ISSN 0105-8517

SEMANTIC FOUNDATIONS OF DATA FLOW ANALYSIS

by

Flemming Nielson

DAIMI PB-131

February 1981

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

ABSTRACT:

Abstract Interpretation (P.Cousot, R.Cousot and others) is a method for program analysis that is able to describe many data flow analyses. We investigate and weaken the assumptions made in abstract interpretation and express abstract interpretation within Denotational Semantics. As an example we specify constant propagation.

Some authors have used abstract interpretation to formulate "available expressions" (a so-called "history-sensitive" data flow analysis). Our development of "available expressions" is better justified, semantically.

In traditional data flow analysis and abstract interpretation it is generally assumed that the "Meet Over all Paths" solution is wanted. We prove that the solution specified by our approach is the "Meet Over all Paths" solution to a certain system of equations obtained from the program.

To indicate the usefulness of our approach we show how to validate a class of program transformations, including "constant folding".

Throughout this paper we use a toy language consisting of declarations, expressions and commands (involving conditional and iteration). Excluded are procedures and jumps.

KEYWORDS:

denotational semantics, non-standard semantics, data flow analysis, abstract interpretation, constant propagation, available expressions, meet over all paths solution, program transformation, constant folding.

ACKNOWLEDGEMENT:

This paper is my Masters Thesis (danish "speciale"). I wish to thank my supervisor Neil Jones for his support during my study. Personal thanks are due to Hanne Riis.

CONTENTS:

Index to the notation and the concepts.

We do not list notation or concepts that is more or less standard. We generally omit notation that is only used in the section where it is defined. References are to pages (1:2), tables (2.2-A) or definitions etc. (2.1.3-10).

CHAPTER 1

Introduction

Efficient implementation of programming languages necessitates program transformations. The transformations may be between intermediate representations of some source program or between source programs (source-to-source transformations). The purpose of the transformations is to obtain a program that somehow is "better" than the original: it can be interpreted more efficiently or more efficient code can be generated from it.

Data Flow Analysis is necessary for all but the simplest program transformations. To detect the applicability of a transformation some information about the program is usually required. It is the purpose of data flow analysis to collect that information.

Often, there is a close connection between program transformations and data flow analyses. Data flow analysis is, however, also studied in its own right. This is mostly the approach we take. We develop a framework that can express several data flow analyses for an example high-level language, and we give semantic characterizations of "constant propagation" and "available expressions". The main ingredients in our approach are Abstract Interpretation and Denotational Semantics.

Literature on program analysis

Early development of data flow analysis was rather ad-hoc. The advent of lattice theoretic formulations of data flow analysis provided an important formalization. It made it possible to succinctly express the desired data flow information without stating which algorithm should compute it. Furthermore, the lattice theoretic approach extended the class of data flow analysis problems that could be handled.

But even in the lattice theoretic approach (e.g. [KaU77], [Kil73]) the data flow analysis is syntactic in nature: One is unable to formally verify the correctness of the obtained data flow information, i.e. that claims made by data flow analysis about executions of programs are indeed satisfied by any program execution. This implies that there is no way to make sure that each and every pitfall has been considered. A simple example is to make available expression analysis without consideration of possible sharing.

Abstract Interpretation, like data flow analysis, deals with program analysis. It is described in the "Cousot papers": [CoC77a], [CoC77b], [CoC77c], [CoH78], [CoC79], [Cou79] and similar ideas appear in [Sin72] and [Weg75]. Like the lattice theoretic approach to data flow analysis it distinguishes between specifying the desired information and specifying how to compute it. Unlike data flow analysis, the abstract interpretation approach is semantic in nature; This makes it possible to prove the correctness of data flow analysis information.

Data flow analyses may, somewhat informally, be classified as being "history-insensitive" or "history-sensitive". The phrase "history-sensitive" is used in [Coc79]; below we briefly explain the intentions with the phrases. "History-insensitive" analyses are analyses such as constant propagation, that characterize the sets of states that reach some point in the program. Many papers apply abstract interpretation to formulate, and prove correct, analyses of this kind. "History-sensitive" analyses may be forward analyses like available expressions, that characterize the computational history, or backward analyses such as live variables, that characterize the computational future. The attempts to handle "history-sensitive" analyses by means of abstract interpretation ([CoC79,p.278], [CoC77a,p.241], [Weg75,p.276]) are less satisfactory: the interpretations may be formulated but correctness is not considered or only considered inadequately.

Overview of this paper

In this paper we give a semantic characterization of typical data flow analyses. The semantic foundation will be Denotational Semantics. This seems to be an appropriate choice for two reasons: Firstly, Denotational Semantics is widely used to define languages. It appears to be a better choice than e.g. operational semantics (see [Sto77, chapter2] for a discussion). Secondly, our development may be related to the compiler development of [MiS76]. As indicated in [Nie79,p.1.45] data flow analysis may be relevant to this approach to compiler development.

Two key papers that have influenced our work are [CoC79] and [Don79]. Cousot&Cousot [CoC79] describe abstract interpretation, which is the method we use to specify data flow analyses. Donzeau-Gouge [Don79] defines non-standard semantics that specify various data flow analyses. Part of our initial motivation was to unify the approach of [Don79] with abstract interpretation [CoC79]. For those parts of [Don79] we have considered we believe that our approach is more systematic and more general than her approach and that it provides a better link to "traditional data flow analysis".

In chapter 2 we review the foundations for this paper. We survey parts of lattice theory: definitions, functions between complete lattices, and construction of complete lattices. Our use of denotational semantics is explained: the notation, the mathematical foundations and the kind of semantics (store semantics in continuation style). Key notions from "traditional" data flow analysis are mentioned: constant propagation, available expressions, MOP (Meet Over all Paths) solution and MFP (Maximal Fixed Point) solution. Our explanation of abstract interpretation extends that given in [CoC79] and we weaken some of the requirements that are normally imposed. Furthermore, we describe some example concretization functions that will be used later.

In chapter 3 we formulate abstract interpretation in the framework of denotational semantics. This generalizes parts of [Don79] and [Don78]. We illustrate our approach by two examples of "constant propagation". One is close to that of [Don79], the other is reasonable close to the traditional notion of constant propagation.

In chapter 4 we show how "available expressions" may be handled satisfactorily, i.e. so that correctness can be proven (in some suitable sense). This shows that abstract interpretation is applicable to some forward "history-sensitive" analyses.

The MOP solution is generally considered the desired solution to a data flow analysis. In chapter 5 we show that our approach does specify the MOP solution.

In chapter 6 we show how our approach can be used to validate a class of program transformations (including "constant folding"). This is briefly related to the methods of other papers.

Finally, chapter 7 contains the conclusions, including an assessment of problems in our approach.

Requirements upon the reader

Chapter 2 mostly is intended as a review. This implies that the reader must be acquainted with lattice theory, denotational semantics, data flow analysis and abstract interpretation. We detail these requirements below.

We expect the reader to have a good knowledge of Denotational Semantics including proof methods (structural induction) and mathematical foundations (lattice theory). This is covered by either [Sto77] or [MiS76], whereas [Gor79] and [Ten76] only cover the use of Denotational Semantics.

Data flow analysis is described in the text books [Hec77] and [AhU78] and the survey paper [Ull75]. We expect the reader to have encountered terms like "constant propagation", "available expressions" and "MOP solution".

By abstract interpretation we understand the framework described in the "Cousot papers" (listed above). It is preferable if the reader is acquainted with at least one of these papers. We do not expect the reader to be acquainted with [Don79] or [Don78].

How to read this paper

To understand chapter 3 the material of chapter 2 is needed. However, subsection 2.4.2 can be postponed until it is needed in section 3.2. Subsection 2.4.3 can be omitted on a first reading. Sections 2.1 and 2.3 need only be skimmed if the reader has sufficient background in lattice theory and data flow analysis.

Chapters 4, 5 and 6 are mostly independent of one another. So it should be possible to read them in any order.

CHAPTER 2

Foundations

This chapter contains a review of foundational material that will be needed. In section 2.1 we review parts of lattice theory. Then (in section 2.2) we introduce our toy language and explain its denotational semantics. In section 2.3 we briefly explain the notions from data flow analysis that we will need. Finally (in section 2.4) we present abstract interpretation. We define some example concretization functions that will be needed later. However, parts of section 2.4 are not review but further development of abstract interpretation.

2.1 Lattice Theory

In subsection 2.1.1 we define the fundamental notions of complete lattices. Then (in subsection 2.1.2) we consider functions upon complete lattices and we review some of the properties of these functions. Finally (in subsection 2.1.3) we consider methods for constructing non-reflexive complete lattices.

The definitions and results of this section are used throughout the paper but an explicit reference is rarely given.

We denote by B the set {true,false}. We use the conditional IF truthvalue THEN truecase ELSE falsecase and sometimes in the form truthvalue -> truecase, falsecase. We use ordinary mathematical notation like $\forall$, $\exists$, $=$, $\neq$, $\wedge$ and $\vee$. We denote by $\mathcal{P}$(L) the power-set of L and by L1 X L2 the cartesian product of sets L1 and L2. We use λ-notation freely [Sto77]. We only consider total functions, so f:L->M means that f is a total function from L to M. Functional composition is denoted f$\circ$g and means λx.f(g(x)). Free variables in formulae are universally quantified. For typographical reasons we often write x[i] or xi instead of x_i.

2.1.1 Fundamental Definitions

DEFINITION 2.1.1-1: (L,$\sqsubseteq$) is a partially ordered set iff

a) L is a set

b) $\sqsubseteq$ is a partial order on L, i.e.

- $\sqsubseteq$ is reflexive ($\forall$l$\in$L: l$\sqsubseteq$l)
- $\sqsubseteq$ is anti-symmetric ($\forall$l1,l2$\in$L: l1$\sqsubseteq$l2 $\wedge$ l2$\sqsubseteq$l1 => l1=l2)
- $\sqsubseteq$ is transitive ($\forall$l1,l2,l3$\in$L: l1$\sqsubseteq$l2 $\wedge$ l2$\sqsubseteq$l3 => l1$\sqsubseteq$l3) []

When $\underline{l}\subseteq$L and l$\in$L {a} we abbreviate $\forall$l'$\in\underline{l}$: l$\sqsubseteq$l' to l$\sqsubseteq\underline{l}$ and $\forall$l'$\in\underline{l}$: l'$\sqsubseteq$l to $\underline{l}\sqsubseteq$l. We define x$\sqsupseteq$y to mean y$\sqsubseteq$x.

{a} In general, if variable v ranges over a set V, then $\underline{v}$ will range over the power-set ($\mathcal{P}$(V)) of V.

DEFINITION 2.1.1-2: $(L,\sqsubseteq)$ is a <u>complete lattice</u> iff
a) $(L,\sqsubseteq)$ is a partially ordered set
b) $\forall \underline{l}\subseteq L: \exists l\in L: \forall l'\in L: \underline{l}\sqsubseteq l' \Leftrightarrow l\sqsubseteq l'$
c) $\forall \underline{l}\subseteq L: \exists l\in L: \forall l'\in L: l'\sqsubseteq \underline{l} \Leftrightarrow l'\sqsubseteq l$ □

Case b) says that for any $\underline{l}$ there is at least one l such that $\underline{l}\sqsubseteq l' \Leftrightarrow l\sqsubseteq l'$. By anti-symmetry of $\sqsubseteq$ it follows that there is exactly one l, called <u>the least upper bound</u> of $\underline{l}$. The least upper bound of $\underline{l}$ is usually denoted $\bigsqcup\underline{l}$, where $\bigsqcup$ is called <u>join</u>. Similar remarks apply to c): the <u>greatest lower bound</u> of $\underline{l}$ is denoted $\bigsqcap\underline{l}$, where $\bigsqcap$ is called <u>meet</u>. This motivates:

DEFINITION 2.1.1-3: For a complete lattice $(L,\sqsubseteq)$ define:
a) $\bigsqcup:\mathcal{P}(L)\to L$ by $\forall \underline{l}\subseteq L: \forall l\in L: \underline{l}\sqsubseteq l \Leftrightarrow \bigsqcup\underline{l}\sqsubseteq l$
b) $\bigsqcap:\mathcal{P}(L)\to L$ by $\forall \underline{l}\subseteq L: \forall l\in L: \underline{l}\sqsupseteq l \Leftrightarrow \bigsqcap\underline{l}\sqsupseteq l$
c) $\bot=\bigsqcup\emptyset=\bigsqcap L$ (called "<u>bottom</u>") and $\top=\bigsqcap\emptyset=\bigsqcup L$ (called "<u>top</u>")
d) $\forall l,l'\in L: l\sqcup l'=\bigsqcup\{l,l'\}$ ($\sqcup$ is also called join)
and $l\sqcap l'=\bigsqcap\{l,l'\}$ ($\sqcap$ is also called meet) □

For a complete lattice $(L,\sqsubseteq)$ each symbol $\sqsubseteq,\bigsqcup,\bigsqcap,\bot,\top,\sqcup,\sqcap$ ought to have been indexed by L. To avoid excessive notation we elide these indices. If $(M,\sqsubseteq)$ is also a complete lattice then we expect the two $\sqsubseteq$ to be different partial orders, and we expect the two $\bot$ to be distinct, etc. We often abbreviate $(L,\sqsubseteq)$ to L.

DEFINITION 2.1.1-4: $(L,\sqsubseteq)$ is a <u>flat lattice</u> iff there are $\bot,\top\in L$ (and $\bot\neq\top$) so that $\forall l,l'\in L: l\sqsubseteq l' \Leftrightarrow (l=\bot \vee l=l' \vee l'=\top)$. □

OBSERVATION 2.1.1-5: Any flat lattice is a complete lattice. □

EXAMPLE 2.1.1-6: Define $N=\{\bot,\top,0,1,2,\ldots\}$ where we tacitly assume that $\bot,\top,0,1,2,\ldots$ are distinct elements. Define $n\sqsubseteq n'$ by $(n=\bot \vee n=n' \vee n'=\top)$. Then N is a flat lattice.

Define $T=\{\bot,\top,true,false\}$ and $t\sqsubseteq t'$ by $(t=\bot \vee t=t' \vee t'=\top)$. Then T is a flat lattice.

Define $Q=\{\bot,\top\}\cup\bigcup\{\text{textstrings of length } m \mid m\geq 0\}$ and $q\sqsubseteq q' \Leftrightarrow (q=\bot \vee q=q' \vee q'=\top)$. Then Q is a flat lattice. □

A useful result is:

LEMMA 2.1.1-7 If L is a complete lattice and $X\subseteq\mathcal{P}(L)$ then
$\bigsqcup\{\bigsqcup\underline{l} \mid \underline{l}\in X\} = \sqcup(\bigcup\{\underline{l} \mid \underline{l}\in X\})$
$\bigsqcap\{\bigsqcap\underline{l} \mid \underline{l}\in X\} = \sqcap(\bigcup\{\underline{l} \mid \underline{l}\in X\})$ □

(Note $\bigcup$ in both equations).

<u>2.1.2 Functions Upon Complete Lattices</u>

In the sequel we assume that $(L,\sqsubseteq)$, $(Li,\sqsubseteq)$ and $(M,\sqsubseteq)$ are complete lattices.

DEFINITION 2.1.2-1: A function $f:L\to M$ is <u>isotone</u> (monotone [Sto77]) iff $\forall l1,l2\in L: l1\sqsubseteq l2 \Rightarrow f(l1)\sqsubseteq f(l2)$. □

DEFINITION 2.1.2-2: A set $\underline{l}\subseteq L$ is <u>directed</u> iff $\forall \text{finite}\,\underline{l}'\subseteq\underline{l}\ \exists l'\in\underline{l}: \underline{l}'\sqsubseteq l'$. □

DEFINITION 2.1.2-3: A function $f:L\to M$ is

a) a $\sqcup$-<u>morphism</u> (distributive [KaU77]) iff
$\forall$finite, nonempty $\underline{L}\sqsubseteq L$: $f(\sqcup\underline{L}) = \sqcup\{f(l)\mid l\in\underline{L}\}$

b) <u>continuous</u> iff $\forall$directed $\underline{L}\sqsubseteq L$: $f(\sqcup\underline{L})= \sqcup\{f(l)\mid l\in\underline{L}\}$

c) a <u>complete-$\sqcup$-morphism</u> iff $\forall\underline{L}\sqsubseteq L$: $f(\sqcup\underline{L})=\sqcup\{f(l)\mid l\in\underline{L}\}$ []

LEMMA 2.1.2-4: For a function $f:L\to M$ we have (in general):

$$\begin{tikzcd}
 & f\ \text{continuous} \arrow[d, Rightarrow, shift right, "/"'] \arrow[dr, Rightarrow] & \\
f\ \text{complete-}\sqcup\text{-morphism} \arrow[ur, Rightarrow] \arrow[dr, Rightarrow] & & f\ \text{isotone} \\
 & f\ \sqcup\text{-morphism} \arrow[u, Rightarrow, shift right, "/"'] \arrow[ur, Rightarrow] &
\end{tikzcd}$$

f continuous $\wedge$ f $\sqcup$-morphism $\wedge$ $f(\bot)=\bot \Leftrightarrow f$ complete-$\sqcup$-morphism []
Proof is shown in appendix 2. []

Note that Lemma 2.1.2-4 (or rather its dual -- see later) contradicts Observation 2.18 and Figure 2.7 of [Hec77].

DEFINITION 2.1.2-5: A function $f:L\to M$ is a <u>complete-$\sqcap$-morphism</u> iff
$\forall\underline{L}\sqsubseteq L$: $f(\sqcap\underline{L}) = \sqcap\{f(l) \mid l\in\underline{L}\}$ []

DEFINITION 2.1.2-6: A function $f:L\to L$ is

a) <u>extensive</u> iff $\forall l\in L$: $f(l) \sqsupseteq l$

b) <u>idempotent</u> iff $\forall l\in L$: $f(f(l))=f(l)$

c) <u>reductive</u> iff $\forall l\in L$: $f(l)\sqsubseteq l$

d) an <u>upper closure operator</u> iff it is isotone, extensive, and idempotent []

DEFINITION 2.1.2-7: A function $f:L\to M$ is <u>strict</u> iff $f\bot=\bot$ and <u>doubly strict</u> iff $f\bot=\bot$ and $f\top=\top$. A function $f:L1\times L2\to M$ is <u>very strict</u> iff it is doubly strict in both arguments, except that $f\bot\top=f\top\bot=\bot$. []

We say that $l\in L$ is <u>proper</u> iff $l\notin\{\bot,\top\}$. If $f:L\to M$ is a function required to be doubly strict then one needs only mention f's definition on proper elements. Similarly, if $f:L1\times L2\to M$ is a function required to be very strict then one needs only mention f's definition on
$\{\langle l1,l2\rangle\mid l1$ proper and $l2$ proper$\}$.

LEMMA 2.1.2-8: For functions $f:L2\to L3$ and $g:L1\to L2$: If f and g are both isotone ($\sqcup$-morphisms, continuous, complete-$\sqcup$-morphisms) then the same holds for $f\circ g$. []

LEMMA 2.1.2-9: Let F be a collection of functions $L\to M$ and $g=\lambda l.\sqcup\{f(l)\mid f\in F\}$. If all $f\in F$ are isotone ($\sqcup$-morphisms, continuous, complete-$\sqcup$-morphisms) then the same holds for g. []

We now consider fixed points of (isotone) functions:

DEFINITION 2.1.2-10: If $f:L \to L$ then $l \in L$ is a <u>fixed point</u> of f iff $f(l)=l$.
Define
$LFP(f)=\sqcap\{l \mid f(l)\sqsubseteq l\}$
$FIX(f)=\sqcup\{f^m(\bot) \mid m\geq 0\}$ where $f^0=\lambda l.l$ and $f^{m+1}=f\circ f^m$. []

THEOREM 2.1.2-11 [Tar55]: If $f:L \to L$ is isotone then
$(\{ l \mid f(l)=l \}, \sqsubseteq)$ is a complete lattice
$LFP(f) = \sqcap\{l \mid f(l)=l\}$ is the bottom of $\{l \mid f(l)=l\}$ []

THEOREM 2.1.2-12 [Sto77][San73]: If $f:L \to L$ is continuous then $FIX(f)=LFP(f)$

It is easy to show that if $(L,\sqsubseteq)$ is a complete lattice then also $(L,\sqsupseteq)$, called the dual of $(L,\sqsubseteq)$, is a complete lattice (partly proved in [Grä71,p.3]). If we make some assertion then the dual assertion is obtained by replacing (concepts defined in terms of) $\sqsubseteq$ with (similar concepts defined in terms of) $\sqsupseteq$. The <u>duality principle</u> says that if some assertion is true in all complete lattices then the dual assertion is also true in all complete lattices [Grä71,p.3][San73,p.19].

