend the unary $R_a$ .

**Theorem 4.4: Composition of $R_z$ 's**
$\forall\, z,z' \in \Sigma_?^*$ , $R_{z'} \circ R_z = R_{z'z}$ .

**Proof:**
Let $z = z{\downarrow}_{1..i}$ , $z' = z'{\downarrow}_{1..j}$ , $x \in \Sigma_?^*$, arbitrary, $x = x{\downarrow}_{1..n}$ .
The proof has 3 cases: $n > i+j$ , $n \leq i$ , $i < n \leq i+j$ . The most general one is $n > i+j$ (i.e. steady state) and it is the only one we show (the others are simpler):
We have $R_{z'z}(x{\downarrow}_{1..n}) = z'{\downarrow}_{1..j}z{\downarrow}_{1..i}x{\downarrow}_{1..n-i-j}$ [[ $n > i+j$ ]]
and $R_z(x{\downarrow}_{1..n}) = z{\downarrow}_{1..i}x{\downarrow}_{1..n-i}$ [[ $n > i+j$ => $n > i$ ]]
Let $x' = R_z(x{\downarrow}_{1..n})$

We have $|x'| = n$ , $x' = x'{\downarrow}_{1..n}$ .
and $\forall k \in \{1..n\}$ , $x'{\downarrow}_k$ = if $1 \le k \le i$ then $z{\downarrow}_k$ else $x{\downarrow}_{k-i}$
and $R_{z'}(x') = z'{\downarrow}_{1..j}x'{\downarrow}_{1..n-j}$ [[ $n > i+j \;\Rightarrow\; n > j$ ]]
and $x'{\downarrow}_{1..n-j} = z{\downarrow}_{1..i}x{\downarrow}_{1..n-j-i} = z{\downarrow}_{1..i}x{\downarrow}_{1..n-i-j}$ [[ $n > i+j \;\Rightarrow\; n-j > i$ ]]
$\therefore$ $R_{z'}(R_z(x)) = z'{\downarrow}_{1..j}z{\downarrow}_{1..i}x{\downarrow}_{1..n-i-j} = R_{z'z}(x)$

$[[]]_{\text{Thm. 4.4}}$

The next property is the essence of the "is-a-pipeline-of" relation which we will define later, in section 4.2 .

**Theorem 4.5: $R_z$ pipeline**
$\forall z, z', x \in \Sigma_?^*$ , if $|z'| = |z|$ then $R_z(xz') = zx$ .

**Proof:**
Immediate verification.

$[[]]_{\text{Thm. 4.5}}$

Finally, this next property is an essential characteiistic of MLP functions in general (which will be key in mechanical proofs of equivalence of sysd's):

**Theorem 4.6: Register-MLP**
Let F: $(\Sigma_?^*)^{\underline{n}} \to \Sigma_?^*$, MLP string-function, $a \in \Sigma_?$, $\forall \underline{x} \in (\Sigma_?^*)^{\underline{n}}$ , $\forall \underline{u} \in (\Sigma_?)^{\underline{n}}$ ,
$R_a(F(\underline{x} . \underline{u})) = a . F(\underline{x})$ .

The proof relies on the following lemma, which is interesting in its own right:

**Theorem 4.7: 1st-order characterization of MLP string-functions**
Let F be a (unary) function: $\Sigma_?^* \to \Sigma_?^*$ , F is MLP <=> $F(\varepsilon) = \varepsilon \wedge \forall x \in \Sigma_?^*, \forall u \in \Sigma_?, \exists v \in \Sigma_? \mid F(x.u) = F(x).v$ .

**Proof:**

=>
Assume $F : \Sigma_?^* \to \Sigma_?^*$, MLP string-function.
We have $|F(\varepsilon)| = |\varepsilon|$ [[ F is length-preserving ]]
$\therefore$ $|F(\varepsilon)| = 0$ [[ property of length ]]
$\therefore$ $F(\varepsilon) = \varepsilon$ [[ property of length ]]

Assume $x \in \Sigma_?^*$ , $u \in \Sigma_?$ ,
We have $F(x) \le F(x.u)$ [[ F is monotonic ]]
$\therefore$ $\exists y \in \Sigma_?^* \mid F(x.u) = F(x).y$ [[ thm. 2.43, 2nd def. of prefix ]]
$\therefore$ $|F(x).y| = |F(x.u)| = |x.u|$ [[ F is length-preserving ]]
$\therefore$ $|F(x)| + |y| = |x| + 1$ [[ properties of length ]]
and $|F(x)| = |x|$ [[ F is length-preserving ]]
$\therefore$ $|y| = 1$
$\therefore$ $y \in \Sigma_?$

$[[]]_{\Rightarrow}$

<=
Assume $F : \Sigma_?^* \to \Sigma_?^* \mid$ [h1] $F(\varepsilon) = \varepsilon \wedge$ [h2] $\forall x \in \Sigma_?^*, \forall u \in \Sigma_?, \exists v \in \Sigma_? \mid F(x.u) = F(x).v$ .

