ISSN 0105-8517

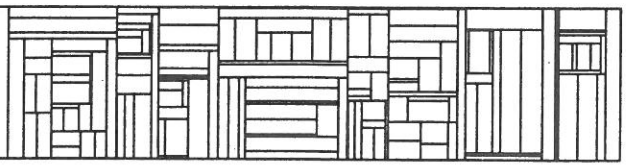
A DENOTATIONAL FRAMEWORK FOR DATA FLOW ANALYSIS

by

Flemming Nielson

DAIMI PB-135
July 1981

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55



A DENOTATIONAL FRAMEWORK FOR DATA FLOW ANALYSIS

Flemming Nielson

Abstract

It is shown how to express data flow analysis in a denotational framework by means of abstract interpretation. A continuation style formulation naturally leads to the MOP (Meet Over all Paths) solution, whereas a direct style formulation leads to the MFP (Maximal Fixed Point) solution.

Contents

0. INTRODUCTION

In this paper data flow analysis is treated from a semantic point of view. Data flow analysis is formulated in a denotational framework and in doing so the method of abstract interpretation is used.

Data flow analysis ([1], [7], [15]) associates properties (data flow information) with program points. The intention is that a property associated with some point must be satisfied in every execution of the program. This is because data flow information usually is used for applications like transforming a program to improve its efficiency. Therefore semantic considerations are needed to ensure the safeness of using using the data flow information, e.g. to assure that the program transformation is meaning preserving. Data flow analysis is usually treated in an operational approach that is not syntax-directed, although there are exceptions (e.g. [13]). In some papers, e.g. [3] and [2], the semantics of data flow analysis is considered in this setting.

Here a denotational approach will be used where semantic functions are "homomorphisms" from syntax to denotations. The motivation for doing so is two-fold. First, the relative merits of an operational versus a denotational approach are not obvious. Previous formulations of data flow analysis in a denotational setting ([5], [4]) have been more ad hoc than the existing operational methods. Also the connection with the usual solutions (MFP and MOP) has not been established. In this paper these two issues are addressed. A second motivation for the denotational approach is to use the data flow information to validate program transformations. The literature on data flow analysis and "program optimization" hardly considers the issue at all. It turns out ([12], [11]) that the approach to data flow analysis developed in this paper forms a suitable platform for doing so.

In section 2 it is described how to systematically develop non-standard denotational semantics specifying data flow information. This is achieved by a series of non-standard semantics leading to a formulation in continuation style. In section 3 it is shown that this formulation yields the

MOP ("meet over all paths") solution. Also the MFP ("maximal fixed point") solution is considered and it is shown how it can be obtained using a direct style formulation. Section 4 contains the conclusions.

1. PRELIMINARIES

This paper builds on concepts from denotational semantics and data flow analysis (including abstract interpretation). The presupposed knowledge of data flow analysis is modest and some of the essential concepts will be reviewed before they are used. In defining semantic equations the notation of [14] and [10] is used, although the domains are not complete lattices but cpo's (as in [9]). Complete lattices are used when data flow analysis is specified. Below some fundamental notions and non-standard notation (≥≥, ==, -t>, -m>, -c>, -a>) are explained.

A partially ordered set $(S, \sqsubseteq)$ is a set S with partial order $\sqsubseteq$, i.e. $\sqsubseteq$ is a reflexive, antisymmetric and transitive relation on S. For $S' \subseteq S$ there may exist a (necessarily unique) least upper bound $\sqcup S'$ in S such that $\forall s \in S: (s \sqsupseteq \sqcup S' \Leftrightarrow \forall s' \in S': s \sqsupseteq s')$. When $S' = \{s_1, s_2\}$ one often writes $s_1 \sqcup s_2$ instead of $\sqcup S'$. Dually a greatest lower bound $\sqcap S'$ may exist. A non-empty subset $S' \subseteq S$ is a chain if S' is countable and $s_1, s_2 \in S' \Rightarrow (s_1 \sqsubseteq s_2 \vee s_2 \sqsubseteq s_1)$. An element $s \in S$ is maximal if $\forall s' \in S: (s' \sqsupseteq s \Rightarrow s' = s)$. A partially ordered set is a cpo if it has a least element ($\bot = \sqcap S$) and any chain has a least upper bound. It is a complete lattice if all $\sqcup S'$ and $\sqcap S'$ exist; then there also is a greatest element ($\top = \sqcup S$). The word domain will be used both for cpo's and complete lattices, and elements of some domain S are denoted s, s', s_1 etc. A domain is flat if any chain contains at most 2 elements, and is of finite height if any chain is finite.

Domains N, Q and T are flat cpo's of natural numbers, quotations and truth values. From cpo's $S_1, \ldots, S_n$ one can construct the separated sum $S_1 + \ldots + S_n$. This is a cpo with a new least element and injection functions $in S_i$, enquiry functions $\in S_i$ and projection functions $\downarrow S_i$. The cartesian product $S_1 \times \ldots \times S_n$ is a cpo with selection functions $\downarrow i$. The cpo S^* of lists is $\{<>\} + S + (S \times S) + \ldots$. Function $\#$ yields the length of a list, function $\dagger i$ removes the first i elements and § concatenates lists. The complete lattice S° is obtained from a set S by adjoining least and greatest elements. By $\wp(S)$ is meant the power-set

of S with set inclusion as partial order. Sometimes a set is regarded as a partially ordered set whose partial order is equality.

All functions are assumed to be total. For partially ordered sets S and S' the set of (total) functions from S to S' is denoted $S \text{ -t> } S'$. The set of monotone (isotone) functions from S to S' is denoted $S \text{ -m> } S'$ and consists of those $f \in S \text{ -t> } S'$ that satisfy $s_1 \sqsubseteq s_2 \Rightarrow f(s_1) \sqsubseteq f(s_2)$. A function $f \in S \text{ -t> } S'$ is continuous if $f(\sqcup S'') = \sqcup\{f(s) \mid s \in S''\}$ holds for any chain $S'' \subseteq S$ whose least upper bound exists. The set of continuous functions from S to S' is denoted by $S \text{ -c> } S'$. A function $f \in S \text{ -t> } S'$ is additive (a complete $\sqcup$-morphism) if $f(\sqcup S'') = \sqcup\{f(s) \mid s \in S''\}$ for any subset $S'' \subseteq S$ whose least upper bound exists. The set of additive functions from S to S' is denoted by $S \text{ -a> } S'$. Any subset of $s \text{ -t> } S'$ is partially ordered by $f_1 \sqsubseteq f_2 \Leftrightarrow \forall s \in S: f_1(s) \sqsubseteq f_2(s)$, and if S' is a cpo or complete lattice the same holds for $S \text{ -t> } S'$, $S \text{ -m> } S'$, $S \text{ -c> } S'$, $S \text{ -a> } S'$.

An element $s \in S$ is a fixed point of $f \in S \text{ -t> } S$ if $f(s) = s$. When S is partially ordered it is the least fixed point provided it is a fixed point and $s' = f(s') \Rightarrow s' \sqsupseteq s$. For S a complete lattice and $f \in S \text{ -m> } S$ the least fixed point always exists and is given by $LFP(f) = \sqcap\{s \mid f(s) \sqsubseteq s\}$. If $f \in S \text{ -c> } S$ then $LFP(f) = FIX(f)$ where $FIX(f) = \sqcup\{f^n(\bot) \mid n \geq 0\}$. If S is only a cpo but $f \in S \text{ -c> } S$ then FIX(f) still is the least fixed point of f.

The symbol == denotes a continuous equality predicate ($S \times S \text{ -c> } T$), whereas = is reserved for true equality. So true == $\bot$ is $\bot$ whereas true = $\bot$ is false. When S is of finite height it is assumed that $s_1 == s_2$ is $\bot$ if one of s_1, s_2 is non-maximal and equals $s_1 = s_2$ otherwise. Similarly $\geq\geq$ is the continuous extension of $\geq$ (the predicate 'greater than or equal to' on the integers). The conditional $t \rightarrow s_1, s_2$ is s_1, s_2 or $\bot$ depending on whether t is true, false or $\bot$. By $f[y/x]$ is meant $\lambda z.\, z == x \rightarrow y, f(z)$. Braces { and } are used to construct sets and to enclose continuations but the context should make clear which is intended.

2. THE FRAMEWORK

In this section it is shown how data flow analysis can be expressed in a denotational framework. The development is performed for a toy language and consists of defining a series of non-standard semantics:

Figure 1.

All the semantics are in continuation style. Semantics ind is the desired formulation of data flow analysis. Semantics sto is a store semantics [10] that is taken to be the canonical definition of the language considered. It will later be motivated why store semantics and continuation style are used.

The toy language consists of commands (syntactic category Cmd), expressions (Exp), identifiers (Ide), operators (Ope) and basic values (Bas). The syntax of commands and expressions can be deduced from table 1, that will be explained shortly. For the remaining syntactic categories it is left unspecified. The syntactic categories are viewed as being sets or partially ordered sets whose partial order is equality.

Store semantics

The store semantics (sto) is given by tables 1, 2 and 3 together. The semantic functions $\mathcal{C}$ (for commands) and $\mathcal{E}$ (for expressions) are defined in table 1. The domains and auxiliary functions needed in table 1 are defined in tables 2 and 3. (For the present ignore domains Occ and Pla and function attach.) The definition of apply (and push) makes use of the undefined semantic function $\mathcal{O}$ for operators (and $\mathcal{B}$ for basic values).

TABLE 1: Semantic functions

$\mathcal{C} \in Cmd \text{ -c> } C \text{ -c> } C$

$\mathcal{C}[\![cmd_1; cmd_2]\!]\ c =$
$\quad \mathcal{C}[\![cmd_1]\!]\{\mathcal{C}[\![cmd_2]\!]\{c\}\}$

$\mathcal{C}[\![ide := exp]\!]\ c =$
$\quad \mathcal{E}[\![exp]\!]\{assign[\![ide]\!]\{c\}\}$

$\mathcal{C}[\![IF\ exp\ THEN\ cmd_1\ ELSE\ cmd_2\ FI]\!]\ c =$
$\quad \mathcal{E}[\![exp]\!]\{cond(\mathcal{C}[\![cmd_1]\!]\{c\}, \mathcal{C}[\![cmd_2]\!]\{c\})\}$

$\mathcal{C}[\![WHILE\ exp\ DO\ cmd\ OD]\!]\ c =$
$\quad FIX(\lambda c'.\mathcal{E}[\![exp]\!]\{cond(\mathcal{C}[\![cmd]\!]\{c'\}, c)\})$

$\mathcal{C}[\![WRITE\ exp]\!]\ c =$
$\quad \mathcal{E}[\![exp]\!]\{write\{c\}\}$

$\mathcal{C}[\![READ\ ide]\!]\ c =$
$\quad read\ \{assign[\![ide]\!]\{c\}\}$

$\mathcal{E} \in Exp \text{ -c> } C \text{ -c> } C$

$\mathcal{E}[\![exp_1\ ope\ exp_2]\!]\ c =$
$\quad \mathcal{E}[\![exp_1]\!]\{\mathcal{E}[\![exp_2]\!]\{apply[\![ope]\!]\{c\}\}\}$

$\mathcal{E}[\![ide]\!]\ c =$
$\quad content[\![ide]\!]\{c\}$

$\mathcal{E}[\![bas]\!]\ c =$
$\quad push[\![bas]\!]\{c\}$

Domain Sta is the domain of states. The connection between identifiers and their values is established without using locations but this is not important for the development to go through. The component Tem is used to hold temporary results arising during evaluation of expressions. This is done by the ordinary method of evaluating expressions on a stack.