The dual assertion of "LFP(f) is the least fixed point of f" is "GFP(f) is the greatest fixed point of f" where $GFP(f)=\sqcup\{l \mid l\sqsubseteq f(l)\}$. Obviously, $\sqcup$ and LFP correspond to $\sqcap$ and GFP in the dual lattice.

2.1.3 Construction of Complete Lattices

We take the ordinary mathematical notion of tupling for granted, e.g. $\langle a,b\rangle$ with $\langle a1,b1\rangle = \langle a2,b2\rangle \Leftrightarrow a1=a2 \wedge b1=b2$; Also that $\langle\rangle$ (the empty tuple) is well-defined.

Cartesian Product

DEFINITION 2.1.3-1: For $n\geq 2$ define $L1\times\ldots\times Ln = \{\langle l1,\ldots,ln\rangle \mid l1\in L1\wedge\ldots\wedge ln\in Ln\}$ and $\langle l1,\ldots,ln\rangle \sqsubseteq \langle l1',\ldots,ln'\rangle \Leftrightarrow l1\sqsubseteq l1'\wedge\ldots\wedge ln\sqsubseteq ln'$. For $i\in\{1,\ldots,n\}$ define $\downarrow i:L1\times\ldots\times Ln \to Li$ by $\langle l1,\ldots,ln\rangle\downarrow i=li$. []

LEMMA 2.1.3-2: $L1\times\ldots\times Ln$ is a complete lattice with
$\sqcup\underline{L} = \langle\ldots,\sqcup\{l\downarrow i \mid l\in\underline{L}\},\ldots\rangle$
$\sqcap\underline{L} = \langle\ldots,\sqcap\{l\downarrow i \mid l\in\underline{L}\},\ldots\rangle$
and $\downarrow i$ is continuous. []
Proof is shown in appendix 2. []

COROLLARY 2.1.3-3: If $fi:L \to Mi$ and $\underline{L}\subseteq L$ then
$\langle\ldots,\sqcup\{fi(l) \mid l\in\underline{L}\},\ldots\rangle = \sqcup\{\langle\ldots,fi(l),\ldots\rangle \mid l\in\underline{L}\}$ []

Separated Sum

DEFINITION 2.1.3-4: For $n\geq 2$ define $L1+\ldots+Ln = \{\bot,\top\}\cup\{\langle i,li\rangle \mid li\in Li\wedge 1\leq i\leq n\}$ with $l\sqsubseteq l' \Leftrightarrow (l=\bot \vee l'=\top\vee(l\downarrow 1=l'\downarrow 1 \wedge l\downarrow 2 \sqsubseteq l'\downarrow 2))$.

Define (informally) "inL1+...+Ln" as the (generic [Gor79]) function $Li \to L1+\ldots+Ln$ so that li inL1+...+Ln = $\langle i,li\rangle$ if $li\in Li$.

Define (informally) ϵLi as the doubly strict function $L1+\ldots+Ln \to T$ so that $\langle j,lj\rangle\epsilon Li = (i=j \to true, false)$.

Define (informally) $|Li$ as the doubly strict function $L1+\ldots+Ln \to Li$ so that $\langle j,lj\rangle|Li = (i=j \to lj, \top)$ [MiS76]. []

The definitions of inL1+...+Ln, ELi and |Li are intended to be as in Denotational Semantics. Note that inL1+...+Ln is ambiguous [MiS76] [Nie79,p.1.7]; the ambiguity will not arise in the situations where we use the notation.

LEMMA 2.1.3-5: L1+...+Ln is a complete lattice. Also inL1+...+Ln, ELi and |Li are continuous. []

We have followed [Sto77] [MiS76] in using the separated sum rather than the coalesced sum [Sto77,p.92,p.110]. In this paper we dispense with error-elements throughout, so + does not introduce a new error element [Sto77,p.145] and T does not contain an error element. The reason is that for our purposes it is unnatural to introduce an error element in $\mathcal{P}$(L) (to be described below) and the convention that ?->l1,l2 is ? (see [Sto77,p.152]) would then require all domains to contain error elements.

DEFINITION 2.1.3-6: For any complete lattice L define $L^\circ=\{\bot,\top\}\cup L$ where $\bot$, $\top$ are distinct elements not in L. Also define $l1 \sqsubseteq l2$ <=> $(l1=\bot \vee l1 \sqsubseteq_L l2 \vee l2=\top)$. []

OBSERVATION 2.1.3-7: L° is a complete lattice. []

We sometimes write S° where S is a set, i.e. a partially ordered set with $\sqsubseteq$ being =. Then S° is a flat lattice. Also any set with one element is often regarded as a complete lattice.

<u>Lists</u>

DEFINITION 2.1.3-8: Define $L^*=\{\bot,\top\}\cup\{<l1,...,lm> \mid m\geq 0 \wedge l1\in L1 \wedge ... \wedge lm\in Lm\}$ and $l\sqsubseteq l'$ <=> $(l=\bot \vee l'=\top \vee [\exists m: \exists l1,...,lm: \exists l'1,...,l'm: l=<l1,...,lm> \wedge l'=<l'1,...,l'm> \wedge l1\sqsubseteq l'1 \wedge ... \wedge lm\sqsubseteq l'm])$.

For i > 0 define $\downarrow i$:L^*->L to be the doubly strict function satisfying $<l1,...,ln>\downarrow i$ = $(i\leq n$ -> $li,\top)$ [MiS76]. Define $\downarrow 0=\downarrow\top$ to be the doubly strict function satisfying $<l1,...,ln>\downarrow 0$ = $<l1,...,ln>\downarrow\top = \top$. Define $\downarrow\bot = \lambda l.\bot$.

Define #:L^*->N to be the doubly strict function satisfying #<l1,...,lm> = m.

For i>0 define $\dagger i$:L^*->L^* to be the doubly strict function satisfying $<l1,...,lj>\dagger i$ = $(i\leq j$ -> $<l[i+1],...,lj>$, <>) [MiS76].

Define §:$L^* \times L^*$->L^* to be the very strict function satisfying <l1,...,ln> § <l[n+1],...,lm> = <l1,...,lm>. []

LEMMA 2.1.3-9: L^* is a complete lattice. Also $\downarrow i$, # and $\dagger i$ are continuous. Function $\lambda l^*.l^*\downarrow\# l^*$ is continuous and $\lambda<l^*,l^{*\prime}>.l^*$§$l^{*\prime}$ is continuous. []

<u>Powersets</u>

DEFINITION 2.1.3-10: $(\mathcal{P}(L),\subseteq)$ is the power-set of L with $\subseteq$ as ordering.
Similarly, $(\mathcal{P}'(L),\supseteq)$ is the power-set of L with $\supseteq$ as ordering. []

OBSERVATION 2.1.3-11: $\mathcal{P}$(L) and $\mathcal{P}'$(L) are complete lattices and $\mathcal{P}$(L) has $\sqcup$ to be $\cup$ whereas $\mathcal{P}'$(L) has $\sqcup$ to be $\cap$. []

There are other possible orderings on the powerset of L. One is the Egli-Milner ordering: $\underline{l1}\sqsubseteq\underline{l2}$ <=> $(\forall l1\in\underline{l1}\ \exists l2\in\underline{l2}: l1\sqsubseteq l2) \wedge (\forall l2\in\underline{l2}\ \exists l1\in\underline{l1}: l1\sqsubseteq l2)$. This ordering depends on the partial order of L, contrary to $\subseteq$ and $\supseteq$. For our purposes, however, orderings $\subseteq$ and $\supseteq$ are more adequate. We also discuss powersets in section 3.1 and chapter 7.

Functions

DEFINITION 2.1.3-12: Define
L-t>M = {f | f:L->M }   (t for total)
L-i>M = {f | f:L->M and f isotone }
L-c>M = {f | f:L->M and f continuous }
and $f \sqsubseteq f'$ <=> $\forall l \in L$: $f(l) \sqsubseteq f'(l)$ []

LEMMA 2.1.3-13:
1) L-t>M is a complete lattice with $\sqcup \underline{f} = \lambda l . \sqcup \{ f(l) \mid f \in \underline{f} \}$ and $\sqcap \underline{f} = \lambda l . \sqcap \{ f(l) \mid f \in \underline{f} \}$ (where $\underline{f} \subseteq$ L-t>M).

2) L-i>M is a complete lattice with $\sqcup$ and $\sqcap$ as above.

3) L-c>M is a complete lattice with $\sqcup$ as above and $\sqcap \underline{f} = \sqcup \{ g \mid g \in$ L-c>M $\wedge \forall f \in \underline{f}: f \sqsupseteq g \}$. This $\sqcap$ may differ from the above. []
Proof is shown in appendix 2. []

We sometimes write S-t>M where S is a set. Clearly S-t>M is also a complete lattice.

In domain definitions -t>, -i> and -c> associate to the right and have lower precedence than + and X.

2.2 Denotational Semantics

In this section we introduce a toy language and explain its denotational semantics. We will later perform data flow analysis on this toy language.

The abstract syntax of our toy language is shown in table 2.2-A. The language consists of programs, declarations, commands, expressions, identifiers, basic values, and operators. The commands include input/output, a conditional, and a while-loop. We have not included nested declarations, procedures, jumps, and escapes. This is motivated in section 3.1.

We shall view a program in two ways. In Denotational Semantics one usually perceives a program as an (abstract) parse tree [Sto77,p.148]. Another approach is to perceive a program as a labelled parse tree. The root is labelled by the empty tuple (<>) and if some node is the j'th son of a node labelled occ then it is labelled occ§<j>. Following [Don79] these labels are called occurrences. If occ is an occurrence labelling some node in the program pro then pro <u>at</u> occ denotes the labelled subtree with that node as its root. - It is not necessary to be more formal with these definitions (until we consider program transformations).

The syntactic categories Pro, Dec, Cmd, Exp, Bas, Ide and Ope are sets, not flat lattices. This is advantageous later.

The semantics of our toy language has been separated into three tables (2.2-B, 2.2-C and 2.2-D). We first give an overview of the tables, then explain them in detail.

Table 2.2-C defines the semantic functions $\mathcal{P}$, $\mathcal{D}$, $\mathcal{C}$ and $\mathcal{E}$ that give meaning to programs, declarations, commands, and expressions (respectively). Some domains, constants and functions are not defined in Table 2.2-C. They are

TABLE 2.2-A --- ABSTRACT SYNTAX

Syntactic Categories

```
pro ∈ Pro    programs
dcl ∈ Dcl    declarations
cmd ∈ Cmd    commands
exp ∈ Exp    expressions
ide ∈ Ide    identifiers
bas ∈ Bas    (representations
             of) basic values
ope ∈ Ope    operators
```

Syntax

```
pro ::= BEG dcl IN cmd END

dcl ::= dcl1 ; dcl2
      | DCL ide := bas

cmd ::= cmd1 ; cmd2
      | ide := exp
      | IF exp THEN cmd1 ELSE cmd2 FI
      | WHILE exp DO cmd OD
      | WRITE exp
      | READ ide

exp ::= exp1 ope exp2
      | ide
      | bas

ide ::= A | B | ...

bas ::= TRUE | FALSE | 0 | 1 | ...

ope ::= + | X | ...
```

TABLE 2.2-B --- COMMONLY USED SEMANTIC DOMAINS

```
sta ∈ Sta = EnvXInpXOutXWit      states
env ∈ Env = Ide° -c> Val         environments
inp ∈ Inp = Val*                 inputs
out ∈ Out = Val*                 outputs
wit ∈ Wit = Val*                 stacks of witnessed values
val ∈ Val = N + T +{"nil"}       values
occ ∈ Occ = N*                   occurrences
pla ∈ Pla = N*XQ                 places
  n ∈ N                          (see  Ex. 2.1.1-6)
  t ∈ T                          (see  Ex. 2.1.1-6)
  q ∈ Q                          (see  Ex. 2.1.1-6)
```

defined in Table 2.2-D or Table 2.2-B, thus completing the semantics. The purpose of Table 2.2-B is to contain some domain definitions that will be used throughout this paper.

That Table 2.2-D is separated from Table 2.2-C is advantageous because we give several semantics for the toy language; many of these can be defined by supplying a substitute for Table 2.2-D. The phrase "substitute for Table 2.2-D" can be made more precise by requiring the substitute to be an "interpretation":

DEFINITION 2.2-1: An <u>interpretation</u> is a table defining

a) complete lattices C, I, A and S for continuations, inputs, answers and states (respectively).
b) constants wrong∈C and finish∈C

TABLE 2.2-C --- SEMANTIC FUNCTIONS

Semantic Equations

```
𝒫 ∈ Pro -t> I -t> A

  𝒫⟦ BEG dcl IN cmd END ⟧ =
    setup( 𝒟⟦dcl⟧<1>;
           𝒞⟦cmd⟧<2>;
           finish)

𝒟 ∈ Dcl -t> Occ -c> C -c> C

  𝒟⟦ dcl1 ; dcl2 ⟧ occ c =
    attach<occ,"(dcl">;
    𝒟⟦dcl1⟧ occ§<1>;
    𝒟⟦dcl2⟧ occ§<2>;
    attach<occ,"dcl)">;c

  𝒟⟦ DCL ide := bas ⟧ occ c =
    attach<occ,"(dcl">;
    push⟦bas⟧;
    assign⟦ide⟧;
    attach<occ,"dcl)">;c

ℰ ∈ Exp -t> Occ -c> C -c> C

  ℰ⟦ exp1 ope exp2 ⟧ occ c =
    attach<occ,"(exp">;
    ℰ⟦exp1⟧ occ§<1>;
    ℰ⟦exp2⟧ occ§<3>;
    apply⟦ope⟧;
    attach<occ,"exp)">;c

  ℰ⟦ ide ⟧ occ c =
    attach<occ,"(exp">;
    content⟦ide⟧;
    attach<occ,"exp)">;c

  ℰ⟦ bas ⟧ occ c =
    attach<occ,"(exp">;
    push⟦bas⟧;
    attach<occ,"exp)">;c

𝒞 ∈ Cmd -t> Occ -c> C -c> C

  𝒞⟦ cmd1 ; cmd2 ⟧ occ c =
   attach<occ,"(cmd">;
   𝒞⟦cmd1⟧ occ§<1>;
   𝒞⟦cmd2⟧ occ§<2>;
   attach<occ,"cmd)">;c

  𝒞⟦ ide := exp ⟧ occ c =
   attach<occ,"(cmd">;
   ℰ⟦exp⟧ occ§<2>;
   assign⟦ide⟧;
   attach<occ,"cmd)">;c

  𝒞⟦ IF exp THEN cmd1 ELSE cmd2 FI⟧occ c=
   attach<occ,"(cmd">;
   ℰ⟦exp⟧ occ§<1>;
   cond( 𝒞⟦cmd1⟧ occ§<2>;
           attach<occ,"cmd)">;c
         , 𝒞⟦cmd2⟧ occ§<3>;
           attach<occ,"cmd)">;c )

  𝒞⟦ WHILE exp DO cmd OD ⟧ occ c =
   attach<occ,"(cmd">;
   FIX(λc'. ℰ⟦exp⟧ occ§<1>;
            cond( 𝒞⟦cmd⟧ occ§<2>;c'
                , attach<occ,"cmd)">;c))

  𝒞⟦ WRITE exp ⟧ occ c =
   attach<occ,"(cmd">;
   ℰ⟦exp⟧ occ§<1>;
   write;
   attach<occ,"cmd)">;c

  𝒞⟦ READ ide ⟧ occ c =
   attach<occ,"(cmd">;
   read;
   assign⟦ide⟧;
   attach<occ,"cmd)">;c
```

```
c) auxiliary functions: setup  ∈ C -c> I -t> A
                        cond   ∈ C X C -c> C
                        attach ∈ Pla -c> C -c> C
d) primitive functions
   apply   ∈  Ope -t> C -c> C          assign ∈ Ide -t> C -c> C
   content ∈  Ide -t> C -c> C          push   ∈ Bas -t> C -c> C
   read    ∈  C -c> C                  write  ∈ C -c> C           []
```

Obviously, Definition 2.2-1 is closly connected with Table 2.2-C. We write Ope -t> C -c> C because we consider Ope as a set, whereas in Denotational Semantics it is often considered as a flat lattice (Ope°). Similar remarks apply to Ide and Bas. If int is the name of an interpretation we write ℰint, int-apply, int-C, etc. to precisely describe which "version" of ℰ, apply, C etc. that we mean.

```
TABLE 2.2-D(a) --- INTERPRETATION std
=======================================

Domains (of std)
----------------
  c ∈ C = S -c> A
  i ∈ I = Inp
  a ∈ A = Out + {"wrong"}
  s ∈ S = Sta

Constants (of std)
------------------
  wrong ∈ C             error continuation
  wrong = λs."wrong" inA

  finish ∈ C
  finish = λ<env,inp,out,wit>. out inA

Pseudo-Semantic Functions (of std)
----------------------------------
  ℬ ∈ Bas° -c> Val
  𝒪 ∈ Ope° -c> (Val X Val -c> Val)

Auxiliary Functions (of std)
----------------------------
  setup ∈ C -c> I -c> A
  setup c i =
    c < λide."nil" inVal, i, <>, <> >

  cond ∈ C X C -c> C                  (conditional)
    Vcond ∈ S -c> T                   ("verify")
    Scond ∈ S -c> T                   ("select")
    Bcond ∈ S -c> S                   ("body")

  cond(c1,c2) s =
    Vcond(s) -> (Scond(s) -> c1, c2) (Bcond(s)),
                 wrong s
  Vcond< env, inp, out, wit > =
    (#wit<<1) -> false, wit↓1ET
  Scond< env, inp, out, <t inVal>§wit > =
    t
  Bcond< env, inp, out, <t inVal>§wit > =
    <env, inp, out, wit>

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

The semantics of our toy language can be characterized as a store semantics [Sto77][MiS76] in continuation style that employs a one-stage mapping between identifiers and values.

We use continuation style rather than direct style, although the toy language can be defined without using continuations. The use of continuations is useful in chapter 4; also [Gor79,p.52] argues in favour of using continuations. It is further discussed in chapter 7.

The association between identifiers and values is by means of a one-stage mapping, i.e. we do not use locations. This tends to simplify the later development, but is otherwise unimportant. In fact the draft of this paper employed a two-stage mapping.

The semantics is not a standard semantics [Sto77][MiS76] but much closer to a store semantics [MiS76]. Why this is desirable cannot be satisfactorily explained until chapter 3 (Theorem 3.1-6).

The essential difference between a store semantics (without locations) and a standard semantics (without locations) is that in the former there is a Wit component in the domain Sta of states (Table 2.2-B). This domain is called "stacks of witnessed values", and the purpose of wit∈Wit=Val* is to hold the partial results arising during evaluation of an expression: Each 𝒢⟦exp⟧ pushes one value upon the stack of witnessed values before supplying the new state to the argument continuation.