Let $x,y \in \Sigma_?^* \mid x \leq y$

then $\exists\, z \in \Sigma_?^* \mid y = x.z$ [[ thm. 2.43, 2nd def. of prefix ]]

We prove by induction on z that $\forall\, z \in \Sigma_?^*$ , $F(x) \leq F(x.z)$ :

- Base case: $z = \varepsilon$,

then $x = x.z$ [[ $x.\varepsilon = x$ , $\forall\, x \in \Sigma_?^*$ ]]

$\therefore$ $F(x) = F(x.z)$ [[ F function! ]]

$\therefore$ $F(x) \leq F(x.z)$ [[ $\leq$ reflexive ]]

- Induction step: assume that $F(x) \leq F(x.z)$, consider $x.(z.u)$ for some $u \in \Sigma_?$ :

We have $x.(z.u) = (x.z).u$ [[ definition of concatenation ]]

$\therefore$ [c1] $F[(x.z).u] = F(x.z).v$ for some $v \in \Sigma_?$ [[ h2 ]]

and $F(x) \leq F(x.z)$ [[ induction hypothesis ]]

and $F(x.z) \leq F(x.z).v$ [[ definition of $\leq$ ]]

$\therefore$ $F(x) \leq F(x.z).v$ [[ transitivity of $\leq$ ]]

$\therefore$ $F(x) \leq F[x.(z.u)]$ [[ c1 ]]

$[[]]_{\text{F monotonic}}$

We now prove by induction on x that $\forall\, x \in \Sigma_?^*$ , $|F(x)| = |x|$ , i.e. F is length-preserving.

- Base case: $x = \varepsilon$,

We have $F(\varepsilon) = \varepsilon$ [[ h1 ]]

$\therefore$ $|F(\varepsilon)| = |\varepsilon|$

- Induction step:

Assume $|F(x)| = |x|$, $u \in \Sigma_?$

We have $F(x.u) = F(x).v$ for some $v \in \Sigma_?$ [[ h2 ]]

$\therefore$ $|F(x.u)| = |F(x).v| = |F(x)| + |v| = |F(x)| + 1$ [[ properties of length ]]

and $|F(x)| = |x|$ [[ induction hypothesis ]]

$\therefore$ $|F(x.u)| = |x| + 1 = |x.u|$ [[ properties of length ]]

$[[]]_{\text{F Length-Preserving}}$

$[[]]_{<=}$

$[[]]_{\text{Thm. 4.7}}$

It is clear that the => part of this lemma generalizes immediately to string-functions of any arity. (For the other direction, there is a technicality in that we have to consider the restriction of F to $(\Sigma_?^*)^{\underline{n}}$ .) Therefore, the proof of the Register-MLP theorem is now extremely simple:

Let $a \in \Sigma_?$, F MLP string-function, $\underline{x} \in (\Sigma_?^*)^{\underline{n}}$ , $\underline{u} \in (\Sigma_?)^{\underline{n}}$

We have $\exists\, v \in \Sigma_? \mid F(\underline{x} . \underline{u}) = F(\underline{x}).v$ [[ thm. 4.7, => part ]]

$\therefore$ $R_a(F(\underline{x} . \underline{u})) = R_a(F(\underline{x}).v) = a.F(\underline{x})$ [[ definition of $R_a$ ]]

$[[]]_{\text{Thm. 4.6}}$

This completes our current algebraic development of the theory of $\boldsymbol{MLP_\Sigma}$ .

## 4.2. Relations on Synchronous Circuits

A key concept in the transformational approach to design is (from [Talcott 86], and in published form in [Mason 86]):

> Operations on programs need meanings to transform and meanings to preserve.

where we replace "program" by "synchronous system" for our purposes. The study of relations on sysd's *is* the study of the various meanings we want to transform or preserve.