TABLE 2: Some domains and auxiliary functions

Domains

Val = T + N + ... + {"nil"}	values
Env = Ide -c> Val	environments
Inp = Val*	inputs
Out = Val*	outputs
Tem = Val*	temporary result stacks
Sta = Env × Inp × Out × Tem	states
Occ = N*	occurrences
Pla = Occ × Q	places

Auxiliary functions

apply⟦ope⟧ ∈ C -c> C

apply⟦ope⟧ = do(Vapply⟦ope⟧, Bapply⟦ope⟧)

Vapply⟦ope⟧ ∈ Sta -c> T ("verify")

Vapply⟦ope⟧ = λ < env, inp, out, tem>.# tem ≥≥ 2

Bapply⟦ope⟧ ∈ Sta -c> Sta ("body")

Bapply⟦ope⟧ = λ <env, inp, out, tem>.

<env, inp, out,<𝒪⟦ope⟧ < tem↓2, tem↓1>>§(tem | 2)>

assign⟦ide⟧, content⟦ide⟧, push⟦bas⟧, read, write defined similarly

Auxiliary functions used in the conditional

Vcond ∈ Sta -c> T ("verify")

Vcond = λ<env, inp, out, tem>. #tem ≥≥ 1 → tem↓1 ∈ T, false

Scond ∈ Sta -c> T ("select")

Scond = λ<env, inp, out, tem>. (tem↓1) | T

Bcond ∈ Sta -c> Sta ("body")

Bcond = λ<env, inp, out, tem>.<env, inp, out, tem †1>

As an example consider apply⟦ope⟧(c)<env, inp, out, tem>. If two temporary results are on top of tem they are replaced by their result and the new state is supplied to c. If tem is not of the expected form an error occurs. The reader acquainted with store semantics [10] should find it straight-forward to supply the omitted definitions.

Table 3 is labelled "interpretation sto" because it is essentially by supplying replacements for table 3 that the remaining semantics (col, sts and ind) are defined. To indicate which semantics is meant, a suffix may be used, e.g. $\mathcal{C}$sto. So $\mathcal{C}$sto⟦cmd⟧(fin-sto) < λide."nil"inVal, inp,<>,<>> specifies the effect of executing program cmd with input inp.

TABLE 3: Interpretation sto

Domains

S = Sta		states
A = Out + {"error"}		answers
C = S -c> A		continuations

Constant

fin ∈ C fin = λ<env, inp, out, tem>. out inA

Auxiliary functions

cond ∈ C × C -c> C
cond(c_1, c_2) = λsta. Vcond(sta) → [Scond(sta) → c_1, c_2](Bcond sta), "error" inA

attach ∈ Pla -c> C -c> C
attach (pla) c = c

do ∈ (Sta -c> T) × (Sta -c> Sta) -c> C -c> C
do(Vg, Bg) c = λsta. Vg(sta) → c(Bg sta), "error" inA

For tables 1, 2 and 3 it is presumably obvious that the functionalities shown are correct. For the remaining semantics the proofs of correctness of functionalities are straight-forward and therefore omitted.

Collecting semantics

For data flow analysis purposes an important concept is that of associating information with a program point. This concept is not expressed in the store semantics and it is therefore convenient to add it to sto, yielding a collecting semantics (col).

A program point can be specified by a tuple <occ, q> ∈ Pla. All the tuples to be considered will be maximal (with respect to the partial ordering). Occurrences (like occ) are used to label nodes of the parse tree. The root is labelled <> and the i'th son of a node labelled occ is labelled occ§<i>. The quotation q is useful for specifying whether the program point is to the left or to the right of the node:

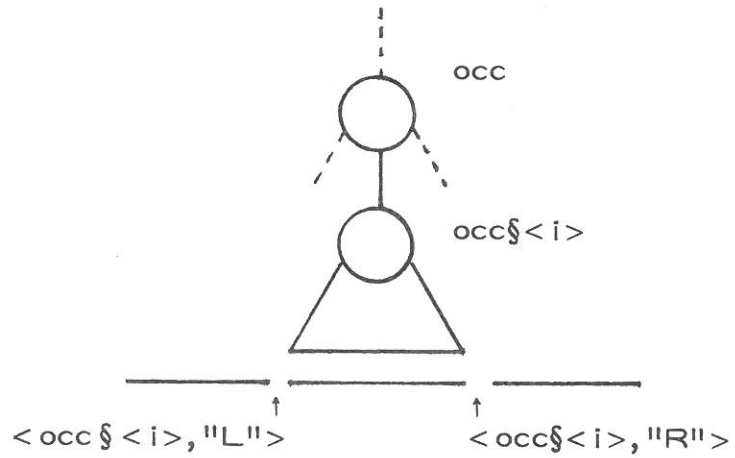


Figure 2.

In the usual view of parse-trees the nodes are not labelled by occurrences. To be able to let the semantic equations associate information with program points (represented by tuples like <occ, q>) it is necessary to supply the appropriate occurrence as an additional argument to the semantic functions $\mathcal{C}$ and $\mathcal{E}$. In this way occurrences are used much like the positions of [6]. Furthermore the semantic equations must be augmented with functions associating information with the program points. The function attach <occ, q> is used for associating information with the program point specified by <occ, q>. The result of performing these changes is sketched in table 4. The proofs of theorems 4 and 5 benefit from the chosen placement of attach.

Because of attach in table 3 the store semantics can be defined using table 4 instead of table 1.

In the collecting semantics (tables 4, 2 and 5) the information to be associated with a program point is the set of states the program can be in whenever control reaches that point. Domain A-col = Pla -c> $\mathcal{P}$(Sta) is used for that and attach-col(pla) is defined accordingly. This choice of A is not the only possibility, e.g. A = (Pla $\times$ Sta)* of [5] can be used instead. While this may give more information it is not needed for a large class of data flow analyses (including those usually handled by means of abstract interpretation). It will later be discussed why the continuations (C-col) are not continuous.

Intuitively sto and col are closely related, because essentially only domain A is different. A connection is formally expressed by:

Theorem 1: For any sta $\in$ Sta, cmd $\in$ Cmd and out $\in$ Out:
$\mathcal{C}$sto$[\![$cmd$]\!]$ <> fin-sto sta = out inA $\Leftrightarrow$
$\exists$ sta' $\in$ Sta: sta' $\downarrow$4 = out $\wedge$ $\mathcal{C}$col$[\![$cmd$]\!]$ <> fin-col sta <<>,"R"> = {sta'}

Proof The proof is by structural induction [14] making use of the predicate P-C $\in$ C-sto $\times$ C-col -t> {true, false} defined by P-C(c-sto, c-col) =
$\forall$ sta $\in$ Sta: $\forall$ out $\in$ Out:
[c-sto(sta) = out inA $\Leftrightarrow$ $\exists$sta' $\in$ Sta: sta' $\downarrow$4 = out $\wedge$ c-col(sta)(<<>,"R">)={sta'}].
By structural induction on cmd (and likewise exp) it is shown that
P-C(c-sto, c-col) $\wedge$ occ $\neq$ <> $\Rightarrow$ P-C($\mathcal{C}$sto$[\![$cmd$]\!]$ occ c-sto, $\mathcal{C}$col$[\![$cmd$]\!]$ occ c-col).
Similarly, by case analysis of cmd it can be shown that
P-C($\mathcal{C}$sto$[\![$cmd$]\!]$ <> fin-sto, $\mathcal{C}$col$[\![$cmd$]\!]$ <> fin-col) using that
P-C(attach-sto(<<>,"R">){fin-sto}, attach-col(<<>,"R">){fin-col}).

The interesting case of the proof is that of the WHILE loop. Abbreviate‡ $g[c_1] = \lambda c_2.\mathcal{E}[\![exp]\!]$occ§<1>{cond($\mathcal{C}[\![cmd]\!]$occ§<2>{$c_2$}, c_1)} and assume P-C(c_1-sto, c_1-col). It is to be shown that P-C(FIX(g-sto$[c_1$-sto$]$), FIX(g-col$[c_1$-col$]$)). It is easy to establish $\forall n \geq 0$: P-C($(g\text{-sto}[c_1\text{-sto}])^n\perp$, $(g\text{-col}[c_1\text{-col}])^n\perp$) but this does not immediately yield the result. The result follows from

‡ For typographical reasons g_{c_1} is written $g[c_1]$.

$\forall sta\, \exists n_0\, \forall n \geq n_0 : (g\text{-}sto[c_1\text{-}sto])^n_\perp\, sta = (\sqcup\{(g\text{-}sto[c_1\text{-}sto])^n_\perp \mid n \geq 0\})sta$
and a similar equation for col. To establish the above equation it suffices to show (dropping suffix sto)
$\forall sta: [(g[c_1])^n_\perp\, sta \neq \perp \Rightarrow \forall c_2: (g[c_1])^n\, c_2\, sta = (g[c_1])^n_\perp\, sta]$
This is because $c_2 = g[c_1]\perp$ implies that $(g[c_1])^n c_2$ is $(g[c_1])^{n+1}\perp$.
Proof is by induction in n and the case n=0 is trivial so consider the induction step. It is straight-forward to see that $(g[c_1])^{n+1}(c_2)(sta)$ independently of c_2 is either $\perp$, "error"inA, $c_1(sta')$ or $(g[c_1])^n(c_2)(sta'')$ where sta' and sta'' are independent of c_2. In the first 3 cases the equation is immediate and in the latter case it follows from the induction hypothesis for n. □

TABLE 4: Modified semantic functions

$\mathcal{C} \in$ Cmd -c> Occ -c> C -c> C

$\mathcal{C}[\![$IF exp THEN cmd$_1$ ELSE cmd$_2$ FI$]\!]$ occ c =
attach <occ, "L"> {
$\mathcal{E}[\![$exp$]\!]$ occ§<1> {
cond($\mathcal{C}[\![$cmd$_1]\!]$ occ§<2> {attach <occ, "R"> {c}}
, $\mathcal{C}[\![$cmd$_2]\!]$ occ§<3> {attach <occ, "R"> {c}})}}

$\mathcal{C}[\![$WHILE exp DO cmd OD$]\!]$ occ c =
attach <occ, "L"> {
FIX($\lambda c'$. $\mathcal{E}[\![$exp$]\!]$ occ§<1> {
cond($\mathcal{C}[\![$cmd$]\!]$ occ§<2> {c'}
, attach <occ, "R"> {c})})}
the remaining clauses are changed similarly to the one for exp$_1$ope exp$_2$

$\mathcal{E} \in$ Exp -c> Occ -c> C -c> C

$\mathcal{E}[\![$exp$_1$ ope exp$_2]\!]$ occ c =
attach <occ, "L"> {
$\mathcal{E}[\![$exp$_1]\!]$ occ§<1> {
$\mathcal{E}[\![$exp$_2]\!]$ occ§<3> {
apply$[\![$ope$]\!]$ {
attach <occ, "R"> {c}}}}}
the remaining clauses are changed similarly to the one for exp$_1$ope exp$_2$