```
TABLE 2.2-D(b) --- INTERPRETATION std
======================================

Primitive Functions (of std)
----------------------------
  g ∈ Par -t> C -c> C
    Vg ∈ Par -t> S -c> T          ("verify")
    Bg ∈ Par -t> S -c> S          ("body")

  g(par) c s =
    Vg(par) s -> c( Bg(par) s ), wrong s

  Vapply⟦ope⟧<env, inp, out, wit> =
    (#wit<<2) -> false, true
  Bapply⟦ope⟧<env, inp, out, <val1,val2>§wit > =
    <env, inp, out, <𝒪⟦ope⟧<val2,val1>§wit >

  Vassign⟦ide⟧<env, inp, out, wit> =
    (#wit<<1) -> false, true
  Bassign⟦ide⟧<env, inp, out, <val>§wit> =
    <env[val/ide], inp, out, wit>

  Vcontent⟦ide⟧<env, inp, out, wit> =
    true
  Bcontent⟦ide⟧<env, inp, out, wit> =
    <env, inp, out, <env⟦ide⟧>§wit>

  Vpush⟦bas⟧<env, inp, out, wit> =
    true
  Bpush⟦bas⟧<env, inp, out, wit> =
    <env, inp, out, <ℬ⟦bas⟧>§wit>

  Vread<env, inp, out, wit> =
    (#inp<<1) -> false, true
  Bread<env, <val>§inp, out, wit> =
    <env, inp, out, <val>§wit>