The following preliminary investigations are just intended to give a taste of the possibilities...

**Definition 4.8: Equivalence Relations on $L_{SD}$**
We can define 4 equivalence relations on sysd's, which are progressively coarser. Let $S_1, S_2 \in L_{SD}$,

- $S_1 = S_2$ <=> $S_1$ and $S_2$ are syntacticly identical. (Not very interesting.)
- $S_1 \hat{=} S_2$ <=> $S_1$ and $S_2$ are isomorphic (i.e. equal up to renaming of syntactic pieces).
- $S_1 \approx S_2$ <=> $[\![ S_1 ]\!] = [\![ S_2 ]\!]$. (Intensional equivalence: they denote the same functional.)
- $S_1 \equiv S_2$ <=> $\mu(S_1) = \mu(S_2)$. (Extensional equivalence: they compute the same functions.)

Note: technically, for $\equiv$, we are comparing tuples (of functions), and we compare coordinate-wise.

More generally, $\equiv$ is a particular case of the fact that for any relation on $MLP_\Sigma$ string-functions, we can define the corresponding extensional relation on $L_{SD}$ as follows:

**Definition 4.9: Induced Extensional Relation from $MLP_\Sigma$ to $L_{SD}$**
Let $\phi$ be a (n-ary) relation on functions of $MLP_\Sigma$. Define $\phi$ on $L_{SD}$ with:
$\forall S_1,\ldots,S_n \in L_{SD}$, $\phi(S_1,\ldots,S_n)$ <=> $\phi(\mu(S_1),\ldots,\mu(S_n))$.

Again, we extend $\phi$-comparison to tuples by comparing them coordinate-wise (and answering True if all comparisons are True).

One such relation which is very relevant to current digital circuit design, is the notion of a string-function being a "pipeline" of another:

**Definition 4.10: Pipeline relation on string-functions**
Let F, G be two string-functions: $\Sigma_?^* \rightarrow \Sigma_?^*$,

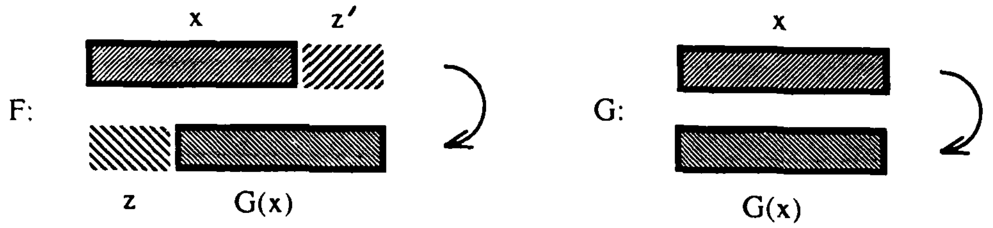
- $F \,\alpha_{z,z'}\, G$ (read "F is-a-pipeline-of G with garbage z and purge z′ ") with $z,z' \in \Sigma_?^*$ <=> $|z| = |z'|$ $\wedge$ $\forall x \in \Sigma_?^*$, $F(xz') = zG(x)$ .
- $F \,\alpha\, G$ (read "F is-a-pipeline-of G") <=> $\exists\, z,z' \in \Sigma_?^* \mid F \,\alpha_{z,z'}\, G$ .

This definition is extended in the obvious way to string-functions of same arity (> 1).

Intuitively, z is the garbage output during pipeline fill-up, and z′ is the (irrelevant) string fed in during pipeline purging.

Pictorially:

Figure 4-1: F is-a-pipeline-of G



**Theorem 4.11: α partial pre-order**
α is a partial pre-order on string-functions (i.e. reflexive and transitive) and is not antisymmetric.

**Proof:**

reflexivity: immediate (take z and z′ to be ε).

transitivity:

Assume $F \alpha_{z,z'} G$ and $G \alpha_{y,y'} H$
Let x arbitrary in $\Sigma_?^*$.
We have $G(xy') = yH(x)$ [[ G α H, instantiating x to x ]]
and $F(xy'z') = zG(xy')$ [[ F α H, instantiating x to xy′ ]]
∴ $F(xy'z') = zyH(x)$ , for arbitrary x
∴ $F \alpha_{y'z',zy} H$
∴ $F \alpha H$

α is not antisymmetric, even when restricted to MLP string-functions:
**Counter-example:**
Let

- $F(x) = 0101... \mid |F(x)| = |x|$
- $G(x) = 1010... \mid |G(x)| = |x|$

then $F \alpha_{0,a} G \wedge G \alpha_{1,b} F$, for any $a,b \in \Sigma$
and yet $F \neq G$ .