TABLE 5: Interpretation col

Domains

$S = Sta$
$A = Pla \text{ -c> } \mathcal{P}(Sta)$
$C = S \text{ -t> } A$

Constant

$fin \in C \qquad fin = \bot$

Auxiliary functions

$cond \in C \times C \text{ -c> } C$
$cond(c_1, c_2) = \lambda sta.\ Vcond(sta) \rightarrow [Scond(sta) \rightarrow c_1, c_2](Bcond\ sta),\ \bot$

$attach \in Pla \text{ -c> } C \text{ -c> } C$
$attach(pla)\ c = \lambda sta.\ c(sta) \sqcup \bot[\{sta\}/pla]$

$do \in (Sta \text{ -c> } T) \times (Sta \text{ -c> } Sta) \text{ -t> } C \text{ -c> } C$
$do(Vg, Bg)\ c = \lambda sta.\ Vg(sta) \rightarrow c(Bg\ sta),\ \bot$

Example

Consider analysing the program x := 1 with some input inp $\in$ Inp. Abbreviate env = $\lambda ide.$ "nil" inVal and a-col = $\mathcal{C}col[\![x:=1]\!]$<> fin-col<env, inp, <>, <>>. The result of analysing x:=1 is a-col, and

a-col = attach <<>, "L"> {
attach << 1>, "L"> {push[[1]] {attach << 1>, "R" > {
assign[[x]] {attach <<>, "R"> {fin}}}}}} <env, inp, <>, <>>

This means that

a-col <<>, "L" > = a-col<< 1>, "L"> = {<env, inp, <>, <>>}
a-col<< 1>, "R"> = {<env, inp, <>, < 1 inVal>>}
a-col<<>, "R"> = {<env[1 inVal/x], inp, <>, <>>}
and a-col(pla) = $\emptyset$ otherwise. □

Static semantics

There are two ways in which the collecting semantics is not the desired formulation of data flow analysis. One is that the program is only executed for one particular input rather than a set of inputs (e.g. Inp). This is remedied by the static semantics (sts) to be considered below. Another is that sets of states are considered rather than approximate descriptions of sets of states. The latter will be remedied by the induced semantics.

In the static semantics (tables 4, 2 and 6) domain S and functions do and cond have been changed. The intention with do(Vg,Bg)(c) is to supply to c a set of transformed states. Excluded from consideration are states that would not be supplied to the continuation in the collecting semantics. The conditional cond is modified so as to "traverse" both the true and false branch with an appropriate set of states. The partial answers from the two branches are then combined. The connection between col and sts is expressed by the theorem below. A more or less similar result appears in [2] but there for an operational semantics.

Theorem 2: For all cmd $\in$ Cmd and s $\in \mathcal{P}$(Sta):

$$\mathbb{C}sts[\![cmd]\!] <> \text{fin-sts } s = \sqcup\{\mathbb{C}col[\![cmd]\!] <> \text{fin-col sta} \mid sta \in s\}$$

Proof The proof is by structural induction making use of the predicate P-C $\in$ C-col $\times$ C-sts $\to$ {true, false} defined by
P-C(c-col, c-sts) = $\forall s \in \mathcal{P}$(Sta): c-sts(s) = $\sqcup\{$c-col(sta) | sta $\in$ s$\}$.
By structural induction on cmd (and likewise exp) it is shown that
P-C(c-col, c-sts) $\Rightarrow$ P-C($\mathbb{C}col[\![cmd]\!]$ occ c-col, $\mathbb{C}sts[\![cmd]\!]$ occ c-sts).
The proof of the WHILE case makes use of the fact that for a complete lattice $\sqcup\{\sqcup\{x_{i,j} \mid i \in I\} \mid j \in J\} = \sqcup\{\sqcup\{x_{i,j} \mid j \in J\} \mid i \in I\}$ which follows from the fact [14] that $\sqcup\{\sqcup\{x_{i,j} \mid i \in I\} \mid j \in J\} = \sqcup(\cup\{\{x_{i,j} \mid i \in I\} \mid j \in J\})$.

A more detailed proof of this theorem, as well as theorems 3 and 4 later, can be found in [11] in a slightly different notation. □

It appears to be mandatory to work from a store semantics in order for this theorem to hold. (In an approach based on a standard semantics presumably only $\supseteq$ holds [11].) The techniques of [10] can be used to transform a standard semantics into a store semantics.

TABLE 6: Interpretation sts

Domains

$S = \mathcal{P}(Sta)$
$A = Pla \text{ -c> } \mathcal{P}(Sta)$
$C = S \text{ -a> } A$

Constant

$fin \in C \qquad fin = \bot$

Auxiliary functions

$cond \in C \times C \text{ -c> } C$
$cond(c_1, c_2) = \lambda s.\ c_1\{Bcond(sta) \mid Vcond(sta)=true \wedge Scond(sta)=true \wedge sta \in s\}$
$\sqcup c_2\{Bcond(sta) \mid Vcond(sta)=true \wedge Scond(sta)=false \wedge sta \in s\}$

$attach \in Pla \text{ -c> } C \text{ -c> } C$
$attach(pla)\ c = \lambda s.\ c(s) \sqcup \bot[s/pla]$

$do \in (Sta \text{ -c> } T) \times (Sta \text{ -c> } Sta) \text{ -t> } C \text{ -c> } C$
$do(Vg, Bg)\ c = \lambda s.\ c\{Bg(sta) \mid Vg(sta) = true \wedge sta \in s\}$

The method of abstract interpretation

Data flow analysis is almost always expressed in terms of approximate descriptions of sets (of states or values) rather than the sets themselves. It is necessary to work with approximate information if data flow analysis is to be carried out automatically, because otherwise the data flow information might not be computable. The approximate data flow information must be "safe", i.e. describe a set that is not smaller than the precise set.

Example (preparing for constant propagation)

Define the complete lattice $\overline{Val} = \{val \in Val \mid val \text{ is maximal}\}^{\circ}$. An element of $\overline{Val}$ is to describe a set of values, i.e. an element of $\mathcal{P}(Val)$. The method of abstract interpretation ([3], [2]) makes use of a concretization function $\gamma_{\vee} \in \overline{Val} \text{ -m> } \mathcal{P}(Val)$ to express the subset of Val that some element of $\overline{Val}$ describes. A natural choice is $\gamma_{\vee}(\bot) = \emptyset$, $\gamma_{\vee}(\top) = Val$ and $\gamma_{\vee}(val) = \{val\}$ otherwise.

The abstraction function $\alpha_{\vee} \in \mathcal{P}(Val) \text{ -m> } \overline{Val}$ can be used to approximate sets of values. It is natural to define $\alpha_{\vee}(v) = \sqcap\{\overline{val} \mid \gamma_{\vee}(\overline{val}) \sqsupseteq v\}$ because then $\alpha_{\vee}(v)$ is the "best" approximation of the set v of values. As an example $\alpha_{\vee}(\{true\}) = true$ and $\alpha_{\vee}(\{true, false\}) = \top$. □

It is useful when the concretization function (γ) and the abstraction function (α) are related as described by one of the two concepts defined below.

Definition $<\alpha,\gamma>$ is a pair of semi-adjoined functions between partially ordered sets L and M iff

(i) $\alpha \in L \text{ -m> } M$ and $\gamma \in M \text{ -m> } L$

(ii) $\gamma \circ \alpha \sqsupseteq \lambda l.l$

Furthermore, $<\alpha,\gamma>$ is a pair of adjoined functions [3] between L and M iff in addition to (i) and (ii) also

(iii) $\alpha \circ \gamma \sqsubseteq \lambda m.m$ □

The pair $<\alpha_{\vee},\gamma_{\vee}>$ of the example is a pair of adjoined functions between complete lattices $\mathcal{P}(Val)$ and $\overline{Val}$. Condition (ii) expresses the intention that a set of values is approximated by a not smaller set of values so that one can only make "errors on the conservative side" [1]. Condition (i) relates the partial orderings so that obtaining more information in $\mathcal{P}(Val) = L$ corresponds to obtaining more information in $\overline{Val} = M$ and vice versa. Condition (iii) is usually satisfied but it is not needed for the substance of this development. An analysis of the concepts semi-adjoined

and adjoined is given in [11]. Whenever $\gamma \in M \text{ -m> } L$ satisfies $\forall M' \subseteq M: \gamma(\sqcap M') = \sqcap\{\gamma(m) \mid m \in M'\}$ there is a unique $\alpha \in L \text{ -m> } M$ so that $<\alpha,\gamma>$ is a pair of adjoined functions. If $<\alpha,\gamma>$ is a pair of adjoined functions (between complete lattices L and M) then $\alpha = \lambda l.\ \sqcap\{m \mid \gamma(m) \sqsupseteq l\}$, α is additive and γ satisfies the condition displayed above [3].

There is an important difference between Val on the one hand and $\overline{\text{Val}}$ and $\mathcal{P}$(Val) on the other. The partial order of Val means "less defined than" (in the sense of Scott), so a natural condition to impose is that the domains must be cpo's. The partial orders of $\overline{\text{Val}}$ and $\mathcal{P}$(Val) mean something like "logically implies", e.g. $\overline{\text{val}}_1 \sqsubseteq \overline{\text{val}}_2$ means that if a set of values is approximately described by $\overline{\text{val}}_1$ then $\overline{\text{val}}_2$ is also a safe description of the set. The condition that will be imposed is that the domains are complete lattices. While this can be weakened it is important that a greatest element is present. This is contrary to Val where a greatest element would be artificial. It is also this difference in partial orders that accounts for why the continuations of col are not continuous.

Induced semantics

The static semantics is not the desired formulation of data flow analysis because it does not work with approximate description elements. Let $\overline{\text{Sta}}$ be a complete lattice deemed to be more approximate than $\mathcal{P}$(Sta). Also let there be a concretization function $\gamma \in \overline{\text{Sta}} \text{ -m> } \mathcal{P}(\text{Sta})$ and an abstraction function $\alpha \in \mathcal{P}(\text{Sta}) \text{ -m> } \overline{\text{Sta}}$ such that $<\alpha,\gamma>$ is a pair of semi-adjoined functions. The induced semantics (ind $<\alpha,\gamma>$) is obtained by modifying the static semantics in essentially two ways. One is to use $\overline{\text{Sta}}$ instead of $\mathcal{P}$(Sta). The other is to redefine the auxiliary functions so that their effect upon some $\overline{\text{sta}}$ is obtained by first applying the analogous mapping of the static semantics to $\gamma(\overline{\text{sta}})$ and then applying α to the result.