  Vwrite<env, inp, out, wit> =
    (#wit<<1) -> false, true
  Bwrite<env, inp, out, <val>§wit> =
    <env, inp, out§<val>, wit>
```

Before explaining the tables in detail we explain the notation. It is influenced by [Mos79], e.g. in using ';' to denote functional application that associates to the right. It has the lowest presedence of all operators. The operator # has higher precedence than ';' but still lower than ↑i and ↓i.

We assume that ++, ∧∧, ∨∨, <<, ... are the very strict extensions of +, ∧, ∨, <, ... on the appropriate flat lattices. Clearly they are continuous. We assume that == is a continuous function L×L->T. Also

DEFINITION 2.2-2: Define the predicate pure[L]:L->B by

a) If L is flat: pure[L](l) = (l∉{⊥,⊤})

b) If L = L1×...×Ln: pure[L]<l1,...,ln> = ∀i∈{1,...,n}:pure[Li](li)

c) If L = L1+...+Ln: pure[L](⊤)=pure[L](⊥)=false
pure[L](li inL) = pure[Li](li)

d) If L = M*: pure[L](l) = (l∉{⊥,⊤} ∧ ∀i∈{1,...,#l}: pure[M](l↓i)) []

ASSUMPTION 2.2-3: pure[L](l1) $\wedge$ pure[L](l2) => (l1 == l2) = (l1=l2) []

Whenever t$\in$T and l1,l2$\in$L by t->l1,l2 is meant l1 if t=true, l2 if t=false, $\perp$ if t=$\perp$ and $\top$ if t=$\top$. We define f[y/x] to mean λz.z==x->y,f(z). One can show that ...[.../...] and ...->...,... are continuous.

Now consider Table 2.2-C. We write $\mathcal{C}$ $\in$ Cmd -t> Occ -c> C -c> C rather than $\mathcal{C}$ $\in$ Cmd$^{\circ}$ -c> Occ -c> C -c> C. Functions $\mathcal{C}$, $\mathcal{E}$ and $\mathcal{D}$ take an occurrence as a parameter: In $\mathcal{E}$⟦exp⟧occ the occurrence occ is intended to be the label of the root of exp, when exp is viewed as a labelled subtree of a program pro. This use of occurrences corresponds to the positions of [Gor79].

The main use of occurrences is to be a component of a place pla$\in$Pla=Occ$\times$Q, which is passed as an argument to the function attach. The intention of <occ,"exp)"> is to say that pro <u>at</u> occ is an expression, and that the corresponding attach function was placed "last" in the sequence of functions. In chapters 3 and 4 attach<occ,"exp)"> plays an important role; in Table 2.2-D it is merely the identity.

Consider Table 2.2-D defining interpretation std (for "standard") and Table 2.2-B. We have placed output in the state. Since FIX(g)=$\perp$<=> g$\perp$=$\perp$ this implies that a looping program does not produce any output. According to [MiS76,p.218] this is of little importance in practice. Finally, we have kept the error handling at a minimum in order to keep the semantics small.

The primitive functions of Table 2.2-D are defined according to the pattern:

g(par)(c)(sta) = Vg(par)(sta)->c(Bg(par)(sta)),(wrong sta)

where Vg $\in$ Par -t> Sta -c> T and Bg $\in$ Par -t> Sta -c> Sta. Here Par is a set of "parameters", that may or may not be present (depending on which g it is). The purpose of Vg (for "verify") is to verify that the state is in a special format. Only then Bg (for "body") is applied to the state. It aids the readability to write the state according to the format (as in [Mos79,p.12]). It should be clear how one can avoid this syntactically sugared notation: e.g. in the case of Bassign⟦ide⟧:

Bassign⟦ide⟧<env,inp,out,wit> = <env[wit↓1/ide],inp,out,wit↑1>

The advantage of expressing primitive functions in a common form is to make the interpretations easier to read and to simplify proofs.

The parameters (in Par), that some primitive functions have, are syntactic objects corresponding to leaves of the parse-tree. In Table 2.2-D there are some unspecified "pseudo"-semantic functions ($\mathcal{B}$ and $\mathcal{O}$) mapping the syntactic objects into semantic objects.

Tables 2.2-B, 2.2-C and 2.2-D (essentially) constitute a denotational definition in the style of [Sto77]. Therefore the mathematical foundations of [Sto77] ensure that $\mathcal{P}$, $\mathcal{D}$, $\mathcal{E}$ and $\mathcal{C}$ are well-defined. In later interpretations we introduce powersets. As it is unclear how to define a retract (in LAMBDA [Sto77]) for powersets we explicitly provide an alternate mathematical foundation:

THEOREM 2.2-4: When int is an interpretation then Table 2.2-C and int define functions $\mathcal{P}$int, $\mathcal{D}$int, $\mathcal{C}$int and $\mathcal{E}$int. They are of functionalities as shown in Table 2.2-C and no function is supplied with an argument of the wrong type. When setup $\in$ C -c> I -i> A then $\mathcal{P}$int $\in$ Pro -t> I -i> A, and when setup $\in$ C -c> I -c> A then $\mathcal{P}$int $\in$ Pro -t> I -c> A. []

Proof is shown in appendix 2. []

In LAMBDA there is no danger of supplying an argument of the wrong type because retracts do a coercion, if necessary, so that even supplying an argument of the wrong type is well-defined. To avoid these issues we explicitly stated that the situation does not arise. Because of our alternate mathematical foundation we must show:

LEMMA 2.2-5: Table 2.2-D specifies an interpretation. []
Proof is shown in appendix 2. []

One assumption fulfilled by Tables 2.2-B, 2.2-C, 2.2-D and later interpretations is:

ASSUMPTION 2.2-6: All domains are complete lattices, and we take fixed points only of continuous functions. []

This is weaker than the usual requirements of Denotational Semantics: For simplicity we do not assume that domains are countably based and continuous [Sto77]; so for us "domain" simply means "complete lattice". Also we allow non-continuous functions because they are needed later (normally all functions must be continuous).

2.3 Data Flow Analysis

In this section we review some of the concepts from data flow analysis. In the literature the details vary, but our treatment agrees with the intentions of most authors.

In traditional data flow analysis a program is viewed as a graph ("flowchart"). The nodes (called basic blocks) contain very simple instructions, e.g. a sequence of assignment statements where the expressions contain only one operator. The instructions in a node are executed in the order of appearance. The possibility of error stops is often ignored. Arcs represent flow of control between nodes. Obviously, programs in our toy language must be transformed before they conform to this view.

In the process of data flow analysis some information is associated with arcs, entries to nodes, or exits from nodes. We prefer to talk in terms of arcs. *Global analysis* associates information with arcs. To reduce the space required to represent this information one usually makes nodes as large as possible. *Local analysis* propagates information inside nodes. It is usually quite simple and is often ignored when a data flow analysis is discussed.

We now review three data flow analysis problems.

Constant propagation is a data flow analysis problem where "constant computations are evaluated at compiletime"[Kil73]. The global analysis consists in associating a pool of tuples with arcs. The first component of a tuple is an identifier and the second component is its constant value (at this arc). Formally, a pool is an element of

$$\{\ pool \subseteq Ide \times Val \mid (\langle ide1,val1\rangle \in pool \wedge \langle ide2,val2\rangle \in pool \wedge ide1=ide2) \Rightarrow val1=val2\ \}$$

Here Ide and Val are sets of identifiers and values (respectively). So-called *transfer functions* [AhU78,p.497] describe how a pool is transformed when passing through a node.

Data flow information computed by constant propagation is often used for the program transformation "constant folding", which is the "replacement of run-time computations by compile-time computations"[AhU78].

Available Expressions is a "history-sensitive" data flow analysis problem (in the terminology of [CoC79,p.278]). We want to determine for each computation of some expression exp whether it "has previously been computed and has suffered no subsequent change in value, where an expression is considered to have changed in value whenever any of the [identifiers] involved in it has changed" [Ros79].

In global analysis one associates with each arc a pool of available expressions, i.e. expressions that have previously been computed and have suffered no subsequent change in value. In this description we have tacitly assumed that sharing does not occur (in accordance with most of the literature). The transfer function is of the form λ*exp*.(*exp* ∩ preserved) ∪ generated, where *exp*, preserved and generated are subsets of Exp.

Live variables is another "history-sensitive" data flow analysis. Here "we wish to know for [identifier] A and [arc] p whether the value of A at p could be used along some path in the ... graph starting at p. If so, we say that A is live at p; otherwise A is dead at p"[AhU78,p.489]. In global analysis one associates a pool of live identifiers to each arc.

We now consider solutions to data flow analysis problems.

Consider a graph representing a program. We assume that the graph contains a unique entry arc (start) and a unique exit arc (stop). We say that <arc1,...,arcn> is a path iff n≥1 and for every i∈{1,...,n-1} that arc[i+1] leaves the node that arci enters. We assume that there is no path of the form <arc,start> or <stop,arc>.

Let L be any complete lattice of data flow information, and let init ∈ L. (Many authors have init=⊥ and slightly different requirements upon L). For each path of the form <arc1,arc2> we are given a forward transfer function Bf<arc1,arc2> ∈ L-t>L or a backward transfer function Bb<arc1,arc2> ∈ L-t>L. The forward transfer function describes the effect of "going" from arc1 to arc2. The backward transfer function specifies the effect of "going" from arc2 to arc1. When <arc1,arc2> and <arc3,arc4> are paths such that arc1 and arc3 enter the same node it is natural to assume that Bf<arc1,arc2> = Bf<arc3,arc4> and similarly for Bb. For convenience we define Bf<arc> = Bb<arc> = λl.l, and when n>2: Bf<arc1,...,arcn> = Bf<arc2,...,arcn> ∘ Bf<arc1,arc2> and Bb<arc1,...,arcn> = Bb<arc1,...,arc[n-1]> ∘ Bb<arc[n-1],arcn>.

Usually one considers a MOP (meet over all paths) solution and a MFP (maximal fixed point) solution. These terms have been coined by authors using lattices dual to ours; as discussed in section 2.1.2 this is of little importance. We define the *forward MOP solution* (to the data flow analysis defined by Bf):

λarc.⊔{ Bf<start,...,arc>init | <start,...,arc> is a path}

This formulation essentially is that of [Hec77,p.169]. Similarly, we can define the backward MOP solution (defined by Bb).

To define the *forward MFP solution* we assume Bf<arc1,arc2> ∈ L-i>L. We define a "system of equations" F ∈ (Arc-t>L) -i> (Arc-t>L) by

F(inf)(arc) = (arc=start) -> init, ⊔{ Bf<pred,arc>(inf(pred)) |
<pred,arc> is a path }

where inf ∈ Arc-t>L. This may be compared to the formulation of [Hec77,p.173] (Kildall's MFP solution) or preferably [Hec77,p.178] (Kam&Ullman's MFP solution). The forward MFP solution is LFP(F) which exists by Theorem 2.1.2-11. Similarly we can define the backward MFP solution.

When Bf<arc1,arc2> (and Bb<arc1,arc2>) are continuous there are iterative methods converging to the MFP solution. When Bf<arc1,arc2> (and Bb<arc1,arc2>) are complete-$\sqcup$-morphisms the MFP solution is equal to the MOP solution, otherwise the MOP solution is better ($\sqsubseteq$) than the MFP solution [CoC79]. One way to obtain the MOP solution (for finite L and Bf<arc1,arc2> and Bb<arc1,arc2> isotone) is the "merge after expansion" algorithm of [Weg75].

The forward solutions are used for constant propagation and available expressions ("forward analyses"), whereas the backward solutions are used for live variables ("backward analysis"). It is generally assumed that the MOP solution is wanted, but the arguments are intuitive: "It appears generally true that for data flow analysis problem, we search for the 'meet over all paths' solution. Intuitively, this solution is the calculation for each node in the program ... graph of the maximum information, relevant to the problem at hand, which can be derived from every possible execution path from the initial node to that node"[Hec77,p.169].

We distinguish between two ways of defining the lattice L of data flow information: the "independent attribute method" and the "relational method"[JoM78]. In order to explain the two methods imagine that there are n identifiers and a set M whose elements describe properties of the values of one identifier. For the explanation assume that M is Val, the set of values.

In the *independent attribute method* we have $L=(\mathcal{P}(M))^n$. If $l \in L$ is associated with some arc, then the description of the set of possible values of the j'th identifier is $l \downarrow j$. Most methods in literature and in practice are of this type.

In the *relational method* we have $L=\mathcal{P}(M^n)$. If $l \in L$ is associated with some arc, then the description of the set of possible values for the j'th identifier is $\{v \downarrow j \mid v \in l\}$. In general the relational method is stronger than the independent attribute method, e.g. in our example when determining whether two identifiers have the same value at some arc. But the relational method is potentially of much higher computational complexity than the independent attribute method[JoM78]. In e.g. the determination of linear relationships ([CoH78]) it appears mandatory to work from a "kind of" relational method.

2.4 Abstract Interpretation

In this section we explain abstract interpretation. A detailed development of the central concepts is given in subsection 2.4.3. This development extends that of [CoC79] in two ways: We hope our motivation of the definitions ("pair of adjoined functions") gives more insight. Furthermore, we introduce concepts (e.g. "pair of semi-adjoined functions") that are less restrictive than those of [CoC79]. A substantial part of our further development will be performed using these definitions.

The overall motivation underlying abstract interpretation is reviewed in subsection 2.4.1. We also state the key definitions and results of subsection 2.4.3, so that 2.4.3 perhaps can be omitted on a first reading. Some example pairs of adjoined functions are defined in subsection 2.4.2; these are used in section 3.2.

2.4.1 The Overall Motivation

We assume, as in section 2.3, that a program is represented by a graph. By Arc we denote the (finite) set of arcs and by L we denote a complete lattice of data flow information. For the purposes of this explanation it is convenient to think of L as $\mathcal{P}$(Sta), where Sta is the set of program states. Since backward analysis is treated in a way similar to forward analysis [CoC79] we will only consider forward analysis. Let init∈L be the set of initial states and let Bf<arc1,arc2> describe the effect upon a set of states as a node is traversed. In this way Bf<arc1,arc2> relates to the static semantics of Floyd (see [CoC77a]). By inf∈Arc-t>L we denote a solution to the data flow analysis specified by Bf. It is convenient to think of inf as the MOP solution so that inf(arc) is the set of states that can occur at arc.

One use of inf is to detect the applicability of some program transformation. Many program transformations (e.g. constant folding) can be characterized by some predicate p:Sta->B, so that the transformation may be performed upon some node if: for any one arc arc' entering that node and for any one sta∈inf(arc') that p(sta) holds. Define lp={sta|p(sta)}; then this can be expressed by ⊔{inf(arc')| arc' enters the node} ⊆ lp.

Presumably, we want to implement the data flow analysis on a computer. Maybe the solution is not computable, or it takes too much time to compute it, or it takes too much space to store the solution. The remedy we consider is to replace L by another complete lattice M of approximate description elements of L for which the above problems maybe do not arise. We do not consider the widening and narrowing of [CoC77a].

EXAMPLE 2.4.1-1: Imagine a program with one identifier taking values in the set INT = {..., -1,0,1, ...} of integers. Then L=$\mathcal{P}$(INT) and init∈L is the set of initial (or input) values. Function Bf<arc1,arc2>:L->L describes the effect upon the set of possible values when going from arc1 to arc2.

A possible predicate is p(i) = (i=27) from which lp={27}. Let inf be the (MOP) solution specifying for each arc the set of values reaching that arc (for some initial value in init). If arc1, ..., arcm are the arcs entering some node n, and if ⊔{inf(arci) | i∈{1,...,m}} ⊆ lp then we may hope to perform constant folding on the node n, e.g. replace some use of the identifier with the constant 27.

A possible choice of M^o is the flat lattice INT . The intention is that e.g. 7∈M corresponds to {7}∈L, while ⊤∈M corresponds to any set in L containing two or more constants. This is further developed below. []

The data flow analysis problem (as specified in terms of L, init, Bf) is transformed to another data flow analysis problem (specified in terms of M, init', Bf'). Denote by inf:Arc->L and inf':Arc->M the respective solutions. We now consider the connection between the two formulations.

We want to relate M to L in order to make use of the data flow information inf'(arc). We transform inf'(arc) to L rather than inf(arc) to M, because it is in L we have lp and perform the tests ...⊆lp {a}; intuitively speaking, we know how to interpret l∈L but not how to interpret m∈M. To this end we define a <u>concretization function</u> conc:M->L (denoted γ in [CoC79]). Then inf'(arc) is a representation in M of conc(inf'(arc))∈L. It is reasonable to

---

{a} In subsection 2.4.3 we show how to transform the test ...⊆lp to a test in M.

require that conc(inf'(arc)) $\sqsubseteq$ lp => inf(arc) $\sqsubseteq$ lp, because otherwise it would be hard to see how inf'(arc) could be exploited. Clearly conc(inf'(arc))$\sqsubseteq$lp => inf(arc) $\sqsubseteq$ lp holds for any lp iff inf(arc) $\sqsubseteq$ conc(inf'(arc)). We take this as the desired correctness condition between inf and inf'. That is, we approximate a set of states by (a description of) a larger set of states. - This may be compared to the "make errors on the conservative side" of [AhU78,p.504].

It is convenient also to define an abstraction function uabs:L->M (denoted α in [CoC79]). It is used to approximate members of L by members of M, e.g. init' = uabs(init). The condition inf(arc) $\sqsubseteq$ conc(inf'(arc)) for arc=start then amounts to init $\sqsubseteq$ conc(uabs(init)). This motivates:

ASSUMPTION 2.4.1-2: conc$\circ$uabs is extensive. []

Following [CoC79] we impose the intuitively desirable:

ASSUMPTION 2.4.1-3: conc and uabs are isotone. []

Intuitively, that uabs is isotone means that it "preserves the amount of information": If we get more information in L then we also get more information in M. Similar remarks apply to conc. Technically, in some proofs (e.g. Theorem 3.1-10) our reasoning exploits the isotony of conc. We do not investigate whether assumption 2.4.1-3 can be weakened.

The above assumptions make it reasonable to request <uabs,conc> to be a pair of semi-adjoined functions between L and M:

DEFINITION 2.4.1-4: <uabs,conc> is a pair of semi-adjoined functions (between L and M) iff
a) uabs $\in$ L -i> M
b) conc $\in$ M -i> L
c) conc$\circ$uabs is extensive []

Often we write: "uabs and conc are semi-adjoined" instead of "<uabs,conc> is a pair of semi-adjoined functions".

OBSERVATION 2.4.1-5: uabs and conc are semi-adjoined iff they are isotone and $\forall l \in L, m \in M$: uabs(l)$\sqsubseteq$ m => l $\sqsubseteq$ conc(m). []

If conc is isotone and conc($\top$)=$\top$ then there always exists a uabs such that uabs and conc are semi-adjoined, e.g. uabs=$\lambda l.\top$.

Sometimes it is desirable that uabs and conc satisfy stronger properties than merely semi-adjoined.

DEFINITION 2.4.1-6: <uabs,conc> is a pair of adjoined functions (between L and M) iff $\forall l \in L\ \forall m \in M$: uabs(l) $\sqsubseteq$ m <=> l $\sqsubseteq$ conc(m). []

In subsection 2.4.3 we motivate this definition and we show that any pair of adjoined functions is also a pair of semi-adjoined functions. One nice property of a pair <uabs,conc> of adjoined functions is that one determines the other, e.g. uabs = $\lambda l. \sqcap \{m|$ conc(m)$\sqsupseteq l\}$ (Lemma 2.4.3-3). Often we write "uabs and conc are adjoined" instead of "<uabs,conc> is a pair of adjoined functions".

DEFINITION 2.4.1-7: A pair <uabs,conc> of adjoined functions is <u>exact</u> iff uabs$\circ$conc = λm.m []

In [CoC79] it is stated that <uabs,conc> is exact iff uabs is onto iff conc is one-one.

EXAMPLE 2.4.1-1(cont.): Define conc:INT$^\circ$->$\mathcal{P}$(INT) by conc($\perp$)=$\emptyset$, conc($\top$)=INT and conc(i)={i} otherwise. Define uabs:$\mathcal{P}$(INT)->INT$^\circ$ by uabs($\emptyset$)=$\perp$, uabs({i})=i and uabs($\underline{i}$)=$\top$ whenever $\underline{i}$ contains two or more elements. Then <uabs,conc> is an exact pair of adjoined functions. []

An approximate data flow propagation function Bf' may be defined by Bf'<arc> = λm.m, Bf'<arc1,arc2> = uabs $\circ$ Bf<arc1,arc2> $\circ$ conc and when m>2: Bf'<arc1, ...,arcm> = Bf'<arc[m-1],arcm> $\circ$ ... $\circ$ Bf'<arc1,arc2>; Bf' is <u>induced</u> by <uabs,conc> from Bf. Suppose init' = uabs(init), that inf and inf' are the MOP solutions and that uabs and conc are adjoined. Then [CoC79,7.1.0.2(2)] implies (for a special choice of Bf) that $\forall$arc: inf(arc) $\sqsubseteq$ conc(inf'(arc)). One can show that $\forall$arc: inf(arc) $\sqsubseteq$ conc(inf'(arc)) holds when uabs and conc are only semi-adjoined provided that $\forall$arc1,$\forall$arc2: Bf<arc1,arc2> is isotone (and init' = uabs(init)).

2.4.2 Defining Example Pairs of Adjoined Functions

In chapter 3 we develop and apply our framework for expressing data flow analyses. For the application we need to define a pair of semi-adjoined functions. Often it is convenient to define a pair <uabs,conc> of semi-adjoined functions by composing others (<uabsi,conci>):

LEMMA 2.4.2-1: If <uabsi,conci> (for i$\in${1,....n}) is a pair of semi-adjoined (adjoined) functions between Li and L[i+1] then <uabsn $\circ$ ... $\circ$uabs1, conc1 $\circ$ ... $\circ$ concn> is a pair of semi-adjoined (adjoined) functions between L1 and L[n+1]. []
Proof is shown in appendix 2. []

In each application one may use pairs <uabsi,conci> that are generally applicable. Typically, <uabsi,conci> depends on the structure of a domain (like those of Table 2.2-B) defined in terms of the usual domain constructors +, X, -c> and *. Some of the pairs are best understood as transforming information between a relational formulation and an independent attribute formulation. Below we define such "generally applicable" pairs of semi-adjoined functions. We assume that L, Li and M are complete lattices.

<u>Cartesian Product</u>

To transform between independent attribute formulation and relational formulation:

DEFINITION 2.4.2-2: Define concX:$\mathcal{P}$(L1)X...X$\mathcal{P}$(Ln) -> $\mathcal{P}$(L1X...XLn) by concX<$\underline{l}$1, ... ,$\underline{l}$n> = {<l1,...,ln> | $\forall$i$\in${1,...,n}: li$\in\underline{l}$i}.
Define uabsX($\underline{l}$) = <..., {l$\downarrow$i| l$\in\underline{l}$}, ...>. []

LEMMA 2.4.2-3: <uabsX,concX> is a pair of adjoined functions. []
Proof is shown in appendix 2. []

To assist in building pairs of semi-adjoined functions from others:

DEFINITION 2.4.2-4: If f is a function Li->Mi then (Ci f):L1X ...XLn -> L1X... XMiX ...XLn is defined by Ci f <l1,...,ln> = <l1,..., f(li), ...,ln>

LEMMA 2.4.2-5: If <uabs,conc> is a pair of adjoined functions (semi-adjoined functions) between Li and Mi then <Ci uabs, Ci conc> is a pair of adjoined functions (semi-adjoined functions) between L1X...XLnand L1X...XMiX...XLn[]
Proof is easy using <-> from the proof of 2.4.2-1. []

Finally we need:

DEFINITION 2.4.2-6:
Define uabs:no[i] :L1X...XLn -> L1X...XL[i-1]XL[i+1]X...XLn by
uabs:no[i]<l1,...,li,...,ln> = <l1,...,l[i-1],l[i+1],...,ln>
Define conc:no[i] :L1X...XL[i-1]XL[i+1]X...XLn -> L1X...XLn by
conc:no[i]<l1,...,l[i-1],l[i+1],...,ln> = <l1,...,l[i-1],$\top$,l[i+1],...,ln>[]

LEMMA 2.4.2-7: <uabs:no[i], conc:no[i]> is an exact pair of adjoined functions between L1X...XLiX...XLn and L1X...XL[i-1]XL[i+1]X...XLn. []

Separated Sum

DEFINITION 2.4.2-8: Define conc+: $\mathcal{P}$(L1)+...+$\mathcal{P}$(Ln) -> $\mathcal{P}$(L1+...+Ln) by

$$\mathrm{conc+}(m) = \begin{cases} \emptyset & \text{if } m=\bot \\ \{\, li \text{ inL1+...+Ln} \mid li \in \underline{l}i \,\} & \text{if } m=\underline{l}i \text{ in}\mathcal{P}(L1)+...+\mathcal{P}(Ln) \\ L1+...+Ln & \text{if } m=\top \end{cases}$$

Define uabs+ by

$$\mathrm{uabs+}(\underline{l}) = \begin{cases} \bot & \text{if } \underline{l}=\emptyset \\ \{\, li \mid (li \text{ inL1+...+Ln}) \in \underline{l} \,\} \text{ in}\mathcal{P}(L1)+...+\mathcal{P}(Ln) & \text{if } \exists i \forall l \in \underline{l}: \exists li \in Li: l=li \text{ inL1+...+Ln} \wedge \underline{l} \neq \emptyset \\ \top & \text{otherwise} \end{cases}$$ []

LEMMA 2.4.2-9: <uabs+,conc+> is a pair of adjoined functions. []
Proof is shown in appendix 2. []

Similarly to Ci we introduce Si:

DEFINITION 2.4.2-10: If f:Li->Mi then (Si f):L1+...+Ln -> L1+...+Mi+...+Ln is defined by

$$\mathrm{Si}\ f\ (l) = \begin{cases} \top & \text{if } l=\top \\ lj \text{ inL1+...+Mi+...+Ln} & \text{if } l=lj \text{ inL1+...+Ln} \wedge i \neq j \\ f(li) \text{ inL1+...+Mi+...+Ln} & \text{if } l=li \text{ inL1+...+Ln} \\ \bot & \text{if } l=\bot \end{cases}$$ []

LEMMA 2.4.2-11: If <uabs,conc> is a pair of semi-adjoined functions between Li and Mi then <Si uabs, Si conc> is a pair of semi-adjoined functions between L1+...+Ln and L1+...+Mi+...+Ln. If <uabs,conc> is a pair of adjoined functions then so is <Si uabs, Si conc>. []
Proof is shown in appendix 2. []

Lists

To transform between relational and independent attribute formulation:

DEFINITION 2.4.2-12: Define $conc^*: (\mathcal{P}(L))^* \to \mathcal{P}(L^*)$ by

$$conc^*(m) = \begin{cases} \emptyset & \text{if } m=\perp \\ \{\langle l1,\dots,ln\rangle \mid \forall i\in\{1,\dots,n\}: li \in m\downarrow i\} & \text{if } \# m \text{ is the integer } n \\ L^* & \text{if } m=\top \end{cases}$$

Define $uabs^*$ by

$$uabs^*(\underline{L}) = \begin{cases} \perp & \text{if } \underline{L}=\emptyset \\ \langle \dots, \{l\downarrow i \mid l\in \underline{L}\}, \dots\rangle & \text{if } \exists n \forall l\in\underline{L} \exists l1,\dots,ln: l=\langle l1,\dots,ln\rangle \\ & \text{and } \underline{L}\neq\emptyset \\ \top & \text{otherwise} \end{cases}$$ □

LEMMA 2.4.2-13: $\langle uabs^*, conc^*\rangle$ is a pair of adjoined functions. □
Proof is similar to the proof of 2.4.2-9. □

Similarly to Ci we introduce S^*:

DEFINITION 2.4.2-14: If f is a function L->M then $(S^*\ f): L^* \to M^*$ is defined by $S^*\ f\ \perp = \perp$, $S^*\ f\ \top = \top$ and $S^*\ f\ \langle l1,\dots,ln\rangle = \langle f(l1), \dots, f(ln)\rangle$ □

LEMMA 2.4.2-15: If $\langle uabs, conc\rangle$ is a pair of semi-adjoined functions between L and M then $\langle S^*\ uabs, S^*\ conc\rangle$ is a pair of semi-adjoined functions between L^* and M^*. If uabs and conc are adjoined then so are S^*uabs and S^*conc. □
Proof is similar to the proof of 2.4.2-11. □

Functions

To transform between relational and independent attribute formulation:

DEFINITION 2.4.2-16: Define $conc\text{-}c{>}: (L\text{-}t{>}\mathcal{P}(M)) \to \mathcal{P}(L\text{-}c{>}M)$ by $conc\text{-}c{>}(g) = \{f\in L\text{-}c{>}M \mid \forall l\in L: f(l)\in g(l)\}$ and $uabs\text{-}c{>}(\underline{f}) = \lambda l.\{f(l) \mid f\in\underline{f}\}$ □

LEMMA 2.4.2-17: $\langle uabs\text{-}c{>}, conc\text{-}c{>}\rangle$ is a pair of adjoined functions. □

Note that conc-c> maps from $L\text{-}t{>}\mathcal{P}(M)$ and not $L\text{-}c{>}\mathcal{P}(M)$. Similar definitions can be made for conc-i> and conc-t>.

DEFINITION 2.4.2-18: If $f: M1 \to M2$ then $(R\ f): (L\text{-}t{>}M1) \to (L\text{-}t{>}M2)$ is defined by $R\ f\ g = \lambda l.f(g(l))$, i.e. $R\ f\ g = f \circ g$. □

LEMMA 2.4.2-19: If $\langle uabs, conc\rangle$ is a pair of semi-adjoined (adjoined) functions between M1 and M2 then $\langle R\ uabs, R\ conc\rangle$ is a pair of semi-adjoined (adjoined) functions between L-t>M1 and L-t>M2. □

In the above definitions some of the functions ought to have been indexed, e.g. R should be indexed by the complete lattice L. Hopefully this and other omissions do not lead to confusion.

None of $\langle uabsX, concX\rangle$, $\langle uabs+, conc+\rangle$, $\langle uabs^*, conc^*\rangle$ or $\langle uabs\text{-}c{>}, conc\text{-}c{>}\rangle$ is (in general) exact. It would of course be possible to obtain this. But we feel that it is not worth the effort to obtain this: To obtain exactness we would have to introduce new domain constructors, one for each of X, +, * and -c>. To do so would only complicate our notation. Furthermore, we do not need exactness in our later example.

2.4.3 Detailed Development

In subsection 2.4.1 we explained why it was reasonable to require <uabs,conc> to be a pair of semi-adjoined functions and we stated the definition of a pair of adjoined functions. In this subsection we motivate why it is often natural to require that uabs and conc are adjoined.

As in 2.4.1 let L and M be complete lattices. We do not investigate whether weaker conditions are sufficient. In the discussion, but not the formal development, we assume $L=\mathcal{P}(Sta)$ where Sta is the set of program states.

Let <uabs,conc> be a pair of semi-adjoined functions between L and M. This ensures that $uabs(l)\in M$ is a "safe" representation of l: $conc(uabs(l))\sqsubseteq lp \Rightarrow l\sqsubseteq lp$. Here we think of l as the set of states occurring at some arc and lp as the set of states for which some program transformation is meaning-preserving.

It may be that the representation $uabs(l)$ of l is coarser than necessary, i.e. $conc(uabs(l))\not\sqsubseteq lp$ even though there is $m\in M$ such that $l\sqsubseteq conc(m)\sqsubseteq lp$. This means that using uabs' with $uabs'(l)=m$ would enable the application of an otherwise rejected program transformation.

EXAMPLE 2.4.3-1(a): Define $L=(\{l1,l2,l3,l4\},\sqsubseteq)$ with $l1\sqsubseteq l2\sqsubseteq l3\sqsubseteq l4$ and $M=(\{m1,m2,m3,m4\},\sqsubseteq)$ with $m1\sqsubseteq m2\sqsubseteq m3\sqsubseteq m4$. Define $conc: M\to L$ by $conc(m4) = l4$, $conc(m3) = conc(m2) = l3$ and $conc(m1) = l1$. Also $l = l2$ and $lp = l3$ and $uabs = \lambda l'.m4$. Then <uabs,conc> is a pair of semi-adjoined functions, but $conc(\ uabs(\ l))\not\sqsubseteq lp$ even though $l \sqsubseteq conc(m3) \sqsubseteq lp$ so that e.g. $uabs' = (\lambda l'.\ l'=l1 \to m1,\ l'\sqsubseteq l3 \to m3,\ m4)$ is preferable to uabs. []

In many situations this phenomenon is undesirable. To avoid it we need $(conc\circ uabs)(l) \sqsubseteq \{conc(m)\ |conc(m)\sqsupseteq l\}$ because when $lp = conc(m) \sqsupseteq l$ we want to avoid $(conc \circ uabs)(l) \not\sqsubseteq lp$. (A "realistic" example along these lines is [CoC79,Example 5.1.0.1]). Since $conc \circ uabs$ is extensive the above condition is equivalent to $conc(uabs(l)) = \sqcap\{\ conc(m)\ |conc(m)\sqsupseteq l\}$. This motivates

DEFINITION 2.4.3-2: <uabs,conc> is a pair of <u>quasi-adjoined</u> functions between L and M iff <uabs,conc> is a pair of semi-adjoined functions between L and M and $\forall l\in L$: $conc(uabs(l)) = \sqcap\{conc(m)\ |conc(m)\sqsupseteq l\}$. []

EXAMPLE 2.4.3-1(b): The pair <uabs,conc> is a pair of semi-adjoined functions that is not quasi-adjoined. The pair <uabs',conc> is a pair of quasi-adjoined functions. Note that uabs' and conc are not adjoined. []

To relate the concepts introduced we state:

LEMMA 2.4.3-3: If <uabs,conc> is a pair of adjoined functions then <uabs,conc> is a pair of quasi-adjoined functions and
$uabs = \lambda l.\ \sqcap\{m|\ conc(m)\sqsupseteq l\}$ and is a complete-$\sqcup$-morphism and
$conc = \lambda m.\ \sqcup\{l'|\ uabs(l')\sqsubseteq m\}$ and is a complete-$\sqcap$-morphism. []
Proof is shown in appendix 2. []

To motivate the definition of adjoined we first consider how to transform the test $l\sqsubseteq lp$ to a test in M. It is desirable to find a "safe" test in M, since during data flow analysis we only have M available. We imagine the test in M to be of the form $uabs(l) \sqsubseteq dabs(lp)$ for some function $dabs: L\to M$. We cannot use dabs = uabs because $uabs(l) \sqsubseteq uabs(lp) \not\Rightarrow l\sqsubseteq lp$. Below we investigate the natural requirements upon dabs; these are dual to the requirements upon uabs.

Since uabs(l) $\sqsubseteq$ dabs(lp) => conc(uabs(l)) $\sqsubseteq$ conc(dabs(lp)) => l $\sqsubseteq$ conc(dabs(lp)) whenever <uabs,conc> is a pair of semi-adjoined functions it is natural to impose:

ASSUMPTION 2.4.3-4: conc$\circ$dabs is reductive. []

and similarly to 2.4.1-3:

ASSUMPTION 2.4.3-5: dabs is isotone. []

DEFINITION 2.4.3-6: <dabs,conc> is a pair of <u>semi-down-adjoined</u> functions between L and M iff dabs:L->M is isotone and conc:M->L is isotone and conc$\circ$dabs is reductive. []

OBSERVATION 2.4.3-7: If uabs and conc are semi-adjoined and dabs and conc are semi-down-adjoined then $\forall$l$\forall$lp: uabs(l) $\sqsubseteq$ dabs(lp) => l$\sqsubseteq$lp. []

EXAMPLE 2.4.3-1(c): Define dabs:L->M by dabs = λl'.m1. Then <dabs,conc> is a pair of semi-down-adjoined functions. Note that uabs(l)$\not\sqsubseteq$dabs(lp). []

Perhaps <uabs,conc> should be called a pair of (semi-,quasi-) <u>up</u>-adjoined functions, but we do not because we only use dabs in this subsection.

Suppose that uabs and conc are semi-adjoined and dabs and conc are semi-down-adjoined. It then may be true that uabs(l) $\not\sqsubseteq$ dabs(lp) even though conc(uabs(l)) $\sqsubseteq$ conc(dabs(lp)). Intuitively this means that we cannot exploit in M all the information of uabs(l) and dabs(lp). An example of this is given in 2.4.3-1(d) below. Also it may be that uabs(l) $\not\sqsubseteq$ dabs(lp) even though $\exists$m,mp$\in$M: m $\sqsubseteq$ mp $\wedge$ l $\sqsubseteq$ conc(m) $\wedge$ conc(mp) $\sqsubseteq$ lp. Intuitively, this suggests that using uabs" with uabs"(l)=m and dabs" with dabs"(lp)=mp would be preferable to using uabs and dabs. An example of this is given in 2.4.3-1(d) below.

Often these two possibilities are undesirable. To avoid them we need:

DEFINITION 2.4.3-8: <dabs,conc> is a pair of <u>quasi-down-adjoined</u> functions between L and M iff <dabs,conc> is a pair of semi-down-adjoined functions and conc $\circ$ dabs = λl'. $\bigsqcup${ conc(m) |conc(m)$\sqsubseteq$l'} []

DEFINITION 2.4.3-9: <dabs,conc> is a pair of <u>down-adjoined</u> functions between L and M iff dabs:L->M and conc:M->L and $\forall$l$\forall$m: dabs(l)$\sqsupseteq$ m <=> l $\sqsupseteq$ conc(m). []

LEMMA 2.4.3-10: If <dabs,conc> is a pair of down-adjoined functions then <dabs,conc> is a pair of quasi-down-adjoined functions and dabs = λl'. $\bigsqcup$(m| conc(m)$\sqsubseteq$l'} and is a complete-$\sqcap$-morphism and conc = λm. $\sqcap${l'| dabs(l')$\sqsupseteq$m} and is a complete-$\sqcup$-morphism. []
Proof is shown in appendix 2. []

EXAMPLE 2.4.3-1(d): As an example undesirable property we have uabs(l) $\not\sqsubseteq$ dabs(lp) even though (for m=m2 and mp=m3): m$\sqsubseteq$mp $\wedge$ l$\sqsubseteq$conc(m) $\wedge$ conc(mp)$\sqsubseteq$lp.

Define dabs':L->M by dabs'(l4)=m4, dabs'(l3)=m2, dabs'(l2)=dabs'(l1)=m1. Then <dabs',conc> is a pair of quasi-down-adjoined functions that is not also down-adjoined. But even though uabs' and conc are quasi-adjoined we have uabs'(l) $\not\sqsubseteq$ dabs'(lp), conc(uabs'(l)) $\sqsubseteq$ conc(dabs'(lp)) and

$m \sqsubseteq mp \wedge l \sqsubseteq conc(m) \wedge conc(mp) \sqsubseteq lp$. So the undesirable properties are not avoided by requiring both pairs to be "quasi" rather than "semi".

Finally define uabs" by uabs"(l4)=m4, uabs"(l3)=uabs"(l2)=m2, uabs"(l1)=m1 and define dabs" by dabs"(l4)=m4, dabs"(l3)=m2, dabs"(l2)=dabs"(l1)=m1. Then uabs" and conc are adjoined and dabs" and conc are down-adjoined. Furthermore, uabs'(l) $\sqsubseteq$ dabs"(lp) and uabs"(l) $\sqsubseteq$ dabs'(lp) and uabs"(l) $\sqsubseteq$ dabs"(lp). []

Based on the above example it seems natural to require that uabs and conc must be quasi-adjoined and dabs and conc be down-adjoined; or that uabs and conc be adjoined and dabs and conc quasi-down-adjoined. That this works in general follows from:

LEMMA 2.4.3-11: If uabs and conc are adjoined and dabs and conc are quasi-down-adjoined or if uabs and conc are quasi-adjoined and dabs and conc are down-adjoined then for all l, lp:

$$uabs(l) \sqsubseteq dabs(lp) \Leftrightarrow (\exists m,mp: m \sqsubseteq mp \wedge l \sqsubseteq conc(m) \wedge conc(mp) \sqsubseteq lp)$$
$$\Leftrightarrow conc(uabs(l)) \sqsubseteq conc(dabs(lp)) \quad []$$

Proof is shown in appendix 2. []

In appendix 1 we show how dabs can sometimes be defined from uabs.

Discussion

Our definition of a "pair of adjoined functions" (Definition 2.4.1-6) is equivalent to the "pair of adjoined functions" of [CoC79,definition5.3.0.1]. This follows from Lemma 2.4.3-3 which shows that it is not necessary to explicitly require uabs and conc to be isotone. The definitions of semi- and quasi-adjoined as well as (semi-,quasi-) down-adjoined are new.

Below we further compare our development with the literature. The motivation leading to introducing "pair of semi-adjoined functions" is straight-forward. To motivate stronger requirements we considered the phenomenon that conc(uabs(l))$\not\sqsubseteq$lp even though $\exists$m: l $\sqsubseteq$ conc(m) $\sqsubseteq$ lp. We showed that this phenomenon cannot occur when uabs and conc are quasi-adjoined.

The motivation leading to quasi-adjoined has been adapted from [CoC79,section 5.1]. There M'$\subseteq$L is assumed so that conc:M'->L is λm.m. The motivation of [CoC79] leads to:

ASSUMPTION [CoC79,5.1.0.2]: $\forall l \in L$: {m'$\in$M' | l$\sqsubseteq$m'} must have a least element[]

DEFINITION [CoC79,5.2.0.1]: The approximation operator is uco:L->M' defined by uco(l) = $\sqcap${ m'$\in$M' | l$\sqsubseteq$m'}. []

That the functionality is right is from the above assumption. Note that uco:L->M' is the approximation operator iff <uco,λm.m> is a pair of quasi-adjoined functions between L and M'. This remark shows that [CoC79,Theorem 5.2.0.2] is a special case of the more generally applicable:

LEMMA 2.4.3-12: If <uabs,conc> is a pair of quasi-adjoined functions between L and M then conc $\circ$ uabs is the unique upper closure operator uco:L->L so that {uco(l) | l$\in$L} = {conc(m) | m$\in$M}. []

Proof is shown in appendix 2. []

When [CoC79] considers the connection between uabs and conc (Section 5.3) it is required that <uabs,conc> must be a pair of adjoined functions. In fact, the accompanying explanation argues in favour of <uabs,conc> being an exact pair of adjoined functions. In other work ([CoC77a], [Cou79]) only exact pairs of adjoined functions are defined (the words "exact" and "adjoined" are not used).

In our view the motivation of [CoC79] only suffices to require that <uabs,conc> must be a pair of quasi-adjoined functions. To find a satisfactory motivation for the adjoined condition we introduced the dual concepts (semi-, quasi-) down-adjoined. We believe that these concepts are interesting in their own right. Finally, a nice thing about "exact" is that when <uabs,conc> is an exact pair of adjoined functions then $\{conc(uabs(l)) \mid l \in L\}$ and M are isomorphic.

By way of digression we note:

LEMMA 2.4.3-13: An upper closure operator need not be continuous. □
Proof is shown in appendix 2. □

This shows that we cannot prove [CoC77b,Corollary 4.1.3.2] in our setting.

CHAPTER 3

History-Insensitive Analyses

In section 3.1 we develop a framework that enables us to define a non-standard semantics that specifies data flow information to be associated with a program. The key ingredients in developing this framework are Denotational Semantics and Abstract Interpretation. The framework is applicable to "history-insensitive" analyses. An example is the "constant propagation" analysis formulated in section 3.2. We also compare our formulation of "constant propagation" to other formulations, including that of traditional data flow analysis.

3.1 The General Framework

Intuitively, $\mathcal{P}\mathrm{std}[\![\mathrm{pro}]\!]\mathrm{inp}$ is the result of executing the program pro with input inp. History-insensitive data flow analysis consists in computing properties of the set of states that are possible at some point (of pro) during an execution where inp is any one element of $\underline{\mathrm{inp}}$, for some subset $\underline{\mathrm{inp}}$ of Inp. Our approach is to consider an interpretation apr such that $\mathcal{P}\mathrm{apr}[\![\mathrm{pro}]\!]\underline{\mathrm{inp}}$ specifies these properties. - As mentioned in section 2.2 we identify a point by a place, i.e. an occurrence and text-string pair.

In this section we define interpretations col, sts and ind. We now briefly relate these to each other and std. Interpretation sts is used to map places to the set of states that are possible there. It thus bears a strong relationship to the static semantics of Floyd (section 2.4.1).

To relate a description of the set of states possible at some place (specified by means of apr) to the actual set of states possible at that place (specified by means of sts) we use the approach of Abstract Interpretation. That is we specify a pair of semi-adjoined functions. Interpretation apr is often defined in terms of sts and the pair of semi-adjoined functions; it is then denoted ind. Interpretation apr or ind used together with the semantic functions is called an approximate semantics (in contrast to the static semantics). We use an approximate semantics to specify the data flow analysis information to be associated with a program.

The connection between sts and ind implies that we can formally relate $\mathcal{P}\mathrm{sts}[\![\mathrm{pro}]\!]$ and $\mathcal{P}\mathrm{ind}[\![\mathrm{pro}]\!]$ (Theorem 3.1-10 and Lemma 3.1-12). In contrast, we cannot find a similar formal relationship between $\mathcal{P}\mathrm{sts}[\![\mathrm{pro}]\!]$ and $\mathcal{P}\mathrm{std}[\![\mathrm{pro}]\!]$. To reduce this gap we introduce col: Now $\mathcal{P}\mathrm{sts}[\![\mathrm{pro}]\!]$ and $\mathcal{P}\mathrm{col}[\![\mathrm{pro}]\!]$ can be formally related and the connection between col and std is "intuitively clear". The results of chapter 5 and 6 can be viewed as supporting these intuitions.

The "Collecting" Interpretation

We cannot formally prove the correctness of a claim: "sta is the set of states that are possible at pla during an execution of pro with input inp ". The claim is operational in nature: we observe an interpreter executing pro and collect the set of states that reach pla. On the other hand 𝒫std⟦pro⟧ is simply a mathematical function so it is meaningless to consider how it is "executed".

We must therefore define what we mean by "the states that reach pla during an execution of pro with input inp ". We define this information to be 𝒫col⟦pro⟧(inp)(pla) where col is specified in Table 3.1-A. Hopefully, this definition captures the intuitive concept: It is upon this definition we develop our framework for describing data flow analyses, and it is with respect to this definition that we characterize the data flow information specified by ind.

Intuitively, col is only a small extension of std. Domain A is defined as Pla -c> 𝒫(Sta) and the function attach is now given a non-trivial meaning. Thus 𝒫col⟦pro⟧(inp)(pla) is a set of states.

LEMMA 3.1-1: Table 3.1-A specifies an interpretation. []
Proof is essentially the same as that of Lemma 2.2-5. []

Technically, all constants, auxiliary functions and primitive functions have been changed as well as domains A and C. Neither continuations (elements of C) nor the function setup can be continuous {a}. Strictly speaking, the notation col-g=std-g is nonsense because col-A≠std-A. But we will tolerate this loose notation since the same λ-expression may be used to define both.

The central ingredient of col is attach(pla)(c)(sta), that records the fact that the point pla of the program is reached with the state sta. The function records this information by joining it with the result of (c sta). Another possibility would be to use S = Sta X (Pla -c> 𝒫(Sta)) as the state and let attach(pla)(c)(s) collect information in s before supplying it to c. An advantage of our approach is that we may obtain data flow information that is not ⊥, even for looping programs. This need not be the case in the other approach (because FIX(g)=⊥ <=> g⊥=⊥). Unlike the case in section 2.2 it is important to obtain data flow information which is not ⊥, even for looping programs.

Data flow information (pertaining to some program pro) is often used to guarantee that applying some program transformation to pro yields an equivalent program. To be useful data flow information therefore must be related to the semantics of pro. When we specify data flow information (by 𝒫ind⟦pro⟧inp) we do relate it to 𝒫col⟦pro⟧. It is therefore undesirable that we cannot characterize 𝒫col⟦pro⟧ in terms of 𝒫std⟦pro⟧. But even then it is possible to formally prove that the specified data flow information can be used to guarantee that the program transformation applied to pro yields an equivalent program. The results of chapter 6 are useful for doing this. We regard it a major positive virtue of our approach that it can be used to prove program transformations correct with respect to a denotational

---

{a} A reason is that sta1⊑sta2 ≠> {sta1}⊆{sta2}; that is: the orderings of Sta and 𝒫(Sta) bear little relationship to one another. This is further discussed later in this section and in chapter 7.