[[]]$_{\text{Thm. 4.11}}$

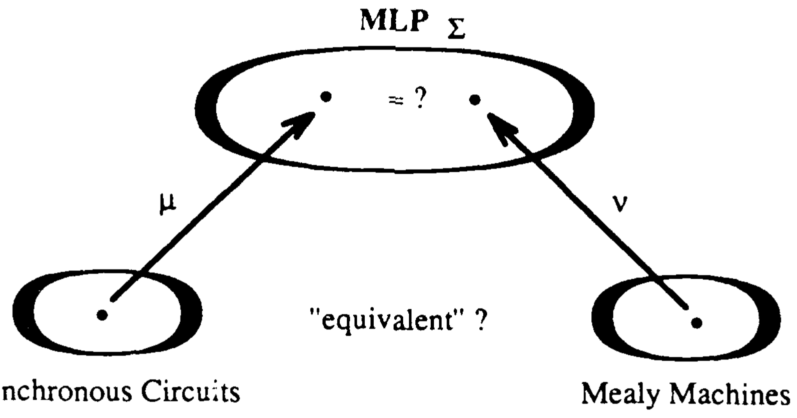
Note: this counter-example brings up the fact that the purge string mentioned in the definition of α is absolutely irrelevant. In fact, if there exists one such purge string, then any other string of the same length will do. This brings up an alternative definition of α which may be also be useful:

**Definition 4.12: Alternate pipeline**
Let F, G be two string-functions of arity 1, $F \alpha_n G$ (read "F is-a-pipeline-of G with latency n") <=>
$\exists z,z' \in \Sigma_?^* \mid |z| = |z'| = n \wedge \forall x \in \Sigma_?^*, F(xz') = zG(x)$ .

## 4.3. Relations between Synchronous Circuits and (Mealy) Sequential Machines

The key idea here is that sequential machines [Booth 67], [Hopcroft-Ullman 79] can be given string-functional semantics ($\nu$) very naturally. Once this is done, then we can use our string-functional semantics for SYSD's ($\mu$) to compare formally both objects, as shown pictorially below. We base our definitions on Mealy machines. Since Moore machines are trivially reducible to Mealy machines (without state explosion) this does not reduce the generality.

Figure 4-2: Formal Comparison of Sequential Machines and Synchronous Circuits



Note: the fact that sequential machines have associated string-functions is not new in any way! What is new is to look at these functions as an extensional characterization of the machines, and to compare them to our extensional characterization of synchronous systems. Usually, the standard theoretical development on sequential machines proceeds with an equivalence relation based on *state* equivalence, i.e. an intensional characterization.

A Mealy machine M is given as a "next-state" function $\gamma_M$ and a "next-output-character" function $\delta_M$, which both depend on the current state and current input character. We then extend these functions to take strings of inputs *exactly as we did when defining the Operational semantics of SYSDs* in section 3.5, by iterating the next-output and next-state functions. Precisely:

**Definition 4.13: String-Functional Semantics of Mealy Machines**
Let $M = \langle \Sigma, Q, q_0, \gamma, \delta \rangle$ be a Mealy Machine, with the intended interpretation:

- $\Sigma$ : alphabet (input and output)
- $Q$ : set of states
- $q_0$ : initial state
- $\gamma : Q \times \Sigma \rightarrow Q$ : next-state function
- $\delta : Q \times \Sigma \rightarrow \Sigma$ : next-output function

Define $\nu(M) = \Delta : \Sigma^* \rightarrow \Sigma^*$ where:

- $\Delta(\varepsilon) = \varepsilon \ \wedge \ \Delta(x.u) = \Delta(x) \,.\, \delta(\Gamma(x),u)$
- $\Gamma(\varepsilon) = q_0 \ \wedge \ \Gamma(x.u) = \gamma\,(\Gamma(x),u)$

The fact that $\Delta$ is MLP should be clear. Formally, the proof would be similar to the ones in section 3.5, and is not repeated.

We can now easily define extensional equivalence of a Synchronous Circuit and a Mealy Machine:

**Definition 4.14: Extensional Equivalence of Mealy Machines and Synchronous Circuits**
Let M be a Mealy Machine, and S be a SYSD, we define $M \equiv S \iff \forall x \in \Sigma^*, \nu(M)(x) = \mu(S)(x)$.