TABLE 7: Interpretation ind$<\alpha,\gamma>$

Domains

$S = \overline{Sta}$
$A = Pla \text{ -c> } S$
$C = S \text{ -m> } A$

$<\alpha,\gamma>$ is a pair of semi-adjoined functions between complete lattices $\mathcal{P}(Sta)$ and $\overline{Sta}$

Constant

$fin \in C \qquad fin = \bot$

Auxiliary functions

$cond \in C \times C \text{ -c> } C$
$cond(c_1, c_2) = \lambda s.\, c_1(\alpha\{Bcond(sta) \mid Vcond(sta)=true \wedge Scond(sta) = true \wedge sta \in \gamma(s)\}) \sqcup c_2(\alpha\{Bcond(sta) \mid Vcond(sta)=true \wedge Scond(sta) = false \wedge sta \in \gamma(s)\})$

$attach \in Pla \text{ -c> } C \text{ -c> } C$
$attach(pla)\ c = \lambda s.\, c(s) \sqcup \bot[s/pla]$

$do \in (Sta \text{ -c> } T) \times (Sta \text{ -c> } Sta) \text{ -t> } C \text{ -c> } C$
$do(Vg, Bg)\ c = \lambda s.\, c(\alpha\{Bg(sta) \mid Vg(sta) = true \wedge sta \in \gamma(s)\})$

Tables 4, 2 and 7 contain the details. That the induced semantics is "safe" follows from:

Theorem 3: For all $cmd \in Cmd$ and $\overline{sta} \in \overline{Sta}$:

$\mathcal{C}sts[\![cmd]\!] <> fin\text{-}sts\ (\gamma(\overline{sta})) \sqsubseteq \gamma \circ (\mathcal{C}ind[\![cmd]\!] <> fin\text{-}ind\ \overline{sta})$

whenever $<\alpha,\gamma>$ is a pair of semi-adjoined functions (between complete lattices $\mathcal{P}(Sta)$ and $\overline{Sta}$) and ind is ind$<\alpha,\gamma>$.

Proof The proof is by structural induction making use of the predicate $P\text{-}C \in C\text{-}sts \times C\text{-}ind \text{ -t> } \{true, false\}$ defined by $P\text{-}C(c\text{-}sts, c\text{-}ind) = [c\text{-}sts \circ \gamma \sqsubseteq \lambda\overline{sta}.\gamma \circ (c\text{-}ind(\overline{sta}))]$. By structural induction on cmd (and likewise exp) it is shown that $P\text{-}C(c\text{-}sts, c\text{-}ind) \Rightarrow P\text{-}C(\mathcal{C}sts[\![cmd]\!] \text{ occ } c\text{-}sts, \mathcal{C}ind[\![cmd]\!] \text{ occ } c\text{-}ind)$. The proof makes use of $\lambda f.\gamma \circ f$ being monotone, that $\gamma \circ \alpha \sqsupseteq \lambda states.\, states$ and that continuations of sts are monotone.

□

It is also possible to specify data flow analysis by means of approximate semantics other than ind$<\alpha,\gamma>$. Then an analogue of theorem 3 can be used to relate such semantics [11].

Example (constant propagation)

One way to specify the data flow analysis "constant propagation" is by the induced semantics ind$<\alpha_S,\gamma_S>$. Domain $\overline{Sta}$ is (Ide -c> $\overline{Val}$) x $\overline{Val}^{\circledast}$ where $\overline{Val}^{\circledast}$ is $\overline{Val}^{*}$ augmented with a greatest element ($\top$). The concretization function $\gamma_S \in \overline{Sta}$ -m> $\mathcal{P}$(Sta) is given by:

$$\gamma_S<\overline{env},\overline{tem}> = \{<env, inp, out, tem> \in Sta \mid$$
$$\forall\, ide \in Ide:\ env[\![ide]\!] \in \gamma_V(\overline{env}[\![ide]\!]) \wedge \quad (i)$$
$$[\overline{tem} = \top \vee [\overline{tem} \notin \{\bot,\top\} \wedge \#tem = \#\overline{tem} \wedge \quad (ii)$$
$$\forall\, j \in \{1,\ldots,\#tem\}:\ tem\downarrow j \in \gamma_V(\overline{tem}\downarrow j)]]\} \quad (iii)$$

For a state <env, inp, out, tem> to be one of those described by $<\overline{env},\overline{tem}>$ the environment env must be one of those described by $\overline{env}$ (condition (i)) and the temporary result stack tem must be one of those described by $\overline{tem}$ (conditions (ii) and (iii)). Conditions (i) and (iii) are reasonably straightforward. Condition (ii) is somewhat more "technical": The greatest element of $\overline{Val}^{\circledast}$ describes any stack in Val *, the least element of $\overline{Val}^{\circledast}$ describes no stacks in Val* and $<\overline{val}_1,\ldots,\overline{val}_n> \in \overline{Val}^{\circledast}$ describes some stacks of n elements. The abstraction function $\alpha_S \in \mathcal{P}$(Sta) -m> $\overline{Sta}$ is defined by α_S(states) = $\sqcap\{\overline{sta} \mid \gamma_S(\overline{sta}) \supseteq states\}$. Then $<\alpha_S,\gamma_S>$ is a pair of adjoined functions between complete lattices $\mathcal{P}$(Sta) and $\overline{Sta}$.

In data flow analysis it is usually assumed that the description lattices are of finite height (to ensure computability of the data flow information). This is the case when Ide is finite. For an example constant propagation analysis consider the program x := 1. Abbreviate $\overline{env}$ = λide. "nil" in $\overline{Val}$ and a-ind = $\mathcal{C}$ind$[\![x:=1]\!]$<> fin-ind $<\overline{env},<>>$. The result of analysing x := 1 then is a-ind, and

$$\text{a-ind}\langle\langle\rangle, \text{"L"}\rangle = \text{a-ind}\langle\langle 1\rangle, \text{"L"}\rangle = \langle\overline{\text{env}}, \langle\rangle\rangle$$
$$\text{a-ind}\langle\langle 1\rangle, \text{"R"}\rangle = \langle\overline{\text{env}}, \langle 1 \text{ in} \overline{\text{Val}}\rangle\rangle$$
$$\text{a-ind}\langle\langle\rangle, \text{"R"}\rangle = \langle\overline{\text{env}}[1 \text{ in} \overline{\text{Val}}/x], \langle\rangle\rangle$$

and a-ind(pla) = $\perp$ otherwise. Note that information is obtained about the evaluation of expressions (here 1) without the need for converting the program to a sequence of one-operator assignments. This is because attach has been placed in the semantic clauses for expressions.

□

In summary, the development performed has succeeded in formally validating the data flow information (specified by the induced semantics) with respect to the collecting semantics (theorems 2 and 3). This is essentially similar to what is done in [5], although the present development is more systematic: a data flow analysis (ind$\langle\alpha,\gamma\rangle$) is obtained merely by specifying an abstraction function (α) and a concretization function (γ), such that the pair ($\langle\alpha,\gamma\rangle$) of functions is a pair of semi-adjoined functions. However, the semantics of a program is really given by the store semantics. None of the theorems (including theorem 1) can be used to validate data flow information with respect to the store semantics. (A similar defect holds for [5].) It seems impossible to do so directly because two programs may have the same denotation in the store semantics and yet different denotations in the induced semantics. An indirect way of relating the collecting semantics (and by theorems 2 and 3 also the induced semantics) to the store semantics is considered in [12] and [11] where the collecting semantics is used to validate program transformations.

3. THE MOP AND MFP SOLUTIONS

In this section the data flow information specified in section 2 is related to the traditionally considered MOP and MFP solutions. "It appears generally true that for data flow analysis problem, we search for the [MOP] solution" [7], and theorem 4 will show that $\mathcal{C}ind[\![cmd]\!](<>)(fin\text{-}ind)(s)$ yields the MOP solution. By using a direct style formulation instead of the continuation style formulation it is possible to obtain the MFP solution (theorem 5).

To compare the traditional (operational) approach to data flow analysis with that of section 2 it is necessary to superimpose a flow chart view upon programs. The flowcharts to be considered will have arcs to correspond to "places". The flowchart constructed from program cmd has unique entry arc $<<>,"E">$, unique exit arc $<<>,"R">$ and is represented by a set of tuples of the form $<pla_1, pla_2, tf>$. Such a tuple is intended to express that when "going" from pla_1 to pla_2 the data flow information is changed as specified by $tf \in TF = S \to S$. Thus tf is a <u>transfer function</u> [1] associated with the basic block (node) that has pla_1 leading in and pla_2 leading out. The basic blocks are smaller than is usual (in fact many are "empty") because attach functions have been placed as they have; while this can be remedied, the placement used makes the proofs of theorems 4 and 5 less involved and simplifies the presentation below.

To construct a flowchart from a program the semantic functions $F\mathcal{C}$ and $F\mathcal{E}$ are used (tables 8, 2, 7 and 9). As an example consider the flowchart in figure 3 (ignoring the dotted rectangle). It has arcs labelled $pla_0, \ldots, pla_4$ and basic blocks with transfer functions $tf_0, \ldots, tf_3$ associated and is represented by $fc = \{<pla_i, pla_{i+1}, tf_i> \mid 0 \le i \le 3\}$. This representation is obtained from the program x := 1 using the function $F\mathcal{C}$. Since $F\mathcal{C}[\![x := 1]\!] <>$ is

$Fattach(pla_1) * F\mathcal{E}[\![1]\!]<1> * Fdo(Vassign[\![x]\!], Bassign[\![x]\!]) * Fattach(pla_4)$

and $F\mathcal{E}[\![1]\!]<1>$ is $Fattach(pla_2) * Fdo(Vpush[\![1]\!], Bpush[\![1]\!]) * Fattach(pla_3)$

the result of $F\mathcal{C}ind[\![x := 1]\!]<> <pla_0, tf_0>$ is $<<pla_4, tf_4>, fc>$.

TABLE 8: Semantic functions specifying the flowchart

$F\mathcal{C} \in Cmd \to Occ \to FG$

$F\mathcal{C}[\![\text{IF exp THEN cmd}_1 \text{ ELSE cmd}_2 \text{ FI}]\!]\ occ =$
Fattach <occ, "L"> *
$F\mathcal{E}[\![exp]\!]\ occ\S<1>$ *
Fcond($F\mathcal{C}[\![cmd_1]\!]\ occ\S<2>$ * Fattach<occ, "R">
, $F\mathcal{C}[\![cmd_2]\!]\ occ\S<3>$ * Fattach<occ, "R">)

$F\mathcal{C}[\![\text{WHILE exp DO cmd OD}]\!]\ occ =$
Fattach<occ, "L"> *
[$F\mathcal{E}[\![exp]\!]\ occ\S<1>$ * Fcond($F\mathcal{C}[\![cmd]\!]\ occ\S<2>$, Fattach<occ, "R">)
Ffix < <occ§<2>, "R">, <occ§<1>, "L">, λs. s>]

the remaining clauses are similar to that of exp_1 ope exp_2

$F\mathcal{E} \in Exp \to Occ \to FG$

$F\mathcal{E}[\![exp_1 \text{ ope } exp_2]\!]\ occ =$
Fattach<occ, "L"> *
$F\mathcal{E}[\![exp_1]\!]\ occ\S<1>$ *
$F\mathcal{E}[\![exp_2]\!]\ occ\S<3>$ *
Fdo(Vapply[[ope]], Bapply[[ope]]) *
Fattach<occ, "R">

the remaining clauses are similar to that of exp_1 ope exp_2

TABLE 9: Additions to interpretation ind (for FC and $F\mathcal{E}$)

Domains

$TF = S \text{ -m> } S$		transfer functions
$FC = \mathcal{P}(Pla \times Pla \times TF)$		(representation of) flowcharts
$PI = Pla \times TF$		partial information
$FG = PI \text{ -t> } PI \times FC$		flowchart generators

Combinator

$$fg_1 * fg_2 = \lambda pi.\langle fg_2(fg_1(pi)\downarrow 1)\downarrow 1,\ fg_2(fg_1(pi)\downarrow 1)\downarrow 2 \sqcup fg_1(pi)\downarrow 2\rangle$$