```
TABLE 3.1-A --- INTERPRETATION col
==================================

Domains (of col)
----------------
    c ∈ C = Sta -t> A
  inp ∈ I = Inp
    a ∈ A = Pla -c> 𝒫(Sta)
  sta ∈ S = Sta

   (See Table 2.2-B for)
   (Pla, Inp, Sta, ...)

Constants
---------
  col-wrong ∈ C
  col-wrong = ⊥

  col-finish ∈ C
  col-finish = ⊥

Pseudo-Semantic Functions
-------------------------
  col-ℬ ∈ Bas° -c> Val
  col-ℬ = std-ℬ

  col-𝒪 ∈ Ope° -c>(ValXVal -c>Val)
  col-𝒪 = std-𝒪

Auxiliary Functions
-------------------
  col-setup ∈ C -c> I -t> A
  col-setup = std-setup

  col-cond ∈ C X C -c> C
    col-Vconc ∈ Sta -c> T
    col-Scond ∈ Sta -c> T
    col-Bcond ∈ Sta -c> Sta
  col-cond = std-cond
    col-Vcond = std-Vcond
    col-Scond = std-Scond
    col-Bcond = std-Bcond

  col-attach ∈ Pla -c> C -c> C
  col-attach(pla)(c)(s) =
    (c s) ⊔ ⊥[{s}/pla]

Primitive Functions
-------------------
  col-g ∈ Par -t> C -c> C
    col-Vg ∈ Par -t> Sta -c> T
    col-Bg ∈ Par -t> Sta -c> Sta
  col-g = std-g
    col-Vg = std-Vg
    col-Bg = std-Bg
```

semantics.

The "Static" Interpretation

Some notion of static semantics is important for formulating an approximate semantics by means of abstract interpretation (e.g. [CoC77a]). Given a set inp of inputs the static semantics determines the set of states that are possible at some place during any one execution of the program pro with input inp∈inp.

We define the static semantics to be 𝒫sts⟦pro⟧inp where sts is specified in Table 3.1-B.

```
TABLE 3.1-B --- INTERPRETATION sts
==================================

Domains (of sts)
----------------
    c ∈ C = 𝒫(Sta) -c> A
  inp ∈ I = 𝒫(Inp)
    a ∈ A = Pla -c> 𝒫(Sta)
  sta ∈ S = 𝒫(Sta)

Constants
---------
  sts-wrong ∈ C
  sts-wrong = ⊥

  sts-finish ∈ C
  sts-finish = ⊥

Pseudo-Semantic Functions
-------------------------
  sts-ℬ ∈ Bas° -c> Val
  sts-ℬ = std-ℬ

  sts-𝒪 ∈ Ope° -c> (Val X Val -c> Val)
  sts-𝒪 = std-𝒪

Auxiliary Functions
-------------------
  sts-setup ∈ C -c> I -c> A
  sts-setup c inp =
    c {<λide."nil" inVal, inp, <>, <>> |inp∈inp}

  sts-cond ∈ C X C -c> C
    sts-Dt-cond ∈ S -c> S
    sts-Df-cond ∈ S -c> S
  sts-cond(c1,c2) sta =
    c1(sts-Dt-cond(sta)) ⊔ c2(sts-Df-cond(sta))
  sts-Dt-cond(sta) =
    {std-Bcond(sta) |sta∈sta ∧ std-Vcond(sta)=true ∧ std-Scond(sta)=true}
  sts-Df-cond(sta) =
    {std-Bcond(sta) |sta∈sta ∧ std-Vcond(sta)=true ∧ std-Scond(sta)=false}

  sts-attach ∈ Pla-c>C-c>C
  sts-attach(pla)(c)(sta) =
    (c sta) ⊔ (⊥[sta/pla])

Primitive Functions
-------------------
  sts-g ∈ Par -t> C -c> C
    sts-Dg ∈ Par -t> S -c> S
  sts-g(par)(c)(sta) =
    c(sts-Dg(par)(sta))
  sts-Dg(par)(sta) =
    {std-Bg(par)(sta) |sta∈sta ∧ std-Vg(par)(sta) = true}
```

LEMMA 3.1-2: Table 3.1-B specifies an interpretation. []
Proof is shown in appendix 3. []

In Table 3.1-B functions Dg (for "do") intuitively correspond to the Bf<arc1, arc2> of sections 2.3 and 2.4. Because of our previous discussion of col it appears reasonable to require that 𝒫sts⟦pro⟧inp = ⋃{𝒫col⟦pro⟧inp |inp∈inp}. Unfortunately, this is not always the case. By Theorem 3.1-6 below the result does hold if ∀inp∈inp: topfree[Inp](inp). Here topfree is defined as follows:

DEFINITION 3.1-3:
1) If L is flat then topfree[L]:L->B is topfree[L](l) = (l≠⊤)

2) topfree[L1+...+Ln] : L1+...+Ln->B is topfree[L1+...+Ln](l) =
(l≠⊤ ∧ [l = (li inL1+...+Ln) => topfree[Li](li)])

3) topfree[L1X...XLn] : L1X...XLn->B is topfree[L1X...XLn]<l1,...,ln> =
(∀i∈{1,...,n}: topfree[Li](li))

4) topfree[L*]: L*->B is topfree[L*](l) =
(l≠⊤ ∧ [l=<l1,...,ln> => ∀i∈{1,...,n}: topfree[L](li)])

5) topfree[L-c>M]: (L-c>M)->B is topfree[L-c>M](f) =
∀l∈L: [topfree[L](l) => topfree[M](f(l))] □

We do not believe it is restrictive only to consider inp such that ∀inp∈inp: topfree[Inp](inp). One reason is that many programs pro have 𝒫col⟦pro⟧inp = λpla.Sta whenever topfree[Inp](inp) = false. An example is pro = BEG DCL x:=0 IN READ x END for inp = ⊤ (see col-read). When inp∈inp this implies ⊔{𝒫col⟦pro⟧ inp |inp∈inp} = λpla.Sta so that we cannot use the data flow information for anything. In fact an intuitively desirable choice of inp seems to be {inp |inp∈Inp ∧ pure[Inp](inp)}.

Before we show the connection between interpretations col and sts we need to impose some assumptions that will be needed in the proof.

ASSUMPTION 3.1-4: ∀bas∈Bas: topfree[Val](std-ℬ⟦bas⟧) and
∀ope∈Ope∀val1,val2∈Val: topfree[Val](val1) ∧ topfree[Val](val2) =>
topfree[Val](std-𝒪⟦ope⟧<val1,val2>) □

OBSERVATION 3.1-5: For oo:LXL->M any one of ++, ∧∧, ∨∨, << we have
topfree[L](l1) ∧ topfree[L](l2) => topfree[M](l1 oo l2) for suitable
L and M. □

THEOREM 3.1-6: If ∀inp∈inp: topfree[Inp](inp) then
𝒫sts⟦pro⟧inp = ⊔{𝒫col⟦pro⟧inp |inp∈inp} □
Proof is shown in appendix 3. □

This theorem may be compared with [CoC77a,p.240] where a static semantics (corresponding to sts) is related to an operational semantics ("corresponding" to col): see conditions (α) and (β) in that paper.

Part of the proof of Theorem 3.1-6 is to show that "⊤ cannot occur". One could as in [Sto77,p.203] state a lemma saying so, but we believe that our formulation is precise.

Store Semantics versus Standard Semantics: In section 2.2 we chose to use a kind of store semantics. This is because the obvious analogue of sts as a standard semantics does not fulfil Theorem 3.1-6. This "obvious analogue" would have 𝒫(EnvXInpXOut) and (indirectly) (𝒫(Val))* rather than 𝒫(EnvXInpXOutXVal*) which is more relational. It is easy to give example programs such that Theorem 3.1-6 would not hold for this version of sts.

The "Induced" Interpretation

For the same reasons as in section 2.4.1 it is for computational purposes desirable to replace the use of sts by some interpretation apr that does not use 𝒫(Sta). We require apr to be an approximate interpretation:

DEFINITION 3.1-7: An <u>approximate interpretation</u> is an interpretation having I=℘(Inp), A = Pla -c> S and C ⊑ S -i> A. []

OBSERVATION 3.1-8: sts is an approximate interpretation. []

Let apr1 (e.g. sts) and apr2 be two approximate interpretations. Assume that <uabs,conc> is a pair of semi-adjoined functions between apr1-S and apr2-S. If we want to use apr2 instead of apr1 then, much as in section 2.4, we want ∀<u>inp</u>∈℘(Inp): 𝒫apr1⟦pro⟧<u>inp</u> ⊑ (R conc) (𝒫apr2⟦pro⟧<u>inp</u>) where R is as in 2.4.2-18. An alternative would be (R uabs) (𝒫apr1⟦pro⟧<u>inp</u>) ⊑ 𝒫apr2⟦pro⟧<u>inp</u>, but our choice is preferable because it is immune to a bad choice of uabs.

It is convenient to find a "local test" apr1 ⊑<uabs,conc> apr2 implying ∀<u>inp</u>: 𝒫apr1⟦pro⟧<u>inp</u> ⊑ (R conc) (𝒫apr2⟦pro⟧<u>inp</u>) but easier to apply:

DEFINITION 3.1-9: Let apr1 and apr2 be approximate interpretations and <uabs,conc> a pair of semi-adjoined functions between apr1-S and apr2-S. Define P-C: (apr1-C X apr2-C) ->B by P-C(apr1-c, apr2-c) = (apr1-c ∘ conc ⊑ (R conc) ∘ apr2-c). Then we write apr1 ⊑<uabs,conc> apr2 iff

a) P-C(apr1-wrong,apr2-wrong) and P-C(apr1-finish,apr2-finish)

b) P-C(apr1-c1,apr2-c1) ∧ P-C(apr1-c2,apr2-c2) =>
   P-C(apr1-cond(apr1-c1,apr1-c2), apr2-cond(apr2-c1,apr2-c2))

c) P-C(apr1-c,apr2-c) =>
   P-C(apr1-attach(pla)(apr1-c), apr2-attach(pla)(apr2-c))

d) P-C(apr1-c,apr2-c) => ∀<u>inp</u>∈℘(Inp):
   apr1-setup(apr1-c)<u>inp</u> ⊑ (R conc) (apr2-setup(apr2-c) <u>inp</u>)

e) For all primitive functions g ∈ Par -t> C -c> C that
   P-C(apr1-c,apr2-c) => P-C(apr1-g(par)(apr1-c), apr2-g(par)(apr2-c)) []

We have chosen to let all approximate interpretations have I = ℘(Inp) rather than assuming the existence of a pair of semi-adjoined functions between apr1-I and apr2-I. This makes the notation less involved.

THEOREM 3.1-10: If <uabs,conc> is a pair of semi-adjoined functions and apr1 ⊑<uabs,conc> apr2 then
∀<u>inp</u>∈℘(Inp): ∀pro∈Pro: 𝒫apr1⟦pro⟧<u>inp</u> ⊑ (R conc) (𝒫apr2⟦pro⟧<u>inp</u>) []
Proof is shown in appendix 3. []

This theorem only assumes that <uabs,conc> is a pair of semi-adjoined functions. Related theorems, with more or less the same underlying motivation, are given in [CoC79,section7.1] but there <uabs,conc> is assumed to fulfil the stronger condition of being a pair of adjoined functions, although the proofs seem only to use properties of semi-adjoined functions.

Let S be a complete lattice and <uabs,conc> a pair of semi-adjoined functions between ℘(Sta) and S. It is useful to define an approximate interpretation that is "induced by" <uabs,conc> from sts. Table 3.1-C defines this interpretation ind, called the <u>induced interpretation</u>. We sometimes write ind<uabs,conc> to be precise about the pair of semi-adjoined functions used. - This notion of induced is similar to the one mentioned in section 2.4.1.

LEMMA 3.1-11: Table 3.1-C specifies an approximate interpretation. []
Proof is similar to the proof of Lemma 3.1-2. []

The usefulness of ind is guaranteed by

LEMMA 3.1-12: sts $\sqsubseteq$<uabs,conc> (ind<uabs,conc>) []
Proof is shown in appendix 3. []