Note: there is an interesting duality to this jump from state machine to string function, in that we can easily define "states" for an arbitrary string function, and trivially obtain a Mealy machine equivalent to an MLP string-function:

- To get the states of a function F on $\Sigma^*$, take the equivalence classes for ~ in $\Sigma^*$, where:
  $x \sim y \iff \forall z \in \Sigma^* \; F(xz)=F(yz)$.
  (A "state" is simply a summary of the past good enough to account for the future.)
- To get a Mealy machine for an MLP F, take those states, and define:
  $\gamma(x^\sim, u) = (x.u)^\sim$ and $\delta(x^\sim, u) = \mathrm{last}(F(x.u))$, where $x^\sim$ is the equivalence class of x under ~ .

Actually, we get the *minimal* state machine extensionally equivalent to F; unfortunately however, this is far from constructive!

# References

[Arnold 81] Arnold, A.
Semantique des Processus Communicants .
*RAIRO Informatique Theorique, v. 15, no. 2, pp. 103-139* , 1981.

[Backus 78] Backus, John.
Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs.
*Communications of the ACM, v. 21, no. 8, pp. 613-641* , 1978.

[Booth 67] Booth, Taylor L.
*Sequential Machines and Automata Theory* .
John Wiley & Sons, New York, 1967.

[Brock-Ackerman 81]
Brock, J. Dean; Ackerman, William B.
Scenarios: A Model of Non-determinate Computation .
In *Formalization of Programming Concepts (LNCS 107), Springer-Verlag*. 1981.

[Brookes 84] Brookes, Stephen D.
*Reasoning about Synchronous Systems* .
Technical Report, CMU-CS-84-145,Computer Science Dept., Carnegie-Mellon Univ., Pittsburgh PA 15213, 1984.

[Burge 75] Burge, W.H.
Stream Processing Functions .
*IBM Journal of Research and Development, v. 19, pp. 12-25* , 1975.

[Burstall-Darlington 77]
Burstall, R. M.; Darlington, John.
A Transformation System for Developing Recursive Programs .
*Journal of the ACM v. 24, no. 1, pp. 44-67* , 1977.

[de Bakker 80] de Bakker, J. W.
*Mathematical Theory of Program Correctness* .
Englewood Cliffs, N.J., 1980.

[Dill 88] Dill, David L.
*Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.*
PhD thesis, CMU-CS-88-119,Computer Science Dept., Carnegie-Mellon Univ., Pittsburgh PA 15213, 1988.

[Faustini 82a] Faustini, Antony Azio.
*The Equivalence of an Operational and a Denotational Semantics for Pure Dataflow* .
PhD thesis, Computer Science Dept, University of Warwick, Coventry, UK, 1982.

[Faustini 82b] Faustini, A.A.
An Operational Semantics for Pure Dataflow .
In *9th Int'l Conf. on Automata, Languages and Programming (LNCS 140), pp. 212-224*. 1982.

[Friedman-Wise 76]
Friedman, D.P.; Wise, D.S.
CONS Should Not Evaluate its Arguments .
In *3rd International Colloq. on Automata, Languages and Programming, pp. 257-284*. 1976.

[Glasgow-MacEwen 87]
Glasgow, Janice I.; MacEwen, Glenn H.
A Computational Model for Distributed Systems Using Operator Nets .
In *PARLE Parallel Architectures and Languages Europe, v. 2 (LNCS 259) pp. 243-260*. 1987.

[Gordon 85] Gordon, M.J.C.
*Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware* .
Technical Report, TR 77,Univ. of Cambridge Computer Lab,Corn Exchange St,Cambridge CB2 3QG,England, 1985.

[Hopcroft-Ullman 79]
Hopcroft, John E.; Ullman, Jeffrey D.
*Introduction to Automata Theory, Languages, and Computation* .
Addison-Wesley, Reading, Massachusetts, 1979.

[Hunt 85] Hunt, Warren A. Jr.
*FM8501: A Verified Microprocessor* .
PhD thesis, TR 47, Institute for Computing Science, Univ. of Texas at Austin, TX 78712, 1985.

[Johnson 83] Johnson, Steven D.
*Synthesis of Digital Designs from Recursion Equations* .
PhD thesis, Indiana University, The MIT Press, Cambridge, Massachusetts, 1983.