Auxiliary functions

$$Fcond \in FG \times FG \text{ -t> } FG$$

$$Fcond(fg_1, fg_2) = \lambda\langle pla, tf\rangle.\langle fg_2\langle pla, tff \circ tf\rangle \downarrow 1, fg_1\langle pla, tft \circ tf\rangle \downarrow 2 \sqcup fg_2\langle pla, tff \circ tf\rangle \downarrow 2\rangle$$

$$tft = \lambda s.\alpha\{Bcond(sta) \mid Vcond(sta)=true \wedge Scond(sta)=true \wedge sta \in \gamma(s)\}$$
$$tff = \lambda s.\alpha\{Bcond(sta) \mid Vcond(sta)=true \wedge Scond(sta)=false \wedge sta \in \gamma(s)\}$$

$$Fattach \in Pla \text{ -t> } FG$$

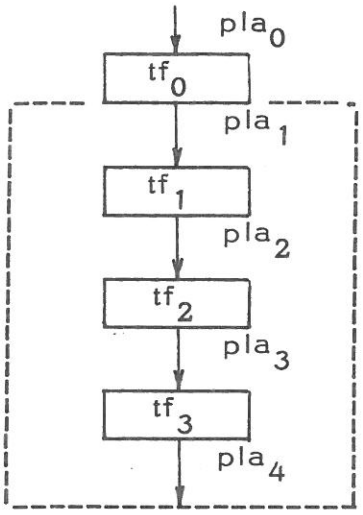
$$Fattach(pla) = \lambda\langle pla', tf\rangle.\langle \langle pla, \lambda s.s\rangle, \{\langle pla', pla, tf\rangle\}\rangle$$

$$Fdo \in (Sta \text{ -c> } T) \times (Sta \text{ -c> } Sta) \text{ -t> } FG$$

$$Fdo(Vg, Bg) = \lambda\langle pla, tf\rangle.\langle \langle pla, \lambda s.\alpha\{Bg(sta) \mid Vg(sta) = true \wedge sta \in \gamma(tf(s))\}\rangle, \emptyset\rangle$$

$$Ffix \in FG \times (Pla \times Pla \times TF) \text{ -t> } FG \quad \text{(infixed)}$$

$$fg\ Ffix\langle pla_1, pla_2, tf\rangle = \lambda pi.\langle fg(pi)\downarrow 1,\ fg(pi)\downarrow 2 \cup \{\langle pla_1, pla_2, tf\rangle\}\rangle$$



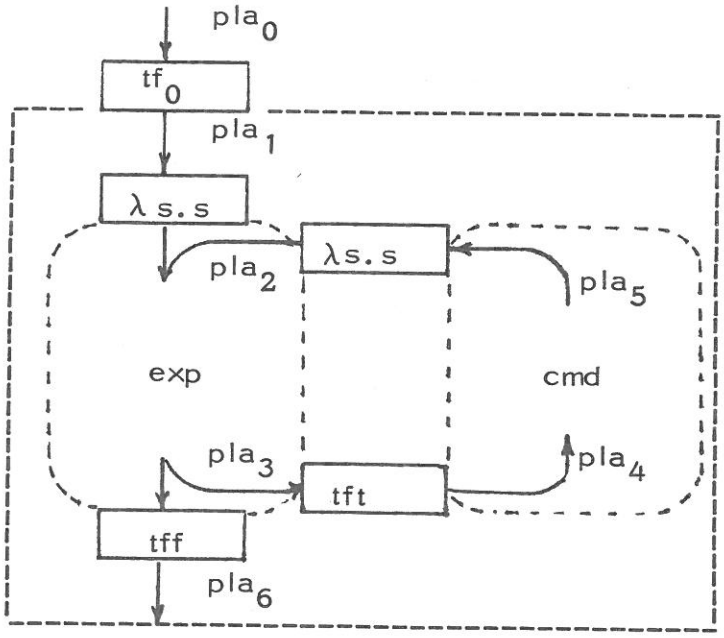
$pla_0 = \langle\langle\rangle, \text{"E"}\rangle$
$pla_1 = \langle\langle\rangle, \text{"L"}\rangle$
$pla_2 = \langle\langle 1\rangle, \text{"L"}\rangle$
$pla_3 = \langle\langle 1\rangle, \text{"R"}\rangle$
$pla_4 = \langle\langle\rangle, \text{"R"}\rangle$

$tf_0 = tf_1 = tf_4 = \lambda s.\, s$
$tf_2 = \lambda s.\, \alpha\{Bpush[\![1]\!]sta \mid Vpush[\![1]\!]sta = true \wedge sta \in \gamma(s)\}$
$tf_3 = \lambda s.\, \alpha\{Bassign[\![x]\!]sta \mid Vassign[\![x]\!]sta = true \wedge sta \in \gamma(s)\}$

Figure 3.

Here the $\langle\rangle$ is the occurrence that is needed as usual. The $\langle pla_0, tf_0\rangle$ is needed because the representation of the basic block to which tf_0 is associated can only be constructed once pla_1 is known. Therefore FC is defined so as to take $\langle pla_0, tf_0\rangle$ as a parameter and construct the relevant tuple $(\langle pla_0, pla_1, tf_0\rangle)$, although it is only the part of the flowchart enclosed in the dotted rectangle that intuitively "corresponds" to x := 1. In order for sequencing (by means of $*$) to work it is necessary to let FC produce not only a flowchart but also a tuple like the $\langle pla_4, tf_4\rangle$ above; indeed $\langle pla_0, tf_0\rangle$ could in principle have been produced by some $FC[\![\ldots]\!]$ if x := 1 had occurred in some context. One way to read $\langle pla_0, tf_0\rangle$ and similarly $\langle pla_4, tf_4\rangle$, is that since arc pla_0 was traversed the (as yet unrecorded) transfer function tf_0 has been encountered.

In general the flowchart associated with program cmd is specified by $FC ind[\![cmd]\!]\langle\rangle\langle pla_0, tf_0\rangle \downarrow 2$. As a further explanation of FC consider the clause for WHILE exp DO cmd OD that is illustrated in figure 4.



$pla_1 = \langle occ, "L" \rangle$

$pla_2 = \langle occ\S\langle 1\rangle, "L" \rangle$

$pla_3 = \langle occ\S\langle 1\rangle, "R" \rangle$

$pla_4 = \langle occ\S\langle 2\rangle, "L" \rangle$

$pla_5 = \langle occ\S\langle 2\rangle, "R" \rangle$

$pla_6 = \langle occ, "R" \rangle$

tff, tft as in table 9

Figure 4.

The dotted rectangles correspond to syntactic subphrases. The flowcharts for exp and cmd have not been shown in detail. The entire flowchart $FC ind[\![\text{WHILE exp DO cmd OD}]\!]\ occ \langle pla_0, tf_0 \rangle \downarrow 2$ is

$$\{\langle pla_0, pla_1, tf_0 \rangle\} \cup$$
$$FE ind[\![\text{exp}]\!]\ occ\S\langle 1\rangle \langle pla_1, \lambda s.s \rangle \downarrow 2 \cup$$
$$\{\langle pla_3, pla_6, tff \rangle\} \cup$$
$$FC ind[\![\text{cmd}]\!]\ occ\S\langle 2\rangle \langle pla_3, tft \rangle \downarrow 2 \cup$$
$$\{\langle pla_5, pla_2, \lambda s.s \rangle\}$$

It is the tuple $\langle pla_5, pla_2, \lambda s.s \rangle$ that accounts for the "iterative nature" of the WHILE construct.

Some properties of FC and FE are mentioned below. They are needed in the proofs of theorems 4 and 5 and may help in giving a better understanding of FC and FE. The proofs of these properties are omitted since they are straight-forward (e.g. by structural induction). The formulation uses the phrase "pla is a descendant of occ", which means that $pla\downarrow 1 = occ\S occ'$ for some maximal occ', and that $pla\downarrow 2 \in \{"L", "R"\}$. Consider $FC ind[\![\text{cmd}]\!]\ occ \langle pla_0, tf_0 \rangle = \langle pi, fc \rangle$ or similarly $FE ind[\![\text{exp}]\!]\ occ \langle pla_0, tf_0 \rangle = \langle pi, fc \rangle$. Let $\langle pla, pla', tf \rangle$ be an arbitrary element of fc, and assume (as will always be the case) that occ and pla_0 are maximal and that pla_0 is not

a descendant of occ. Then,

- pi=<<occ,"R">, λs.s>
- pla' is maximal and is a descendant of occ
- pla is maximal and is either pla_0 or a descendant of occ, but it is not <occ,"R">. Intuitively this means that the flowchart fc is immediately left after traversing <occ,"R">.
- $<pla_0, <occ,"L">, tf_0>$ is the only tuple of fc with pla_0 as the first component. This means that the first arc of fc to be traversed is <occ,"L"> and also that $fc\setminus\{<pla_0,<occ,"L">,tf_0>\}$ is independent of $<pla_0, tf_0>$ (i.e. independent of "how the flowchart is entered").

The MOP solution

For a flowchart fc with entry arc pla and description element s holding there let MOPS(fc,pla,s) ∈ Pla –t> S denote the MOP ("meet over all paths") solution. It is defined by $MOPS(fc,pla,s) = \lambda pla'.\ \sqcup\{ tf_n(\ldots(tf_1(s))) \mid <pla_0,pla_1,tf_1>,\ldots,<pla_{n-1},pla_n,tf_n> \in fc \wedge pla_0 = pla \wedge pla_n = pla'\}$. This formulation essentially is that of [7]. One difference is that [7] uses ⊓ instead of ⊔, i.e. uses the dual lattice, but this is not crucial [13]. Another is that MOPS(fc,pla,s)(pla) need not be s but is likely to be ⊥. The similarity between the approach of section 2 and the traditional approach is expressed by:

Theorem 4: For all cmd ∈ Cmd, s ∈ S and pla ∈ Pla:
$\mathfrak{C}ind[\![cmd]\!] <> fin\text{-}ind\ s\ pla =$
$MOPS(\ F\mathfrak{C}ind[\![cmd]\!]\ (<>)(<<>,"E">,\lambda s.s>)\downarrow 2,\ <<>,"E">,s)(pla)$
where ind = $ind_{<\alpha,\gamma>}$ for $<\alpha,\gamma>$ a pair of semi-adjoined functions between complete lattices $\mathcal{P}(Sta)$ and S.

Proof: See appendix 1. □

Note that both sides of the equality sign yield ⊥ when pla is not maximal or not a descendant of occ.

The MFP solution

Denote by MFPS(fc, pla, s) $\in$ Pla -t> S the MFP ("maximal fixed point") solution for flowchart fc with entry arc pla and description element s holding there. It is defined by MFPS(fc, pla, s) = LFP(step(fc, pla, s)) where step(fc, pla, s) $\in$ (Pla -t> S) -m> (Pla -t> S) is the "data flow equations", i.e. specifies the effect of advancing data flow information along one basic block. It is defined by step(fc, pla, s) = $\lambda a.\lambda pla'. \bigsqcup \{ tf(a[s/pla]pla'') \mid \langle pla'', pla', tf \rangle \in fc \}$. Here pla, pla' and pla'' are going to be maximal so that a[s/pla]pla is s and a[s/pla]pla'' is a(pla'') otherwise. Again, this formulation is essentially the one given in [7] (where greatest fixed points and $\bigsqcap$ are used instead of least fixed points and $\bigsqcup$).