```
TABLE 3.1-C --- INTERPRETATION ind
==================================

Domains (of ind)
----------------
    c ∈ C = S -i> A
  inp ∈ I = ℘(Inp)
    a ∈ A = Pla -c> S
    s ∈ S
                  uabs: ℘(Sta)->S
                  conc: S->℘(Sta)

Constants
---------
  ind-wrong ∈ C
  ind-wrong = ⊥

  ind-finish ∈ C
  ind-finish = ⊥

Pseudo-Semantic Functions
-------------------------
  ind-𝓑 ∈ Bas° -c> Val
  ind-𝓑 = std-𝓑

  ind-𝒪 ∈ Ope° -c> (Val X Val -c> Val)
  ind-𝒪 = std-𝒪

Auxiliary Functions
-------------------
  ind-setup ∈ C -c> I -i> A
  ind-setup c inp =
    c( uabs {<λide."nil"inVal, inp, <>, <>> |inp∈inp})

  ind-cond ∈ C X C -c> C
    ind-Dt-cond ∈ S -i> S
    ind-Df-cond ∈ S -i> S
  ind-cond(c1,c2) s =
    c1(ind-Dt-cond(s)) ⊔ c2(ind-Df-cond(s))
  ind-Dt-cond =
    uabs∘sts-Dt-cond∘conc
  ind-Df-cond =
    uabs∘sts-Df-cond∘conc

  ind-attach ∈ Pla -c> C -c> C
  ind-attach(pla)(c)(s) =
    (c s) ⊔ (⊥[s/pla])

Primitive Functions
-------------------
  ind-g ∈ Par -t> C -c> C
    ind-Dg ∈ Par -t> S -i> S
  ind-g(par)(c)(s) =
    c(ind-Dg(par)(s))
  ind-Dg(par)(s) =
    uabs∘sts-Dg(par)∘conc
```

One nice property of ind is the following. Suppose apr is any approximate interpretation with apr-S = (ind<uabs,conc>)-S. Then $\langle\lambda s.s,\lambda s.s\rangle$ is a pair of adjoined functions between S and S, and

LEMMA 3.1-13:
1) ind<uabs,conc> $\sqsubseteq$<λs.s,λs.s> apr => sts $\sqsubseteq$<uabs,conc> apr

2) If <uabs,conc> is an exact pair of adjoined functions then
ind<uabs,conc> $\sqsubseteq$<λs.s,λs.s> apr <=> sts $\sqsubseteq$<uabs,conc> apr. []

Proof is shown in appendix 3. []

The lemma states that whenever <uabs,conc> is an exact pair of adjoined functions then ind<uabs,conc> is, in some sense, "the best upper approximation to sts". This may be compared with [CoC79,section7.2] where <uabs,conc> only needs to be a pair of adjoined functions. By a slight change of Table 3.1-C (e.g. ind-g(par)(ind-c) = R(uabs $\circ$ conc) $\circ$ ind-c $\circ$ ind-Dg(par) instead of the earlier ind-c $\circ$ ind-Dg(par)) it appears that we can weaken "exact" to "uabs $\circ$ conc is continuous" (for Lemma 3.1-11 to hold). However, we choose not to do this.

<u>Properties of approximate and induced interpretations</u>

The definition of ...$\sqsubseteq$<uabs,conc>... is not the only possible one. Three other ways of defining P-C (and thereby ...$\sqsubseteq$<uabs,conc>...) are

P-C1(apr1-c,apr2-c) = [(R uabs) $\circ$ apr1-c $\circ$ conc $\sqsubseteq$ apr2-c]
P-C2(apr1-c,apr2-c) = [(R uabs) $\circ$ apr1-c $\sqsubseteq$ apr2-c $\circ$ uabs]
P-C3(apr1-c,apr2-c) = [apr1-c $\sqsubseteq$ (R conc) $\circ$ apr2-c $\circ$ uabs]

Let P-C be the predicate used in 3.1-9. For <uabs,conc> a pair of adjoined functions all of P-C, P-C1, P-C2 and P-C3 are equivalent, whereas for <uabs,conc> a pair of semi-adjoined functions they can be different. Predicates P-C1 and P-C2 are not desirable because then we cannot show sts $\sqsubseteq$<uabs,conc> ind. Predicate P-C implies P-C3 so we have chosen the stronger predicate. We regard it desirable that P-C does not depend on uabs.

Theorem 3.1-10 assures us that the "local test" apr1 $\sqsubseteq$<uabs,conc> apr2 implies the "global test" $\forall$pro $\forall$<u>inp</u>: $\mathcal{P}$apr1$[\![$pro$]\!]$<u>inp</u> $\sqsubseteq$ (R conc)($\mathcal{P}$apr2$[\![$pro$]\!]$<u>inp</u>). We do not have the converse result, contrary to the situation in [CoC79,section7.1]. We now intuitively explain why this is so. Suppose apr1 $\not\sqsubseteq$<uabs,conc> apr2 because some primitive function g does not fulfil condition "e)" of Definition 3.1-9. Then we want to find pro and inp such that $\mathcal{P}$apr1$[\![$pro$]\!]$<u>inp</u> $\not\sqsubseteq$ (R conc)($\mathcal{P}$apr2$[\![$pro$]\!]$<u>inp</u>). Intuitively we want pro to supply g with some c and s that make "e)" of 3.1-9 fail. But it may be that such a program pro does not exist, because the semantic equations impose restrictions upon the parameters that g can possibly get.

We do not regard it as a disadvantage of our approach that the converse result of 3.1-10 does not hold. Indeed, one of the motivations behind introducing "interpretation" was to avoid that our development depends strongly on one particular language. There are, however, methods that may be used to avoid apr1 $\not\sqsubseteq$<uabs,conc> apr2 when $\forall$pro$\forall$<u>inp</u>: $\mathcal{P}$apr1$[\![$pro$]\!]$<u>inp</u> $\sqsubseteq$ (R conc)($\mathcal{P}$apr2$[\![$pro$]\!]$<u>inp</u>). One method is to replace C and S by smaller domains. As an example, C of sts may be replaced by the set of complete-$\sqcup$-morphisms from S to A. If this method is not sufficient then one may restrict the continuations (c) and states (s) to be considered in "e)" of 3.1-9.

One way to compare approximate interpretations is apr1 $\sqsubseteq$<uabs,conc> apr2. Below we restrict ourselves to induced interpretations ind<uabs,conc> where <uabs,conc> is a pair of adjoined functions (between $\mathcal{P}$(Sta) and a complete lattice S) with uabs($\top$)=$\top$. Then we can give results corresponding to the "hierarchy of program analysis frameworks" of [CoC79,section 8].

For any two induced interpretations ind1=ind<uabs1,conc1> and ind2=ind<uabs2,conc2> we may want to find a "local test" that implies ∀pro∀inp: (R conc1)($\mathcal{P}$ind1⟦pro⟧inp) ⊑ (R conc2)($\mathcal{P}$ind2⟦pro⟧inp). The test ind1 ⊑<uabs,conc> ind2 depends on the choice of <uabs,conc> and we want to avoid having to choose <uabs,conc>. That this is possible follows from Lemma 3.1-15 below. The proof uses the following lemma that shows that conc ∘ uabs essentially determines how approximate ind<uabs,conc> is.

LEMMA 3.1-14: Let <uabs,conc> be a pair of adjoined functions between $\mathcal{P}$(Sta) and some complete lattice S such that uabs(⊤)=⊤. Then <conc ∘ uabs, λ sta.sta> is a pair of semi-adjoined functions and ∀pro ∀inp:
(R uabs)($\mathcal{P}$(ind<conc ∘ uabs,λsta.sta>)⟦pro⟧inp) = $\mathcal{P}$(ind<uabs,conc>)⟦pro⟧inp □
Proof is shown in appendix 3. □

LEMMA 3.1-15: Let <uabs1,conc1> and <uabs2,conc2> be pairs of adjoined functions (between $\mathcal{P}$(Sta) and complete lattices S1 and S2, respectively) such that uabs1(⊤)=⊤ and uabs2(⊤)=⊤. If conc1 ∘ uabs1 ⊑ conc2 ∘ uabs2 then ∀pro ∀inp: (R conc1)($\mathcal{P}$ind<uabs1,conc1>⟦pro⟧inp) ⊑
(R conc2)($\mathcal{P}$ind<uabs2,conc2>⟦pro⟧inp) □
Proof is shown in appendix 3. □

One can view ind<conc ∘ uabs, λ sta.sta> as a representative of a set of induced interpretations {ind<uabs',conc'>| conc' ∘ uabs'= conc ∘ uabs}. With a slight change of our definition of induced interpretation we can show that these representatives form a complete lattice.

LEMMA 3.1-16:
{uco | uco:$\mathcal{P}$(Sta)->$\mathcal{P}$(Sta) is an upper closure operator} =
{conc ∘ uabs| <uabs,conc> is a pair of adjoined functions between $\mathcal{P}$(Sta) and some complete lattice S, and such that uabs(⊤)=⊤}.
Proof is shown in appendix 3. □

LEMMA 3.1-17: Abbreviate id=λsta.sta and assume ind<uabs,conc> also contains a primitive function dummy = λ c.c ∘ conc ∘ uabs. Let uco1 and uco2 be upper closure operators on $\mathcal{P}$(Sta). Then
uco1 ⊑ uco2 <=> (ind<uco1,id>) ⊑<id,id> (ind<uco2,id>) □
Proof is shown in appendix 3. □

LEMMA 3.1-18 [CoC79,Th.8.0.1][War42,Th.5.3]: The set of upper closure operators on a complete lattice is a complete lattice. □

COROLLARY 3.1-19: If ind<uabs,conc> also contains a primitive function dummy = λc.c ∘ conc ∘ uabs then
{ind<conc ∘ uabs, λ sta.sta> | <uabs,conc> is a pair of adjoined functions between $\mathcal{P}$(Sta) and some complete lattice S such that uabs(⊤)=⊤}
is a complete lattice ordered by ...⊑<λ sta.sta, λ sta.sta>... . □

We have used "pair <uabs,conc> of adjoined functions with uabs(⊤)=⊤" rather than "exact pair <uabs,conc> of adjoined functions". This is because the pairs of adjoined functions <uabsX,concX>, <uabs+,conc+>, <uabs*,conc*> and <uabs-c>,conc-c>> are not exact but do have uabs...(⊤)=⊤.

Discussion

In developing the preceeding framework we have chosen one particular route of development. This is not the only one. One of the "arbitrary" decisions was to make a collecting interpretation (col) before we considered interpretations involving sets of states (sts and ind). Consider developing directly from std an interpretation int involving sets of states. Then int and std would be related as are sts and col. Some problems then would occur: The std interpretation has std-C = Sta -c> A whereas presumably int-C = $\mathcal{P}$(Sta)-c>$\mathcal{P}$(A). The proof of the analogue of Theorem 3.1-6 probably would involve P-C:(std-C$\times$int-C)->B defined by P-C(std-c,int-c) = $\forall \underline{sta}$: {std-c(sta) | sta$\in\underline{sta}$} = int-c($\underline{sta}$). Unfortunately, P-C($\perp$,$\perp$)=false which causes problems in the structural induction (the WHILE loop). A possible remedy is int-C = $\mathcal{P}$(Sta) -c> { $\underline{a}\in\mathcal{P}$(A) | $\perp\in\underline{a}$}, but we have avoided these problems in our approach because col-A=sts-A.

We have found it convenient to use $\mathcal{P}$(Sta) in order to employ the framework of P.Cousot&R.Cousot. Since it is not obvious how to define reflexive domains involving powersets we have avoided reflexive domains. This implies that we are unable to handle language constructs like procedures, jumps, VALOF..., RESULTIS... .

Since these restrictions are severe we informally explain why we have been unable to find a LAMBDA-definable domain ("Scott domain") that may be used instead of $\mathcal{P}$(Sta).

Let T"=({yes,no},$\sqsubseteq$) where t1 $\sqsubseteq$ t2 <=> (t1$\neq$yes $\vee$ t2$\neq$no). Then $\mathcal{P}$(Sta) and Sta -t> T" are lattice isomorphic (see [Grä73] for definitions). But Sta -t> T" is not LAMBDA definable and we would have to use Sta -c> T". We discuss some drawbacks of Sta -c> T" (actually of Sta -i> T" $\supseteq$ Sta -c> T") below:

a) Suppose pro $\underline{at}$ occ is an expression with constant value val such that pure[Val](val). A possible program transformation would replace the expression by a constant evaluating to val. But $\top\sqsupseteq$val implies that there is no a $\in$ Pla -c> Sta -c> T" such that a<occ,"exp)"> <env,inp,out,wit> = yes => wit$\downarrow$1=val. To be able to represent the desired information we probably would have to work with {sta$\in$Sta | topfree[Sta](sta)} (which is not a complete lattice) rather than Sta. We prefer, however, to stay within the more or less standard framework of [Sto77], i.e. to use complete lattices.

b) Suppose we have storable continuations. From an informal sketch of the std ==> col ==> sts development we are lead to believe that the problems will be even more severe than in the above case. When a stored continuation is applied (in sts) the resulting data flow information is likely to be λpla.Sta, i.e. completely useless. It appears that the problems are not remedied by using {sta$\in$Sta | topfree[Sta](sta)}.

Some alternatives to Sta -c> T" are Sta -c> T or Sta -c> T', where T' is the dual lattice of T"[MiS76]. But this does not seem to be appropriate either.

For some simple languages we probably could introduce procedures, labels and VALOF..., RESULTIS... if we had worked from a standard semantics employing a two-stage mapping: [Nie79] gives an example showing that the standard semantics needs not contain reflexive domains even if the store semantics does. But if we work from a standard semantics then Theorem 3.1-6 will not hold.

3.2 Example and Comparison with other Approaches

In this section we show how the general framework of section 3.1 can be used to derive a formulation of the data flow analysis "constant propagation". We then compare our method (as exemplified by the constant propagation example) to methods described in the literature. These are the "traditional data flow analysis" concept of constant propagation, the methods of Donzeau-Gouge and the method of Cousot&Cousot.

It is convenient to specify our constant propagation example as an instance of a class of data flow analyses, which we shall call "value subset analysis". Let Val be the complete lattice of Table 2.2-B. Then a "value subset analysis" is specified by a pair <uabsVa,concVa> of semi-adjoined functions between $\mathcal{P}$(Val) and a complete lattice Va. This pair is used to define the pair <uabs,conc> of semi-adjoined functions (explained below):

uabs = (uabs:no2) $\circ$ (uabs:no3) $\circ$
        C4[(S* uabsVa) $\circ$ uabs*] $\circ$
        C1[(R uabsVa) $\circ$ uabs-c>] $\circ$
        uabsX
conc = concX $\circ$
        C1[conc-c> $\circ$ (R concVa)] $\circ$
        C4[conc* $\circ$ (S* concVa)] $\circ$
        (conc:no3) $\circ$ (conc:no2)

Functions C1, uabs*, ... are defined in section 2.4.2. By Lemma 2.4.2-1:

OBSERVATION 3.2-1: If <uabsVa,concVa> is a pair of semi-adjoined (adjoined) functions between $\mathcal{P}$(Val) and Va then <uabs,conc> is a pair of semi-adjoined (adjoined) functions between $\mathcal{P}$(Sta) and (Ide^{o} -t> Va) X Va*. []

The first task of uabs is to bring $\mathcal{P}$(EnvXInpXOutXWit) to independent attribute form $\mathcal{P}$(Env)X$\mathcal{P}$(Inp)X$\mathcal{P}$(Out)X$\mathcal{P}$(Wit). Then the environment $\mathcal{P}$(Ide^{o}-c>Val) is changed to Ide^{o}-t>$\mathcal{P}$(Val) and then to Ide^{o}-t>Va by use of uabsVa. The witnessed stack is changed from $\mathcal{P}$(Val*) to ($\mathcal{P}$(Val))* and then to Va* by use of uabsVa. Finally the input and output are removed. Note that the order of uabs:no2 and uabs:no3 is crucial.

Several data flow analyses are special kinds of value subset analyses. They include type-determination, constant propagation and the analysis of signs of numerical values. In the formulation of <uabsVa,concVa> it may be useful to employ <uabs+,conc+> and S[i] of section 2.4.2.

Below we specify a constant propagation example because this data flow analysis is found in both "traditional data flow analysis" and [Don79]. We do this by specifying Va and <uabsVa,concVa>.

We define Va = Val$\cup${all,none} with partial order $\sqsubseteq$ such that va1 $\sqsubseteq$ va2 <=> (va1=none $\vee$ va1=va2 $\vee$ va2=all); hence (Va,$\sqsubseteq$) is a flat lattice. We define concVa:Va->$\mathcal{P}$(Val) by concVa(all) = Val, concVa(none) = $\emptyset$ and concVa(val) = {val} otherwise. Also uabsVa(<u>val</u>) = $\sqcap${va | concVa(va) $\sqsupseteq$ <u>val</u>}. Clearly <uabsVa,concVa> is an exact pair of adjoined functions.

Comparison with traditional data flow analysis

The traditional notion of constant propagation was reviewed in section 2.3. To develop a more or less similar constant propagation example in our framework we use <uabsVa, concVa> and <uabs,conc> as specified above. Since traditional constant propagation associates information only with exits of

basic blocks we redefine:
  ind-attach pla c s =
  (pla↓2=="cmd)" -> (c s) $\sqcup$ $\perp$[s/pla], (c s))
Hence, information is only associated with exits of commands. Although exits of commands only roughly correspond to exits of basic blocks we regard the relationship as reasonable. An advantage of our approach is that it unifies global and local analysis because it treats commands and expressions in the same way.

Since $\mathcal{P}$ind⟦pro⟧<u>inp</u> $\in$ Pla -c> ((Ide^{o} -t> Va) X Va*) we can construct a "constant pool" at pla by: {(ide,va) |va=($\mathcal{P}$ind⟦pro⟧<u>inp</u> pla)↓1⟦ide⟧ $\wedge$ va$\notin${all,none}}. This shows a rather close correspondence between the form of the answer expected in traditional data flow analysis and the one we specify. One minor difference is that we work with complete lattices (hence $\perp$, $\top$, none, all) whereas traditional data flow analysis only considers sets and therefore does not consider $\top$ and $\perp$ elements.

This correspondence between traditional data flow analysis and our method only considers the form of the solution. It must be complemented by the results of chapter 5 where we show that $\mathcal{P}$ind⟦pro⟧<u>inp</u> is the MOP solution to a certain (traditional) data flow analysis problem. This MOP solution might not be computable.

<u>Comparison with [Don79]</u>

Since [Don79] is not widely available we briefly overview it below. Sections 1 to 4 of that paper define a toy language, which extends our language to contain VALOF... and RESULTIS... constructs. The semantics is a standard semantics (not store semantics) that employs a 1-stage mapping between identifiers and values (i.e. no locations). The meaning function $\mathcal{P}$ for programs is of functionality Pro -c> Inp -c> Out where $\mathcal{P}$⟦pro⟧inp is the result of "executing" pro with input inp. The "domains" are restricted to be the partially ordered sets that can be defined in DSL[Mos79]. One consequence is that almost no "domain" contains top-elements; thus virtually no complete lattices can be defined.

Sections 5, 6 and 7.1 develop her method and apply it to a constant propagation example. Her notion of constant propagation is slightly different from the traditional concept: she directly associates expresions with their possible values. In the traditional approach one performs global analysis to obtain a pool at each command-exit, and assumes that an unspecified local analysis from these pools associates expressions with values.

The non-standard semantics of sections 5, 6 and 7.1 has $\mathcal{P}$ $\in$ Pro -c> Inp' -c> Occ -c> Prop. For the purposes of this presentation one can think of Inp' as $\mathcal{P}$(Inp). Domain Occ is used in the same way as we use Pla. Domain Prop is essentially {x |x$\in$N+T+{"var"}$\wedge$topfree[N+T+{"var"}](x)} that plays the role of Val. A subtle difference that we ignore is that + is a kind of coalesced sum. To explain in our notation the intention of Prop we define V={x$\in$N+T |topfree[N+T](x)} and the function conc':Prop->$\mathcal{P}$(V) by
  conc'("var") = V
  conc'(t inProp) = { t inN+T, $\perp$}
  conc'(n inProp) = { n inN+T, $\perp$}
  conc'($\perp$) = {$\perp$}
It is somewhat unclear to us whether { t inN+T} or { t inN+T, $\perp$} should be used above. We suspect it is { t inN+T, $\perp$}.

This connection between $\mathcal{P}$(V) and Prop cannot be described by a pair of semi-adjoined functions, because uabs'(V) = "var" implies that uabs' cannot

be isotone. If Prop had been V∪{"var"} with "var" as the top-element then this complication would not arise; but this domain is *not* definable in DSL.
The non-standard semantics of [Don79] contains attach functions in the semantic equations for expressions only. Essentially their purpose is as in our approach. The non-standard semantics has Sta = Exp -c> Prop rather than Sta = Ide -c> Prop. Also several "update" functions (to update the state) are placed in conjunction with the attach function: This appears to be necessary for her "common subexpression elimination" example (section 7.2), but unimportant for the constant propagation example. In our formulation (Table 2.2-C) we have no function that is used as [Don79] uses update.

To develop a more or less similar constant propagation example we use ind<uabs,conc> where <uabsVa,concVa> is as earlier (and determines <uabs,conc>). Since we have placed an attach function in all semantic equations, not only those pertaining to expressions, we redefine ind-attach to:

ind-attach pla c s =
(pla↓2=="exp)" -> (c s) ⊔ ⊥[s/pla], (c s))

The functionality of 𝒫ind⟦pro⟧inp is Pla -c> ((Ide -t> Va) × Va*). To obtain a result on the same form as that of [Don79] we write

λocc.𝒫ind⟦pro⟧{inp |pure[Inp](inp)}<occ,"exp)">↓2↓1

We thereby obtain the same overall effect as [Don79].

When we proved ind correct we showed (Theorems 3.1-6 and 3.1-10 and Lemma 3.1-12) that for inp⊆{inp∈Inp |topfree[Inp](inp)}:

⨆{𝒫col⟦pro⟧inp |inp∈inp} ⊑ (R conc)(𝒫ind⟦pro⟧inp)

A proof of correctness is also given in [Don79]. To explain this proof we informally sketch how to conduct a similar proof in our setting.
First define a "trace"-interpretation tra that mostly is the same as col. Some changes are

tra-A = {0,1,2,...} -t> (Pla×Sta)°
tra-attach pla c sta =
pla↓2="exp)"->(λn. n=0-> <pla,sta> in(Pla×Sta)°, c(sta)(n-1)), c(sta)

It seems plausible to conjecture that 𝒫col⟦pro⟧inp = build(𝒫tra⟦pro⟧inp) where build:tra-A -> col-A is

build(tra-a)pla = {sta |∃n: tra-a(n) = <pla,sta> in(Pla×Sta)°}

Let inp be as above and inp∈inp, n and pla arbitrary. Then we must show that [<pla,sta> in(Pla×Sta)° = 𝒫tra⟦pro⟧(inp)(n)] implies sta∈conc(𝒫ind⟦pro⟧ inp pla). By the above conjecture this is essentially what we have proven.

In summary, there appears to be a close connection between the effects of our ind interpretation and the method of [Don79] (ignoring sections 7.2 and 7.3). Donzeau-Gouge has been careful to make her correctness proofs modular, so that little is to be proven for each data flow analysis. We do believe that specifying a data flow analysis by a pair of semi-adjoined functions allows for even more modularity. Also we believe that this is a more *systematic* approach than that of [Don79]. Our approach is more *general* because [Don79] has no analogue of sts, i.e. an interpretation that may fulfil Theorem 3.1-6. The main reason for this is that we work from a store semantics contrary to her standard semantics: As was discussed in section 3.1 it is unlikely that Theorem 3.1-6 would hold for any approach based on a standard semantics.

Comparison with [Don78]

The approach of [Don78] differs from the approach of [Don79] in a number of ways.

Sections 1 and 2 of [Don78] define a toy language, which is like ours except that it includes a data structure "tables". The semantics is a standard semantics (not a store semantics) and the association of identifiers to values is by means of a 2-stage mapping, i.e. there are locations. The meaning function $\mathcal{P}$ for programs is of functionality Pro -c> Inp -c> Sta$\times$Out where $\mathcal{P}[\![$pro$]\!]$inp$\downarrow$1 is the final state and $\mathcal{P}[\![$pro$]\!]$inp$\downarrow$2 is the output resulting from "executing" pro with input inp. The domains are partially ordered sets but they are not restricted to being definable in DSL.

Sections 3 and 4 are not relevant for this comparison. Sections 5 and 6 contain a non-standard semantics performing a kind of "value subset analysis". The main difference from [Don79] and our approach is that no notion of occurrences is present. The functionality of $\mathcal{P}$ of this semantics is Pro -c> Inp' -c> Sta'$\times$Out' where Inp', Sta', and Out' specify properties of the input, final state and output. So $\mathcal{P}[\![$pro$]\!]$inp' does not associate (a description of) states with places. The result from this semantics appears less usable than the results specified in our and [Don79]'s approach.

The non-standard interpretation cannot be described satisfactorily in our framework. One reason is the different functionalities of $\mathcal{P}$. Another reason is, that [Don78] works with a 2-stage mapping and locations are not approximated: no representation of sets of locations is considered.

The way that values is approximated can roughly be described by concVa. Some conditions are imposed on concVa; one implies that $\forall$va$\in$Va:$\perp\in$concVa(va). This appears to be needed in the proof of Lemma 20 (a correctness proof).

To some extent the non-standard interpretation bears the same relationship to std as ind does to col. I.e. it is more or less the same development that has been performed. In section 3.1 we mentioned some difficulties in basing sts upon std. One proposal was to define a domain sts-A so that $\perp\in$sts-a for any sts-a$\in$sts-A. Hence $\perp\in$conc(ind-a) for any ind-a$\in$ind-A. This is related to the condition of [Don78] mentioned above.

In summary, [Don78] seems to be a forerunner for [Don79] and less adequate in spite of the wider class of ("history-insensitive") data flow analyses considered.

Comparison with [CoC77a] and [CoC79]

Since we have used the ideas of Cousot&Cousot there is much similarity in approaches. The main differences are the perception of programs and the semantic foundations.

Our view, and that of [Don79], of a program is high-level in the sense of [Ros77]: we view a program as a parse-tree. In contrast, [CoC79], [CoC77a] and the traditional data flow analysis view a program as a graph. Arguments in favour of a high-level view can be found in [Ros77].

In our approach we have worked from a denotational definition of the language. The approach of [CoC77a] and [CoC79] is different: In [CoC77a] a static semantics is developed from an operational semantics. In [CoC79] the static semantics is merely stated (and it is mentioned that a MOP solution is wanted). We believe that a denotational semantics is a more satisfactory starting point than an operational semantics. See [Sto77] for a discussion.

CHAPTER 4

History-Sensitive Analyses

In this chapter we consider "available expressions" analysis as an example of a forward, history-sensitive data flow analysis. This data flow analysis was reviewed in section 2.3. We do not consider other forward data flow analyses.

For lack of time we do not sketch an approach to "live variables", which is a backward "history-sensitive" analysis.

In section 4.1 we consider interpretations that record the computational history. For these history-interpretations we reformulate the general framework of section 3.1. In section 4.2 we develop our notion of available expressions interpretation. This is compared with the literature in section 4.3.

4.1 Reformulating the Framework

According to our approach any data flow interpretation must be related to a static semantics by means of a pair of semi-adjoined functions, as we did for the constant propagation interpretation of section 3.2. When we come to the available expression interpretation (ae) in section 4.2 it seems impossible to do so, when the static semantics is sts. Intuitively, sts contains too little information about the computational history.

In this section we consider interpretations that record the computational history. It is convenient to do so by re-developing the framework of section 3.1, i.e. to specify interpretations his-std, his-col, his-sts and his-ind. To aid the readability of the definitions we introduce the shorthands: When $sta \in Sta$ then sta.env means $sta\downarrow 1$, sta.inp means $sta\downarrow 2$, sta.out means $sta\downarrow 3$ and sta.wit means $sta\downarrow 4$. When $sta^* \in Sta^*$ we let sta^*.last mean $sta^*\downarrow \# sta^*$ and feel free to write e.g. sta^*.last.env meaning $(sta^*\downarrow \# sta^*)\downarrow 1$.

The "Standard" Interpretation

We pattern his-std closely after std. The main difference is that his-std-S is not Sta but Sta^*. The intention is that $sta^* \in Sta^*$ records the sequence of states that have previously been computed. There are more complicated alternatives to using Sta^*, but the his-sts developed using Sta^* is adequate for giving a semantic characterization of ae. Table 4.1-A specifies his-std.

LEMMA 4.1-1: Table 4.1-A specifies an interpretation. □
Proof is similar to that of Lemma 2.2-5. □

To aid the intuition we state:

```
TABLE 4.1-A --- INTERPRETATION his-std
======================================

Domains (of his-std)
--------------------
    c ∈ C = Sta* -c> A
  inp ∈ I = Inp
        A = Out + {"wrong"}
 sta* ∈ S = Sta*

        ( See Table 2.2-B for )
        ( Pla, Inp, Sta ...   )

Constants
---------
  his-std-wrong ∈ C
  his-std-wrong = λsta*. "wrong" inA

  his-std-finish ∈ C
  his-std-finish = λsta*. (sta*.last.out) inA

Pseudo-Semantic Functions
-------------------------
  his-std-𝓑 ∈ Bas° -c> Val
  his-std-𝓑 = std-𝓑

  his-std-𝒪 ∈ Ope° -c> (Val X Val -c> Val)
  his-std-𝒪 = std-𝒪

Auxiliary Functions
-------------------
  his-std-setup ∈ C -c> I -c> A
  his-std-setup c inp =
    c <  <λide."nil" inVal, inp, <>, <>>  >

  his-std-cond ∈ C X C -c> C
    his-std-Vcond ∈ Sta* -c> T
    his-std-Scond ∈ Sta* -c> T
    his-std-Bcond ∈ Sta* -c> Sta*

  his-std-cond(c1,c2) sta* =
    his-std-Vcond(sta*) ->
        [ his-std-Scond(sta*) -> c1, c2 ](his-std-Bcond sta*)
        , his-std-wrong(sta*)
  his-std-Vcond sta* =
    std-Vcond (sta*.last)
  his-std-Scond sta* =
    std-Scond (sta*.last)
  his-std-Bcond sta* =
    sta* § < std-Bcond(sta*.last) >

  his-std-attach ∈ Pla -c> C -c> C
  his-std-attach pla c sta* =
    c(sta*)

Primitive Functions
-------------------
  his-std-g ∈ Par -t> C -c> C
    his-std-Vg ∈ Par -t> Sta* -c> T
    his-std-Bg ∈ Par -t> Sta* -c> Sta*

  his-std-g(par)(c)(sta*) =
    his-std-Vg(par)(sta*) ->
      c(his-std-Bg(par)(sta*)) ,
      his-std-wrong(sta*)
  his-std-Vg(par)(sta*) =
    std-Vg(par)(sta*.last)
  his-std-Bg(par)(sta*) =
    sta* § < std-Bg(par)(sta*.last) >
```

LEMMA 4.1-2: $\forall inp \in Inp\ \forall pro \in Pro$: $\mathcal{P}std[\![pro]\!]inp = \mathcal{P}his\text{-}std[\![pro]\!]inp$. □
Proof is by structural induction and is omitted. □

The above lemma says that his-std and std are "congruent" [Sto77], i.e. when used with the semantic functions they associate the same denotation with a program. This relationship between his-std and std parallels the relationship between the standard, store and stack semantics of [MiS76]. There the formulation of the semantics is gradually transformed to a machine-like formulation. In our case we changed the formulation so as to incorporate more "history"-information.

As will be shown below there are no similar relationship between col and his-col, etc. . Intuitively, this is because we record the "internals" of the semantics in the result of the semantic functions.

The "Collecting" Interpretation

As in section 3.1 we must define what we mean by "the set of state sequences that are possible at pla during an execution of program pro with input inp ". We define this to be $\mathcal{P}his\text{-}col[\![pro]\!](inp)(pla)$, where his-col is defined in Table 4.1-B.

LEMMA 4.1-3: Table 4.1-B specifies an interpretation. □
Proof is similar to that of Lemma 3.1-1. □

Similarly to the case with std and col we cannot characterize $\mathcal{P}his\text{-}col[\![pro]\!]$ in terms of $\mathcal{P}his\text{-}std[\![pro]\!]$. We can, however, relate his-col to col:

LEMMA 4.1-4: $\forall pro\ \forall inp\ \forall pla$:
$\mathcal{P}col[\![pro]\!]inp\ pla = \{sta^{*}.last \mid sta^{*} \in \mathcal{P}his\text{-}col[\![pro]\!]inp\ pla\}$ □
Proof is as the proof of Lemma 4.1-2. □

Interpretation his-col has been obtained from his-std in the same way as col was obtained from std. An alternative is to obtain his-col by modifying col. But our choice emphasizes the systematic nature of the development. - Similar remarks apply to his-sts and his-ind below.

The "Static" Interpretation

We then develop a static semantics by defining his-sts (Table 4.1-C). As in section 3.1 his-sts is fundamental for abstract interpretation to be applicable. It is also with respect to his-sts that we give a semantic characterization of the ae-interpretation.

LEMMA 4.1-5: Table 4.1-C specifies an approximate interpretation. □
Proof is similar to those of 3.1-2 and 3.1-8. □

The connection between his-sts and his-col is of course similar to the connection between sts and col:

LEMMA 4.1-6: If $\forall inp \in \underline{inp}$: topfree[Inp](inp) then
$\mathcal{P}his\text{-}sts[\![pro]\!]\underline{inp} = \sqcup\{\mathcal{P}his\text{-}col[\![pro]\!]\ inp \mid inp \in \underline{inp}\}$ □
Proof is shown in appendix 4. □

Lemmas 4.1-6, 4.1-4 and Theorem 3.1-6 show a certain relationship between his-sts and sts. Another formulation of essentially the same relationship is

TABLE 4.1-B --- INTERPRETATION his-col

```
Domains (of his-col)
--------------------
    c ∈ C = Sta* -t> A
  inp ∈ I = Inp
    a ∈ A = Pla -c> ℘(Sta*)
 sta* ∈ S = Sta*

Constants
---------
  his-col-wrong ∈ C
  his-col-wrong = ⊥

  his-col-finish ∈ C
  his-col-finish = ⊥

Pseudo-Semantic Functions
-------------------------
  his-col-ℬ ∈ Bas° -c> Val
  his-col-ℬ = std-ℬ

  his-col-𝒪 ∈ Ope° -c> (Val X Val -c> Val)
  his-col-𝒪 = std-𝒪

Auxiliary Functions
-------------------
  his-col-setup ∈ C -c> I -t> A
  his-col-setup = his-std-setup

  his-col-cond ∈ C X C -c> C
    his-col-Vcond ∈ Sta* -c> T
    his-col-Scond ∈ Sta* -c> T
    his-col-Bcond ∈ Sta* -c> Sta*

  his-col-cond = his-std-cond
    his-col-Vcond = his-std-Vcond
    his-col-Scond = his-std-Scond
    his-col-Bcond = his-std-Bcond

  his-col-attach ∈ Pla -c> C -c> C
  his-col-attach pla c sta* =
    (c sta*) ⊔ ⊥[ {sta*} / pla ]

Primitive Functions
-------------------
  his-col-g ∈ Par -t> C -c> C
    his-col-Vg ∈ Par -t> Sta* -c> T
    his-col-Bg ∈ Par -t> Sta* -c> Sta*

  his-col-g = his-std-g
    his-col-Vg = his-std-Vg
    his-col-Bg = his-std-Bg
```

Lemma 4.1-10 below.

<u>The "Induced" Interpretation</u>

As in section 3.1 it is possible to consider an induced interpretation. Let S be a complete lattice and <uabs,conc> a pair of semi-adjoined functions between ℘(Sta*) and S. Then his-ind (or his-ind<uabs,conc>) of Table 4.1-D is said to be induced from his-sts by <uabs,conc>.

LEMMA 4.1-7: Table 4.1-D specifies an approximate interpretation. []
Proof is similar to the proof of Lemma 3.1-11. []

```
TABLE 4.1-C --- INTERPRETATION his-sts
======================================

Domains (of his-sts)
--------------------
    c ∈ C = 𝒫(Sta*) -c> A
  inp ∈ I = 𝒫(Inp)
    a ∈ A = Pla -c> 𝒫(Sta*)
    s ∈ S = 𝒫(Sta*)

Constants
---------
  his-sts-wrong ∈ C
  his-sts-wrong = ⊥

  his-sts-finish ∈ C
  his-sts-finish = ⊥

Pseudo-Semantic Functions
-------------------------
  his-sts-ℬ ∈ Bas° -c> Val
  his-sts-ℬ = std-ℬ

  his-sts-𝒪 ∈ Ope° -c> (Val X Val -c> Val)
  his-sts-𝒪 = std-𝒪

Auxiliary Functions
-------------------
  his-sts-setup ∈ C -c> I -c> A
  his-sts-setup c inp =
    c { < <λide."nil" inVal, inp, <>, <>> > | inp∈inp}

  his-sts-cond ∈ C X C -c> C
    his-sts-Dt-cond ∈ S -c> S
    his-sts-Df-cond ∈ S -c> S
  his-sts-cond(c1,c2) s =
    c1(his-sts-Dt-cond(s)) ⊔ c2(his-sts-Df-cond(s))
  his-sts-Dt-cond s =
    { his-std-Bcond(sta*) | sta* ∈ s ∧
           his-std-Vcond(sta*) = true  ∧
           his-std-Scond(sta*) = true  }
  his-sts-Df-cond s =
    { his-std-Bcond(sta*) | sta* ∈ s ∧
           his-std-Vcond(sta*) = true  ∧
           his-std-Scond(sta*) = false }

  his-sts-attach ∈ Pla -c> C -c> C
  his-sts-attach(pla)(c)(s) =
    (c s) ⊔ ⊥[s/pla]

Primitive Functions
-------------------
  his-sts-g ∈ Par -t> C -c> C
    his-sts-Dg ∈ Par -t> S -c> S
  his-sts-g(par)(c)(s) =
    c( his-sts-Dg(par)(s))
  his-sts-Dg(par)(s) =
    { his-std-Bg(par)(sta*) | sta* ∈ s ∧
           his-std-Vg(par)(sta*) = true }
```

LEMMA 4.1-8: his-sts ⊑<uabs,conc> (his-ind<uabs,conc>) □

Proof is similar to the proof of Lemma 3.1-12. □

The following shows that the static interpretation (sts) can be seen as an induced interpretation of the static "history" interpretation (his-sts). Define uabs:𝒫(Sta*)->𝒫(Sta) by uabs(s) = {sta*.last |sta*∈s} and conc:𝒫(Sta)->𝒫(Sta*) by conc(sta) = {sta* |sta*.last∈sta}.