[Johnson 84] Johnson, Steven D.
Applicative Programming and Digital Design .
In *11th Symp. on Principles of Programming Languages, Salt Lake City, pp. 218-227.* 1984.

[Kahn 74] Kahn, Gilles.
The Semantics of a Simple Language for Parallel Programming .
In *IFIP Congress 74, Amsterdam, pp. 993-998.* 1974.

[Kleene 67] Kleene, Stephen Cole.
*Introduction to Metamathematics* .
North-Holland, 1967.

[Kloos 87] Kloos, Carlos Delgado.
*Semantics of Digital Circuits* .
PhD thesis, Lecture Notes in Computer Science 285, Springer-Verlag, 1987.

[Landin 65] Landin, P.J.
A Correspondence Between ALGOL 60 and Church's Lambda Notation: Part I and II .
*Communications of the ACM, v. 8, no. 2, pp. 89-101, no. 3, pp. 158-165* , 1965.

[Leiserson-Saxe 83]
Leiserson, Charles E.; Saxe, James B.
Optimizing Synchronous Systems .
*Journal of VLSI and Computer Systems, v. 1, no. 1, pp. 41-67* , 1983.

[Manna 74] Manna, Zohar.
*Mathematical Theory of Computation* .
McGraw-Hill, New York, 1974.

[Manna-Waldinger 85]
Manna, Zohar; Waldinger, Richard.
*The Logical Basis for Computer Programming, Vol. 1* .
Addison-Wesley, Reading, Massachusetts, 1985.

[Mano 76] Mano, M. Morris.
*Computer System Architecture* .
Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1976.

[Mason 86] Mason, Ian A.
*The Semantics of Destructive LISP* .
PhD thesis, CSLI Lecture Notes no. 5, Ventura Hall, Stanford University, CA 94305, 1986.

[Mead-Conway 80]
Mead, Carver A.; Conway, Lynn A.
*Introduction to VLSI Systems* .
Addison-Wesley, Reading, Massachusetts, 1980.

[Melton-Schmidt 86]
Melton, Austin C.; Schmidt, David A.
A Topological Framework for CPOs Lacking Bottom Elements .
In *Mathematical Foundations of Programming Semantics (LNCS 239), pp. 196-204.* 1986.

[Moschovakis 71] Moschovakis, Yiannis N.
Axioms for Computation Theories - First Draft .
In *Proc. Logic Colloquium 1969, pp. 199-255.* 1971.

[Moschovakis 77] Moschovakis, Yiannis N.
On the Basic Notions in the Theory of Induction .
In *Proc. Logic, Foundations of Mathematics, and Computability Theory, pp. 207-236.* 1977.

[Moschovakis 83] Moschovakis, Yiannis N.
Abstract Recursion as a Foundation for the Theory of Algorithms .
In *Proc. Logic Colloquium 1983, Lecture Notes in Math. 1104, pp.289-364.* 1983.

[Russell-Kinniment-Chester-McLauchlan 85]
Russell, G.; Kinniment, D.J.; Chester, E.G.; McLauchlan, M.R.
*C.A.D. for V.L.S.I.* .
Van Nostrand Reinhold (UK) Co. Ltd, 1985.

[Scherlis-Scott 83]
Scherlis, William L.; Scott, Dana S.
First Steps Towards Inferential Programming .
In *IFIP Congress 83, Paris, v. 9, pp. 199-212.* 1983.

[Schmidt 86] Schmidt, David A.
*Denotational Semantics* .
Allyn and Bacon, Boston, 1986.

[Sheeran 83] Sheeran, Mary.
*μFP - an Algebraic VLSI Design Language* .
PhD thesis, PRG-39, Oxford Univ. Computing Lab, 8-11 Keble Rd. Oxford OX1 3QD, England, 1983.

[Talcott 85] Talcott, Carolyn L.
*The Essence of RUM: A theory of the intensional and extensional aspects of Lisp-type computation.*
PhD thesis, STAN-CS-85-1060, Computer Science Dept., Stanford University, CA 94305, 1985.

[Talcott 86] Talcott, Carolyn.
Notes on Transformations .
1986.
Unpublished manuscript.

[van de Snepscheut 85]
van de Snepscheut, Jan L.A.
*Trace Theory and VLSI Design* .
PhD thesis, Lectures Notes in Computer Science 200, Springer-Verlag, 1985.

# Index