To obtain a denotational semantics specifying the MFP solution, new semantic functions DCind and DEind are defined (tables 10, 2, 7, and 11). They are in "direct style" and the functionality for DCind[[cmd]] (occ) is S -m> S x A. From description element s $\in$ S is obtained the tuple $\langle s', a \rangle$ where the component a specifies partial data flow information for cmd. When occ is maximal it is easy to show that s' = a <occ, "R"> (see appendix 2, lemma 3).

In the clause for DCind[[IF exp THEN cmd_1 ELSE cmd_2 FI]] use is made of the function Dcond. The approximate description element supplied to "the rest of the program" is $s_1 \sqcup s_2$, where s_1 is produced along one branch and s_2 along the other. This differs from Cind where "the rest of the program" is analysed with s_1 and s_2 separately and only the resulting data flow information is combined.

Note in the clause for DCind[[WHILE exp DO cmd OD]] that $\lambda s'.Dg(s' \sqcup s)$ rather than e.g. $\lambda s'.Dg(s')$ is used. Consider some iteration of the WHILE construct where s holds at pla_2 (see figure 4). Then some s' will be computed to hold at pla_5. It is then $s' \sqcup s$ (not s') that holds at pla_2 and must be used in the next iteration if the MFP solution is to be obtained.

TABLE 10: Semantic functions in direct style

$D\mathcal{C} \in Cmd \text{ -c> } Occ \text{ -c> } S \text{ -m> } S \times A$

$D\mathcal{C}[\![\text{IF exp THEN cmd}_1 \text{ ELSE cmd}_2 \text{ FI}]\!]\ occ =$
$\quad Dattach\langle occ, \text{"L"}\rangle \ *$
$\quad D\mathcal{E}[\![exp]\!]\ occ\S\langle 1\rangle \ *$
$\quad Dcond(\ D\mathcal{C}[\![cmd_1]\!]\ occ\S\langle 2\rangle \ * \ Dattach\langle occ, \text{"R"}\rangle$
$\qquad , D\mathcal{C}[\![cmd_2]\!]\ occ\S\langle 3\rangle \ * \ Dattach\langle occ, \text{"R"}\rangle\ )$

$D\mathcal{C}[\![\text{WHILE exp DO cmd OD}]\!]\ occ =$
$\quad Dattach\langle occ, \text{"L"}\rangle \ *$
$\quad FIX(\lambda Dg.\lambda s.\ [D\mathcal{E}[\![exp]\!]\ occ\S\langle 1\rangle \ *$
$\qquad\qquad Dcond(\ D\mathcal{C}[\![cmd]\!]\ occ\S\langle 2\rangle \ * \ (\lambda s'.Dg(s' \sqcup s))$
$\qquad\qquad\quad , Dattach\langle occ, \text{"R"}\rangle)]\ s\ )$

the remaining clauses are similar to that of exp_1 ope exp_2

$D\mathcal{E} \in Exp \text{ -c> } Occ \text{ -c> } S \text{ -m> } S \times A$

$D\mathcal{E}[\![exp_1 \text{ ope } exp_2]\!]\ occ =$
$\quad Dattach\langle occ, \text{"L"}\rangle \ *$
$\quad D\mathcal{E}[\![exp_1]\!]\ occ\S\langle 1\rangle \ *$
$\quad D\mathcal{E}[\![exp_2]\!]\ occ\S\langle 3\rangle \ *$
$\quad Ddo(Vapply[\![ope]\!], Bapply[\![ope]\!]) \ *$
$\quad Dattach\langle occ, \text{"R"}\rangle$

the remaining clauses are similar to that of exp_1 ope exp_2

TABLE 11: Additions to interpretation ind (for $D\mathbb{C}$ and $D\mathcal{C}$)

Auxiliary functions

$Dcond \in (S \text{ -m> } S \times A) \times (S \text{ -m> } S \times A) \text{ -c> } (S \text{ -m> } S \times A)$

$Dcond(dg_1, dg_2) = \lambda s.\, dg_1(tft(s)) \sqcup dg_2(tff(s))$

tft, tff as in table 9

$Dattach \in Pla \text{ -c> } (S \text{ -m> } S \times A)$

$Dattach(pla) = \lambda s.\, \langle s, \bot[s/pla] \rangle$

$Ddo \in (Sta \text{ -c> } T) \times (Sta \text{ -c> } Sta) \text{ -t> } (S \text{ -m> } S \times A)$

$Ddo(Vg, Bg) = \lambda s.\, \langle \alpha\{Bg(sta) \mid Vg(sta) = true \wedge sta \in \gamma(s)\}, \bot \rangle$

Combinator $*$ as in table 9.

The connection between the MFP solution and $D\mathbb{C}ind$ is given by:

Theorem 5: For all $cmd \in Cmd$, $s \in S$ and $pla \in Pla$:

$D\mathbb{C}ind[\![cmd]\!]\, \langle\rangle\, s \downarrow 2\, pla =$

$MFPS(F\mathbb{C}ind[\![cmd]\!](\langle\rangle)(\langle \langle\rangle, \text{"E"} \rangle, \lambda s.\, s \rangle) \downarrow 2\ ,\ \langle\langle\rangle, \text{"E"}\rangle\ ,\ s)\, pla$

where $ind = ind\langle\alpha,\gamma\rangle$ for $\langle\alpha,\gamma\rangle$ a pair of adjoined functions between complete lattices $\mathcal{P}(Sta)$ and S, where S is of finite height.

Proof When $\langle\alpha,\gamma\rangle$ is as above then α is additive and hence continuous. Also γ is continuous when S is of finite height. The only properties of $\langle\alpha,\gamma\rangle$ to be used in the proof is that $\langle\alpha,\gamma\rangle$ is a pair of semi-adjoined functions between complete lattices $\mathcal{P}(Sta)$ and S such that α and γ are continuous. For the proof proper see appendix 2. □

Both sides of the equality are $\bot$ when pla is not maximal. In the literature it is usually assumed that S is of finite height because then the MFP solution is computable (provided the transfer functions are). This holds even if S is infinite (as in the constant propagation analysis of section 2). In contrast, the MOP solution need not be computable even when S is of finite height [8]. It follows from a theorem of J. B. Kam [7] that the constant propagation

analysis is an example of this.

That $\mathbb{C}\mathrm{ind}[\![\mathrm{cmd}]\!](<>)(\mathrm{fin\text{-}ind})(s) \sqsubseteq D\mathbb{C}\mathrm{ind}[\![\mathrm{cmd}]\!](<>)(s) \downarrow 2$ follows from theorems 4 and 5 when $<\alpha,\gamma>$ satisfies the conditions of (the proof of) theorem 5. This is because $\mathrm{MOPS}(\mathrm{fc},\mathrm{pla},s) \sqsubseteq \mathrm{MFPS}(\mathrm{fc},\mathrm{pla},s)$ (see e.g. [3]). Since sts is a special case of ind (with $S = \mathcal{P}(\mathrm{Sta})$ and $\alpha = \gamma = \lambda\mathrm{states}.\mathrm{states}$) it is meaningful to consider $D\mathbb{C}\mathrm{sts}$. The transfer functions of sts are additive and it therefore follows from [3] that $\mathbb{C}\mathrm{sts}[\![\mathrm{cmd}]\!](<>)(\mathrm{fin\text{-}sts})(s) = D\mathbb{C}\mathrm{sts}[\![\mathrm{cmd}]\!](<>)(s) \downarrow 2$.

4. CONCLUSION

It has been shown how data flow analysis can be specified in a denotational approach, by systematically transforming a store semantics to an "induced semantics" parameterized by a pair of semi-adjoined functions. It is claimed that the approach is no less systematic than existing operational methods. Semantic characterizations are with respect to the collecting semantics. To give a "semantic characterization" of the collecting semantics in terms of a store semantics it seems to be necessary to consider program transformations (as is done in [12], [11]).

The data flow analyses considered in this paper could be called "history-insensitive". In [11] a similar development yields a semantic characterization of "available expressions" (a "history-sensitive" analysis). Unfortunately the machinery required to establish the semantic characterization is somewhat complex. More research is needed to handle "live variables" (a "future-sensitive" analysis) and languages with arbitrary jumps and procedures.

The relationship between continuation-style and MOP and between direct-style and MFP shows that for data flow analysis purposes continuation-style is inherently more accurate than direct style. However, it should be kept in mind that the MOP solution can always be specified as the MFP solution to a different data flow analysis problem (whose MFP solution need not be computable) [3].

Acknowledgement

I should like to thank Neil Jones for his continuing interest in and helpful comments upon this work, and Patrick Cousot, Gordon Plotkin and Hanne Riis for helpful comments.

References

1. Aho, A.V. and Ullman, J.D.: Principles of Compiler Design. Addison-Wesley, London, 1977.

2. Cousot, P. and Cousot, R.: Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. Proc. 4th ACM Symp. on Principles of Programming Languages (1977) 238-252.

3. Cousot, P. and Cousot, R.: Systematic design of program analysis frameworks. Proc. 6th ACM Symp. on Principles of Programming Languages (1979) 269-282.

4. Donzeau-Gouge, V.: Utilisation de la semantique denotationelle pour l'étude d'interpretations non-standard. Report no. 273, INRIA, France (1978).

5. Donzeau-Gouge, V.: Denotational definition of properties of program computations. In: Program Flow Analysis: Theory and Applications (S.S. Muchnick and N.D. Jones, Eds.), Prentice-Hall, New Jersey, 1981, pp. 343-379.

6. Gordon, M.J.C.: The Denotational Description of Programming Languages: An Introduction. Springer Verlag, Berlin, 1979.

7. Hecht, M.S.: Flow Analysis of Computer Programs. North-Holland, New York, 1977.

8. Kam, J.B. and Ullman, J.D.: Monotone data flow analysis frameworks. Acta Informatica 7 (1977) 305-317.

9. Milner, R.: Program semantics and mechanized proof. In: Foundations of Computer Science II (K.R. Apt and J.W. de Bakker, Eds.), Mathematical Centre Tracts 82, Amsterdam (1976), 3-44.

10. Milne, R. and Strachey, C.: A Theory of Programming Language Semantics. Chapman and Hall, London, 1976.

11. Nielson, F.: Semantic foundations of data flow analysis. M.Sc. Thesis, Report no. PB-131, Aarhus University, Denmark (1981).

12. Nielson, F.: Program transformations in a denotational setting. Report no. PB-140, Aarhus University, Denmark (1981).

13. Rosen, B.K.: Monoids for rapid data flow analysis. SIAM J. Comput. 9 (1980) 159-196.

14. Stoy, J.E.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press, Cambridge, MA, 1977.

15. Ullman, J.D.: A survey of data flow analysis techniques. 2nd USA-Japan Computer Conf. (1975) 335-342.

Appendix 1: Proof of theorem 4

In this appendix suffix ind is omitted. For the proof the function Close $\in$ FC $\times$ Pla $\times$ $\mathcal{P}$(S) -t> Pla -t> $\mathcal{P}$(S) is useful. It is defined by Close(fc, pla', S')(pla) =

$$\{tf_n(\ldots(tf_1(s)))\mid s \in S' \wedge <pla_0, pla_1, tf_1>, \ldots, <pla_{n-1}, pla_n, tf_n> \in fc \wedge pla_0 = pla' \wedge pla_n = pla\}$$

Then MOPS(fc, pla_0, s_0) = λpla. $\sqcup$[Close(fc, pla_0, $\{s_0\}$)(pla)] = $\sqcup \circ$ Close(fc, pla_0, $\{s_0\}$).

The proof of the theorem is by a structural induction showing P-Cmd(cmd) and P-Exp(exp). Predicate P-Cmd $\in$ Cmd -t> {true, false} is defined by P-Cmd(cmd) = $\forall$ c $\in$ C: $\forall$ S' $\subseteq$ S:

pla' maximal and occ maximal and pla' not a descendant of occ $\Rightarrow$
$\sqcup\{\mathcal{C}[\![cmd]\!](occ)(c)(s)\mid s \in S'\} = \sqcup\{c(s)\mid s \in cl<occ, "R">\} \sqcup (\sqcup \circ cl)$
where cl = Close(F$\mathcal{C}$[[cmd]] occ <pla', λs. s> $\downarrow$2, pla', S').

Predicate P-Exp is defined similarly. Clearly the theorem follows from P-Cmd(cmd) because fin = $\perp$.

Since this proof is long, only the most difficult case will be considered. It amounts to showing P-Cmd(WHILE exp DO cmd OD) assuming P-Cmd(cmd) and P-Exp(exp). So let occ be maximal and abbreviate $pla_0, \ldots, pla_6$ and tft, tff as in figure 4. Further abbreviate $g[c] = \lambda c'. g_1\{cond(g_2 c', g_3 c)\}$ where $g_1 = \mathcal{E}[\![exp]\!]$ occ§<1> and $g_2 = \mathcal{C}[\![cmd]\!]$ occ§<2> and g_3 = attach(pla_6). Similarly fg = fg_1 * Fcond(fg_2, fg_3) where fg_1 = F$\mathcal{E}$[[exp]] occ§<1> and fg_2 = F$\mathcal{C}$[[cmd]] occ§<2> and fg_3 = Fattach(pla_6).

Lemma 1: For arbitrary c, c' $\in$ C and S' $\subseteq$ S:

$$\sqcup\{g[c]c'\ s\mid s \in S'\} = \sqcup\{c(s)\mid s \in cl(pla_6)\} \sqcup \sqcup\{c'(s)\mid s \in cl(pla_5)\} \sqcup (\sqcup \circ cl)$$

where cl = Close(fg<pla_1, λs. s> $\downarrow$2, pla_1, S').

Proof: Abbreviate $cl_2[S''] = Close(fg_2<pla_3, \lambda s.\, s> \downarrow 2,\ pla_3, S'')$. Then by P-Cmd(cmd), $\sqcup\{g_2(c')(s) \mid s \in S''\} = \sqcup\{c'(s) \mid s \in cl_2[S'']pla_5\} \sqcup (\sqcup \circ cl_2[S''])$. Abbreviate $cl_3[S''] = Close(fg_3<pla_3, \lambda s.\, s> \downarrow 2, pla_3, S'')$ so that $\sqcup\{g_3(c)(s) \mid s \in S''\} = \sqcup\{c(s) \mid s \in cl_3[S'']pla_6\} \sqcup (\sqcup \circ cl_3[S''])$. Then $\sqcup\{cond(g_2\, c',\ g_3\, c)\, s \mid s \in S''\}$

$= \sqcup\{c'(s) \mid s \in cl_2[\{tft(s) \mid s \in S''\}]\ pla_5\} \sqcup (\sqcup \circ cl_2[\{tft(s) \mid s \in S''\}]) \sqcup$
$\sqcup\{c(s) \mid s \in cl_3[\{tff(s) \mid s \in S''\}]\ pla_6\} \sqcup (\sqcup \circ cl_3[\{tff(s) \mid s \in S''\}])$

$= \sqcup\{c'(s) \mid s \in (cl_2'[S''] \sqcup cl_3'[S'])\ pla_5\} \sqcup$
$\sqcup\{c(s) \mid s \in (cl_2'[S''] \sqcup cl_3'[S'])\ pla_6\} \sqcup$
$(\sqcup \circ (cl_2'[S''] \sqcup cl_3'[S'']))$

where $cl_2'[S''] = Close(fg_2<pla_3, tft> \downarrow 2,\ pla_3,\ S'') = cl_2[\{tft(s) \mid s \in S''\}]$ and $cl_3'[S''] = Close(fg_3<pla_3, tff> \downarrow 2,\ pla_3,\ S'') = cl_3[\{tff(s) \mid s \in S''\}]$. The last step is by the definition of Close and the properties of FC mentioned in section 3.

It will now be shown that $cl_2'[S''] \sqcup cl_3'[S''] = cl'[S'']$ where $cl'[S''] = Close(Fcond(fg_2, fg_3)<pla_3, \lambda s.\, s> \downarrow 2,\ pla_3,\ S'') = Close(fg_2<pla_3, tft> \downarrow 2 \cup fg_3<pla_3, tff> \downarrow 2,\ pla_3,\ S'')$. Inequality $\sqsubseteq$ is immediate since Close is monotone in its first argument. To establish the converse inequality consider any sequence $<pla_0', pla_1', tf_1'>, \ldots, <pla'_{n-1}, pla'_n, tf_n'>$ that is in $fg_2<pla_3, tft> \downarrow 2 \cup fg_3<pla_3, tff> \downarrow 2$ and has $pla_0' = pla_3$. Such a sequence is said to be "mentioned" in $cl'[S''](pla_n')$. It is not difficult to see (using properties of FC) that the sequence is entirely either in $fg_2<pla_3, tft> \downarrow 2$ (hence "mentioned" in $cl_2'[S''](pla_n')$) or in $fg_3<pla_3, tff> \downarrow 2$ (hence "mentioned" in $cl_3'[S''](pla_n')$). Thus $cl'[S''](pla_n') \sqsubseteq cl_2'[S''](pla_n') \cup cl_3'[S''](pla_n')$.

Next, abbreviate $cl_1[S'] = Close(fg_1<pla_1, \lambda s.\, s> \downarrow 2, pla_1, S')$. Then P-Exp(exp) asserts (with S'' above being $cl_1[S'](pla_3)$) that $\sqcup\{g[c]c'(s) \mid s \in S'\}$

$= \sqcup\{c'(s) \mid s \in (cl_1[S'] \sqcup cl'[cl_1[S']pla_3])\ pla_5\} \sqcup$
$\sqcup\{c(s) \mid s \in (cl_1[S'] \sqcup cl'[cl_1[S']pla_3])\ pla_6\} \sqcup$
$(\sqcup \circ (cl_1[S'] \sqcup cl'[cl_1[S'](pla_3)]))$

where properties of FC and Close assure $cl_1[S'](pla_5) = \emptyset = cl_1[S'](pla_6)$.

Finally, it must be shown that $cl_1[S'] \sqcup cl'[cl_1[S']pla_3] = cl[S']$. Consider the inequality $\sqsupseteq$. Again the key to the proof is to consider a sequence $\langle pla_0', pla_1', tf_1' \rangle, \ldots, \langle pla_{n-1}', pla_n', tf_n' \rangle$ that is "mentioned" in $cl[S'](pla_n')$ and has $pla_0' = pla_1$. If pla_n' is a descendant of $occ\S\langle 1 \rangle$ the properties of $F\mathcal{E}$ and $F\mathcal{C}$ assert that all of $pla_1', \ldots, pla_n'$ are, so the entire sequence is "mentioned" in $cl_1[S'](pla_n')$. Otherwise there is some $m < n$ so $pla_1', \ldots, pla_m'$ are all the descendants of $occ\S\langle 1 \rangle$ (again by properties of $F\mathcal{E}$ and $F\mathcal{C}$). Then $\langle pla_0', pla_1', tf_1' \rangle, \ldots, \langle pla_{m-1}', pla_m', tf_m' \rangle$ is "mentioned" in $cl_1[S'](pla_m')$. Also $pla_m' = pla_3$ (because $\langle pla_m', pla_{m+1}', tf_{m+1}' \rangle$ is in $fg_2\langle pla_3, tft \rangle \downarrow 2 \cup fg_3\langle pla_3, tff \rangle \downarrow 2$) so exp is left through pla_3 and not entered again. Also $\langle pla_m', pla_{m+1}', tf_{m+1}' \rangle, \ldots, \langle pla_{n-1}', pla_n', tf_n' \rangle$ is "mentioned" in $cl'[cl_1[S'](pla_3)](pla_n')$. From this $cl_1[S'](pla_n') \cup cl'[cl_1[S'](pla_3)](pla_n') \supseteq cl[S'](pla_n')$. The converse inequality is similar. □

In the next lemma the iterative nature of WHILE is dealt with. The proof makes use of $Luk \in FC \times Pla \times \mathcal{P}(S) \times Integer \times FC \text{ -t> } Pla \text{ -t> } \mathcal{P}(S)$ that is like Close but constrains the number of times some part of the flowchart is traversed. It is defined by $Luk(fc_1, pla', S', k, fc_2)(pla) =$

$$\{tf_n'(\ldots(tf_1'(s)))\,|\,s \in S' \wedge \langle pla_0', pla_1', tf_1' \rangle, \ldots, \langle pla_{n-1}', pla_n', tf_n' \rangle \in fc_1 \cup fc_2 \wedge pla_0' = pla' \wedge pla_n' = pla \wedge |\{i\,|\,\langle pla_{i-1}', pla_i', tf_i' \rangle \in fc_2\}| = k\}$$

Lemma 2: For any $c \in C$ and $S' \subseteq S$:

$$\sqcup\{FIX(g[c])s\,|\,s \in S'\} = \sqcup\{c(s)\,|\,s \in cl(pla_6)\} \sqcup (\sqcup \circ cl)$$

where $cl = Close((fg\ Ffix\langle pla_5, pla_2, \lambda s.\, s \rangle)\langle pla_1, \lambda s.\, s \rangle \downarrow 2,\ pla_1,\ S')$.

Proof: Abbreviate $lu[k] = Luk(fg\langle pla_1, \lambda s.\, s \rangle \downarrow 2,\ pla_1,\ S',\ k,\ \{\langle pla_5, pla_2, \lambda s.\, s \rangle\})$. It corresponds to executing the body of the WHILE k+1 times. Since $cl = \sqcup\{lu[k]\,|\,k \geq 0\}$ it suffices to show by induction in k that $\sqcup\{(g[c])^{k+1} \perp s\,|\,s \in S'\} = \sqcup\{c(s)\,|\,s \in lu[k](pla_6)\} \sqcup (\sqcup \circ lu[k]) \sqcup \ldots \sqcup (\sqcup \circ lu[0])$.

The case $k = 0$ is by lemma 1. For the inductive step the hypothesis and lemma 1 establish

$\sqcup\{(g[c])^{k+1} \perp s \mid s \in S'\}$

$= \quad \sqcup\{c(s) \mid s \in cl'(pla_6)\} \sqcup (\sqcup \circ cl') \sqcup (\sqcup \circ lu[k]) \sqcup \ldots \sqcup (\sqcup \circ lu[0])$

for $cl' = Close(fg\langle pla_1, \lambda s.\, s\rangle \downarrow 2,\ pla_1,\ lu[k](pla_6))$. The result follows from $cl' = lu[k+1]$ which can be shown by the methods used in the proof of lemma 1. □

From lemma 2 P-Cmd(WHILE exp DO cmd OD) easily follows.

Appendix 2: Proof of theorem 5

In this appendix the suffix ind is omitted. The proof is by structural induction showing P-Cmd(cmd) and P-Exp(exp). Predicate P-Cmd $\in$ Cmd -t> $\{$true, false$\}$ is defined by P-Cmd(cmd) = $\forall s \in S$: pla_0 maximal and occ maximal and pla_0 not a descendant of occ $\Rightarrow$

(i) $DC[\![cmd]\!](occ)(s) \downarrow 1 = DC[\![cmd]\!](occ)(s) \downarrow 2 <occ, "R">$

(ii) $DC[\![cmd]\!](occ)(s) \downarrow 2 = MFPS(FC[\![cmd]\!](occ)<pla_0, \lambda s.\, s> \downarrow 2, pla_0, s)$.

Predicate P-Exp is defined similarly.

Since the proof is long only the most difficult case will be considered. It amounts to showing P-Cmd(WHILE exp DO cmd OD) assuming P-Cmd(cmd) and P-Exp(exp). Assume that occ is maximal and let $pla_1, \ldots, pla_6$, tft, tff, fg_1, fg_2, fg_3 and fg be as in appendix 1. Also abbreviate $dg_1 = DC[\![exp]\!]occ\S<1>$ and $dg_2 = DC[\![cmd]\!]occ\S<2>$ and $dg_3 = Dattach(pla_6)$ and $dg[\, dg' \,] = \lambda s.\, [\, dg_1 * Dcond(dg_2 * (\lambda s'.\, dg'(s' \sqcup s)), dg_3)]s$. The following fact is frequently used without explicit mentioning.

Fact: step(fc, ···, ···)(a)(pla) = $\perp$ if there is no pla' and tf so <pla', pla, tf> $\in$ fc.

Lemma 3: $\forall s \in S$: $FIX(\lambda dg'.\, dg[\, dg' \,])s \downarrow 1 = FIX(\lambda dg'.\, dg[\, dg' \,])s \downarrow 2\ pla_6$

Proof: The lemma follows from

$$\forall\, s \in S: (\lambda dg'.\, dg[dg'])^n \perp s \downarrow 1 = (\lambda dg'.\, dg[dg'])^n \perp s \downarrow 2\ pla_6$$

which is proved by induction in n. The case n = 0 is trivial so consider the inductive step. For arbitrary $s \in S$ it follows from P-Cmd(cmd) and P-Exp(exp) that $(\lambda dg'.\, dg[dg'])^{n+1} \perp s = (\lambda dg'.\, dg[dg'])^n \perp (s \sqcup a_2(pla_5)) \sqcup dg_3(tff(a_1(pla_3))) \sqcup <\perp, a_1 \sqcup a_2>$ where $a_1 = dg_1(s) \downarrow 2$ and $a_2 = dg_2(tft(a_1(pla_3))) \downarrow 2$. It is easy to see that $a_1(pla_6) = \perp = a_2(pla_6)$ because of the fact mentioned above. Hence the result follows. □

From lemma 3 condition (i) of P-Cmd(WHILE exp DO cmd OD) easily follows. For condition (ii) it is useful to abbreviate $fc_1 = fg_1<pla_1, \lambda s.\, s> \downarrow 2$, $fc_2 = fg_2<pla_3, tft> \downarrow 2$, $fc_3 = fg_3<pla_3, tff> \downarrow 2$ and $fc_4 = \{<pla_5, pla_2, \lambda s.\, s>\}$.

Then define

$$M_0(s) = MFPS(fg\langle pla_1, \lambda s.\, s\rangle \downarrow 2,\ pla_1,\ s) = MFPS(fc_1 \cup fc_2 \cup fc_3,\ pla_1,\ s).$$

Intuitively $M_0(s)$ is the effect of the WHILE construct when no iterations are performed.

Lemma 4: For arbitrary $dg' \in S \text{ -m> } (S \times (Pla \text{ -c> } S))$ and $s \in S$:
$(dg[dg'](s)) \downarrow 2 = dg'\,(M_0(s)(pla_5) \sqcup s) \downarrow 2 \sqcup M_0(s)$.

Proof: From P-Exp(exp) follows $dg_1(s) = \langle a_1(pla_3), a_1\rangle$ where $a_1 = MFPS(fc_1, pla_1, s)$. From P-Cmd(cmd) follows $dg_2(tft(a_1(pla_3))) = \langle a_2(pla_5), a_2\rangle$ where $a_2 = MFPS(fc_2, pla_3, a_1(pla_3))$. It is easy to see $dg_3(tff(a_1(pla_3))) = \langle a_3(pla_6), a_3\rangle$ where $a_3 = MFPS(fc_3, pla_3, a_1(pla_3))$. Since $a_1(pla_5) = \bot = a_3(pla_5)$ this yields $(dg[dg'](s)) \downarrow 2 = a_1 \sqcup a_2 \sqcup a_3 \sqcup (dg'((a_1 \sqcup a_2 \sqcup a_3)(pla_5) \sqcup s) \downarrow 2)$.

It remains to be shown that $a_1 \sqcup a_2 \sqcup a_3 = M_0(s)$. First it will be shown that $a_2 \sqcup a_3 = a'$, where $a' = MFPS(fc_2 \cup fc_3,\ pla_3,\ a_1(pla_3))$. Then it will be shown that $a_1 \sqcup a' = M_0(s)$.

That $a_2 \sqcup a_3 \sqsupseteq a'$ follows from
$(a_2 \sqcup a_3)pla \sqsupseteq step(fc_2 \cup fc_3, pla_3, a_1(pla_3))(a_2 \sqcup a_3)pla$ which is shown by cases of pla. Consider the case where pla is a descendant of $occ\S\langle 2\rangle$. Then (using the fact and properties of FC)

$$\begin{aligned}(a_2 \sqcup a_3)pla &= step(fc_2, pla_3, a_1(pla_3))\ a_2\ pla \\ &= step(fc_2, pla_3, a_1(pla_3))\ (a_2 \sqcup a_3)\ pla \\ &= step(fc_2 \cup fc_3, pla_3, a_1(pla_3))(a_2 \sqcup a_3)\ pla\end{aligned}$$

The remaining cases are similar. The converse inequality $a_2 \sqcup a_3 \sqsubseteq a'$ follows from $a_2 \sqsubseteq a'$ and $a_3 \sqsubseteq a'$, which are easy to establish because step is monotone in its first argument.

That $a_1 \sqcup a' \sqsupseteq M_0(s)$ follows from
$(a_1 \sqcup a')pla \sqsupseteq step(fc_1 \cup fc_2 \cup fc_3, pla_1, s)(a_1 \sqcup a')pla$ which is shown by cases of pla. Consider the case where pla is a descendant of $occ\S\langle 2\rangle$. Then

$$(a_1 \sqcup a')pla = step(fc_2 \cup fc_3, pla_3, a_1(pla_3))\ a'\ pla$$
$$= step(fc_2 \cup fc_3, pla_1, s)\ (a_1 \sqcup a')\ pla$$
$$= step(fc_1 \cup fc_2 \cup fc_3, pla_1, s)\ (a_1 \sqcup a')\ pla$$

The remaining cases are similar. That $a_1 \sqcup a' \sqsubseteq M_0(s)$ is by $a_1 \sqsubseteq M_0(s)$ (which is easy) and $a' \sqsubseteq M_0(s)$. The latter follows from $M_0(s)(pla) \sqsupseteq step(fc_2 \cup fc_3, pla_3, a_1(pla_3))(M_0(s))(pla)$. This is proved by cases of pla using $a_1 \sqsubseteq M_0(s)$ and monotonicity of the transfer functions. □

To express the effect of the WHILE loop when it is iterated an arbitrary number of times it is useful to define $M_\infty(s) = MFPS(\ (fg\ Ffix <pla_5, pla_2, \lambda s.\ s>)\ <pla_1, \lambda s.\ s> \downarrow 2, pla_1, s) = MFPS(fc_1 \cup fc_2 \cup fc_3 \cup fc_4, pla_1, s)$.

Lemma 5: For arbitrary $s \in S$:

$$FIX(\lambda dg'.\ dg[dg'])\ s \downarrow 2 = M_\infty(s)$$

Proof: It simplifies the proof to omit the first component of $dg[dg']s$ from consideration. To this end define $M[M']s = M'(M_0(s)(pla_5) \sqcup s) \sqcup M_0(s)$. Then using lemma 4 it is not difficult to show $FIX(\lambda dg'.\ dg[dg'])\ s \downarrow 2 = FIX(\lambda M'.\ M[M'])\ s = LFP\ (\lambda M'.\ M[M'])\ s$.

That $FIX(\lambda M'.\ M[M'])\ s \sqsubseteq M_\infty(s)$ is obtained from $M_\infty(s) \sqsupseteq M_\infty(M_0(s)(pla_5) \sqcup s) \sqcup M_0(s)$. Since $step(fc_1 \cup fc_2 \cup fc_3 \cup fc_4, pla_1, s) \sqsupseteq step(fc_1 \cup fc_2 \cup fc_3, pla_1, s)$ it follows that $M_\infty(s) \sqsupseteq M_0(s)$. By the methods used in lemma 4 it can be shown that

$$M_\infty(s) \sqsupseteq step(fc_1 \cup fc_2 \cup fc_3 \cup fc_4, pla_1, M_\infty(s)(pla_5) \sqcup s)(M_\infty(s))$$

so that $M_\infty(s) \sqsupseteq M_\infty(M_\infty(s)(pla_5) \sqcup s) \sqsupseteq M_\infty(M_0(s)(pla_5) \sqcup s)$.

The converse inclusion $FIX(\lambda M'.\ M[M'])\ s \sqsupseteq M_\infty(s)$ follows from $FIX(\lambda M'.\ M[M'])\ s \sqsupseteq step(fc_1 \cup fc_2 \cup fc_3 \cup fc_4, pla_1, s)(FIX(\lambda M'.\ M[M'])\ s)$. By the continuity of α and γ it follows that $step(\cdots, \cdots, \cdots)$ is continuous. It then suffices to show $(\lambda M'.\ M[M'])^{n+1}\ \bot\ s \sqsupseteq step(fc_1 \cup fc_2 \cup fc_3 \cup fc_4, pla_1, s)((\lambda M'.\ M[M'])^n\ \bot\ s)$. Define $b_1(s) = M_0(s)$ and $b_{n+1}(s) = M_0(b_n(s)(pla_5) \sqcup s)$. Since it is easily seen that b_n is monotone and $b_{n+1} \sqsupseteq b_n$ it can be shown (by induction in n) that $\forall s: b_n(M_0(s)(pla_5) \sqcup s) = M_0(b_n(s)(pla_5) \sqcup s)$ so that $\forall s: (\lambda M'.\ M[M'])^n\ \bot\ s = b_n(s)$ follows. Finally $b_{n+1}(s)(pla) \sqsupseteq step(fc_1 \cup fc_2 \cup fc_3 \cup fc_4, pla_1, s)$

$(b_n(s))(pla)$ is shown by cases of pla. □

From lemma 5 condition (ii) of P-Cmd(WHILE exp DO cmd OD) easily follows.