```
TABLE 4.1-D --- INTERPRETATION his-ind
======================================

Domains (of his-ind)
--------------------
    c ∈ C = S -i> A
  inp ∈ I = 𝒫(Inp)
    a ∈ A = Pla -c> S
    s ∈ S   [ uabs:𝒫(Sta*)->S ]
            [ conc:S->𝒫(Sta*) ]

Constants
---------
  his-ind-wrong ∈ C
  his-ind-wrong = ⊥

  his-ind-finish ∈ C
  his-ind-finish = ⊥

Pseudo-Semantic Functions
-------------------------
  his-ind-𝓑 ∈ Bas° -c> Val
  his-ind-𝓑 = std-𝓑

  his-ind-𝒪 ∈ Ope° -c> (Val X Val -c> Val)
  his-ind-𝒪 = std-𝒪

Auxiliary Functions
-------------------
  his-ind-setup ∈ C -c> I -i> A
  his-ind-setup c inp =
    c( uabs{< <λide."nil" inVal, inp, <>, <>> > |inp∈inp} )

  his-ind-cond ∈ C X C -c> C
    his-ind-Dt-cond ∈ S -i> S
    his-ind-Df-cond ∈ S -i> S
  his-ind-cond(c1,c2) s =
    c1(his-ind-Dt-cond(s)) ⊔ c2(his-ind-Df-cond(s))
  his-ind-Dt-cond(s) =
    uabs( his-sts-Dt-cond( conc(s)))
  his-ind-Df-cond(s) =
    uabs( his-sts-Df-cond( conc(s)))

  his-ind-attach ∈ Pla -c> C -c> C
  his-ind-attach pla c s =
    (c s) ⊔ (⊥[s/pla])

Primitive Functions
-------------------
  his-ind-g ∈ Par -t> C -c> C
    his-ind-Dg ∈ Par -t> S -i> S

  his-ind-g par c s =
    c( his-ind-Dg(par)(s))
  his-ind-Dg par s =
    uabs( his-sts-Dg(par)( conc(s)))
```

LEMMA 4.1-9: <uabs,conc> is an exact pair of adjoined functions. []
Proof is shown in appendix 4. []

LEMMA 4.1-10: his-ind<uabs,conc> equals sts
[and sts ⊑<λsta.sta,λsta.sta> (his-ind<uabs,conc>)
and (his-ind<uabs,conc>) ⊑<λsta.sta,λsta.sta> sts] []
Proof is shown in appendix 4. []

We believe that the above lemma very nicely shows the power of the the method of Abstract Interpretation. We dispense with a formal definition of when two interpretations are equal.

4.2 Available Expressions

In this section we develop an "available expressions" analysis in our framework. In subsection 4.2.1 we perform continuation removal ([Sto77], [MiS76]) for expressions. Thereby it is possible to denote the value computed by some expression. This is used in subsection 4.2.2 where we develop pairs of adjoined functions that relate sets of sequences of states to sets of available expressions. In subsection 4.2.3 we formulate interpretation ae and it is given a semantic characterization in terms of his-sts.

4.2.1 Continuation Removal

Intuitively, a semantic characterization of "exp is available at pla" amounts to relating the value of exp when "evaluated at pla" to its value when previously evaluated. It is therefore necessary to be able to denote the value to which exp evaluates. The functionality of ℰ makes ℰstd⟦exp⟧ inadequate for this. Instead we define the continuation removed function Pℰstd⟦exp⟧ such that ℰstd⟦exp⟧(occ)(c)(sta) = c(sta') whenever Pℰstd⟦exp⟧(occ)(sta) =...sta'... .

For technical reasons it is convenient later in this section to use a more or less similar function Pℰhis-sts.

In Table 4.2-A we have defined the function Pℰ that will be used with both std and his-sts.

TABLE 4.2-A --- CONTINUATION REMOVAL FOR ℰ

Semantic Functions

```
Pℰ ∈ Exp -t> Occ -c> S -c> R

    Pℰ⟦ exp1 ope exp2 ⟧ occ =
        Pattach<occ,"exp)"> *
        Papply⟦ope⟧ *
        Pℰ⟦exp2⟧ occ§<3> *
        Pℰ⟦exp1⟧ occ§<1> *
        Pattach<occ,"(exp">

    Pℰ⟦ ide ⟧ occ =
        Pattach<occ,"exp)"> *
        Pcontent⟦ide⟧ *
        Pattach<occ,"(exp">

    Pℰ⟦ bas ⟧ occ =
        Pattach<occ,"exp)"> *
        Ppush⟦bas⟧ *
        Pattach<occ,"(exp">
```

To be able to use Pℰ with std we augment std as shown in Table 4.2-B. Here we define domain std-R and combinators std-* and std-⊕ as well as functions std-Pattach and std-Pg (for g any primitive function) {a}. Similarly, Table 4.2-D contains the augmentations to his-sts {b}.

Both std-* and his-sts-* (and sts-*) are associative so no parentheses are

---

{a} Pcond is not used until chapter 6.

{b} Table 4.2-C contains augmentations to sts. But this will not be used until chapter 6.

needed in the definition of Pℰ in Table 4.2-A:

LEMMA 4.2.1-1: Tables 4.2-A, 2.2-D and 4.2-B define Pℰstd that is of functionality as shown. Similarly, Tables 4.2-A, 4.1-C and 4.2-D define Pℰhis-sts that is of functionality as shown. In the definitions of Pℰstd and Pℰhis-sts no function is supplied with an argument of the wrong type. []

The intention with combinators * and ⊕ is that c⊕(ss1*ss2) = (c⊕ss1)⊕ss2. We often omit prefixes std- and his-sts- when it may be deduced from context which * or ⊕ is meant. Hence the relationship between ℰ and Pℰ can be stated as:

LEMMA 4.2.1-2: ℰstd⟦exp⟧occ c = c ⊕ Pℰstd⟦exp⟧occ and
ℰhis-sts⟦exp⟧occ c = c ⊕ Pℰhis-sts⟦exp⟧occ. []
Proof is shown in appendix 4. []

```
TABLE 4.2-B --- ADDITIONS TO std
================================

Domains (of std)
----------------
  S = Sta
  R = Sta + {"wrong"}

Combinators (of std)
--------------------
  * ∈ [S -c> R] X [S -c> R] -c> [S -c> R]
  ss1 * ss2 = λs. ss2(s) Ɛ Sta ->
                  ss1(ss2(s) |Sta), ss2(s)

  ⊕ ∈ C X [S -c> R] -c> C
  c ⊕ ss = λs. ss(s) Ɛ Sta ->
               c( ss(s) |Sta), "wrong" inA

Auxiliary Functions
-------------------
  std-Pcond ∈ [S -c> R] X [S -c> R] -c> [S -c> R]
  std-Pcond(ss1,ss2) s =
    std-Vcond(s) -> [ std-Scond(s) -> ss1, ss2 ](std-Bcond s)
                  , "wrong" inR

  std-Pattach ∈ Pla -c> S -c> R
  std-Pattach(pla)(s) =
    s inR

Primitive Functions
-------------------
  std-Pg ∈ Par -t> S -c> R
  std-Pg(par)(s) =
    std-Vg(par)(s) -> std-Bg(par)(s) inR
                    , "wrong" inR
```

We need the above result for his-sts in the proof of Lemma 4.2.3-5. The result for std supports the intuition that Pℰstd⟦exp⟧ describes the effect of exp. In the proof of Lemma 4.2.3-5 we will also need the following two lemmas:

LEMMA 4.2.1-3: For any expression exp and state <env,inp,out,wit> and occurrence occ:
wit∉{⊥,⊤} => [∃val∈Val:
Pℰstd⟦exp⟧occ<env,inp,out,wit> = <env,inp,out,<val>§wit> inR] []
Proof is by structural induction. []

```
TABLE 4.2-C --- ADDITIONS TO sts
================================

Domains (of sts)
----------------
   A = Pla -c> P(Sta)
   S = P(Sta)
   R = S X A

Combinators (of sts)
--------------------
  * ∈ [S -c> R] X [S -c> R] -c> [S -c> R]
  ss1 * ss2 = λs. < ss1(ss2(s)↓1)↓1
                  , ss1(ss2(s)↓1)↓2 ⊔ ss2(s)↓2 >

  ⊕ ∈ C X [S -c> R] -c> C
  c ⊕ ss = λs. c(ss(s)↓1) ⊔ (ss(s)↓2)

Auxiliary Functions
-------------------

  sts-Pcond ∈ [S -c> R] X [S -c> R] -c> [S -c> R]
  sts-Pcond(ss1,ss2) sta =
    ss1(sts-Dt-cond(sta)) ⊔ ss2(sts-Df-cond(sta))

  sts-Pattach ∈ Pla -c> S -c> R
  sts-Pattach pla sta =
    <sta, ⊥[sta/pla]>

Primitive Functions
-------------------
  sts-Pg ∈ Par -t> S -c> R
  sts-Pg par sta =
    < sts-Dg(par)(sta), ⊥ >
```

LEMMA 4.2.1-4: ∀s∈𝒫(Sta*): If ∀sta*∈s: sta*.last.wit∉{⊥,⊤} then
{sta*.last | sta*∈(P𝒞his-sts⟦exp⟧occ s)↓1}
= {P𝒞std⟦exp⟧occ(sta*.last)|Sta | sta*∈s} []
Proof is shown in appendix 4. []

It is easy to show that P𝒞std⟦exp⟧occ is independent of occ. We therefore take the notational liberty of eliding occ in the sequel, because it complicates the exposition to state "where exp and occ are such that exp = pro <u>at</u> occ" whenever P𝒞std⟦exp⟧(occ) is used.

Our notion of continuation removal is similar to that of [Sto77] and [MiS76] although the details and motivations differ.

<u>4.2.2 Pairs of adjoined functions for available expressions</u>

We now informally explain a semantic notion of available expressions. Suppose pro is a program and pla a place such that pla designates a point immediately preceding an evaluation of expression exp. Intuitively, (the value of) this expression is available at pla iff in every computation some expression (e.g. exp itself) has previously been evaluated to the same value to which exp now evaluates. More formally this means (by Lemma 4.2.1-2 and 4.2.1-3 and by tacitly ignoring some special cases) that for any input inp and sta*∈𝒫his-col⟦pro⟧(inp)(pla) there is an i∈{1,...,#sta*} such that (sta*↓i).wit↓1 equals (P𝒞std⟦exp⟧(sta*.last)|Sta).wit↓1. This notion is semantic, in contrast to the traditional definition of available expressions,

```
TABLE 4.2-D --- ADDITIONS TO his-sts
=====================================

Domains (of his-sts)
--------------------
   A = Pla -c> ℘(Sta*)
   S = ℘(Sta*)
   R = S X A

Combinators (of his-sts)
------------------------
  * ∈ [S -c> R] X [S -c> R] -c> [S -c> R]
  ss1 * ss2 = λs.< ss1(ss2(s)↓1)↓1
                 , ss1(ss2(s)↓1)↓2 ⊔ ss2(s)↓2 >

  ⊕ ∈ C X [S -c> R] -c> C
  c ⊕ ss = λs. c(ss(s)↓1) ⊔ (ss(s)↓2)

Auxiliary Functions
-------------------
  his-sts-Pattach ∈ Pla -c> S -c> R
  his-sts-Pattach pla s =
    < s, ⊥[s/pla] >

Primitive Functions
-------------------
  his-sts-Pg ∈ Par -t> S -c> R
  his-sts-Pg par s =
    < his-sts-Dg par s, ⊥ >
```

which is syntactic: the manipulation of sets of expressions is given no semantic characterization.

In this subsection we formalize the above description of (semantically) available expressions. In subsection 4.2.3 we formulate an interpretation (ae) that computes available expressions according to the usual (syntactic) definition. The purpose of the present subsection is to define a pair of adjoined functions such that ae can be given a semantic characterization (in subsection 4.2.3). This amounts to giving a semantic characterization of an apparently syntactic [CoC77a,p.242] data flow analysis.

Recall that $\mathcal{P}'(\ldots)$ is dual to $\mathcal{P}(\ldots)$, i.e. with $\supseteq$ as ordering (2.1.3-10):

DEFINITION 4.2.2-1: Let $r:Sta^* \to \mathcal{P}'(Exp)$ be any function. Define uabs-r$:\mathcal{P}(Sta^*) \to \mathcal{P}'(Exp)$ and conc-r$:\mathcal{P}'(Exp) \to \mathcal{P}(Sta^*)$ by uabs-r$(s) = \bigcap\{r(sta^*) \mid sta^* \in s\}$ and conc-r$(\underline{exp}) = \{sta^* \mid r(sta^*) \supseteq \underline{exp}\}$ []

LEMMA 4.2.2-2: <uabs-r,conc-r> is a pair of adjoined functions. []

The intention with $r(sta^*)$ is that it is a set of expressions exp such that there is an $i \in \{1,\ldots,\#sta^*\}$ so that $(sta^* \downarrow i).wit\downarrow 1$ equals $(P\tilde{?}std[\![exp]\!](sta^*.last)|Sta).wit\downarrow 1$. To make this precise we need:

DEFINITION 4.2.2-3: Define sametop: (Sta+{"wrong"}) X (Sta+{"wrong"}) -> B by sametop(a1,a2) =

$(a1 \mathrel{E} Sta)=true \wedge (a2 \mathrel{E} Sta)=true \wedge$
$\#((a1|Sta).wit) \in \{1,2,\ldots\} \wedge \#((a2|Sta).wit) \in \{1,2,\ldots\} \wedge$
$(a1|Sta).wit\downarrow 1 = (a2|Sta).wit\downarrow 1$ []

DEFINITION 4.2.2-4: Define $r1:Sta^* \to \mathcal{P}'(Exp)$ to be doubly strict so that $sta^* \notin \{\perp,\top\}$ implies $r1(sta^*) = \{exp \mid \exists j \in \{1,\ldots,\#sta^*\}$:

$\text{sametop}(\text{sta}^*\!\downarrow\! j \ \text{in}(\text{Sta}+\{\text{"wrong"}\}),\ \text{P}\mathscr{E}\text{std}[\![\text{exp}]\!](\text{sta}^*.\text{last}))\}$ □

Function r1 formalizes a semantic notion of availability: his-ind induced by <uabs-r1,conc-r1> computes (semantically) available expressions. We use $\mathcal{P}'(\text{Exp})$ rather than $\mathcal{P}(\text{Exp})$ because only then does $\cap$ correspond to $\sqcup$.

We now set forth to develop the function r3 that will be used in subsection 4.2.3 to show the correctness of ae. As an intermediate step we develop r2:

DEFINITION 4.2.2-5: Define $\text{r2}:\text{Sta}^*\to\mathcal{P}'(\text{Exp})$ to be doubly strict so that $\text{sta}^*\notin\{\bot,\top\}$ implies $\text{r2}(\text{sta}^*) =$

$$\{\text{exp}\mid\exists j\in\{1,\ldots,\#\text{sta}^*\}:\ \text{gen2}(\text{sta}^*\!\downarrow\! j,\text{exp})\ \wedge$$
$$\forall k\in\{j+1,\ldots,\#\text{sta}^*\}:\ \text{pre2}(\text{sta}^*\!\downarrow\!(k-1),\text{sta}^*\!\downarrow\! k,\text{exp})\}$$

where $\text{gen2}(\text{sta},\text{exp}) = \text{sametop}(\text{sta in}(\text{Sta}+\{\text{"wrong"}\}),\ \text{P}\mathscr{E}\text{std}[\![\text{exp}]\!]\text{sta})$
and $\text{pre2}(\text{sta1},\text{sta2},\text{exp}) = \text{sametop}(\text{P}\mathscr{E}\text{std}[\![\text{exp}]\!]\text{sta1},\ \text{P}\mathscr{E}\text{std}[\![\text{exp}]\!]\text{sta2})$. □

The intention is that gen2 is true for those expressions that have been semantically "generated", and that pre2 is true for the expressions that have been semantically "preserved".

OBSERVATION 4.2.2-6: $\text{uabs-r1} \sqsubseteq \text{uabs-r2}$ □

An important feature of r2 (which is shared by r3 below) is that its definition can be expressed inductively. This is adequate when <uabs-r3,conc-r3> is used to give a semantic characterization of ae. For the inductive definition (assuming $\text{sta}^*\notin\{\bot,\top\}$):

$$\text{r2}(\text{sta}^*\S\langle\text{sta}\rangle)$$
$$= \{\text{exp}\mid\exists j\in\{1,\ldots,\#\text{sta}^*+1\}:\ \text{gen2}((\text{sta}^*\S\langle\text{sta}\rangle)\!\downarrow\! j,\text{exp})\ \wedge$$
$$\forall k\in\{j+1,\ldots,\#\text{sta}^*+1\}:\ \text{pre2}((\text{sta}^*\S\langle\text{sta}\rangle)\!\downarrow\!(k-1),(\text{sta}^*\S\langle\text{sta}\rangle)\!\downarrow\! k,\text{exp})\}$$
$$= \{\text{exp}\mid[\text{exp}\in\text{r2}(\text{sta}^*)\wedge\text{pre2}(\text{sta}^*.\text{last},\text{sta},\text{exp})]\vee\text{gen2}(\text{sta},\text{exp})\}$$
$$= [\text{r2}(\text{sta}^*)\cap\{\text{exp}\mid\text{pre2}(\text{sta}^*.\text{last},\text{sta},\text{exp})\}]\cup\{\text{exp}\mid\text{gen2}(\text{sta},\text{exp})\}$$

Note the resemblance of the above definition to the transfer function for available expressions (section 2.3).

In the description of available expressions (section 2.3) we cited from [Ros79]: "...where an expression is considered to have changed in value whenever any one of the [identifiers] involved in it has changed". To capture this notion we define r3.

DEFINITION 4.2.2-7: Define $\mathcal{I}:\text{Exp}\to\mathcal{P}(\text{Ide})$ by
$\mathcal{I}[\![\text{exp1 ope exp2}]\!] = \mathcal{I}[\![\text{exp1}]\!]\cup\mathcal{I}[\![\text{exp2}]\!]$
$\mathcal{I}[\![\text{ide}]\!] = \{\text{ide}\}$
$\mathcal{I}[\![\text{bas}]\!] = \emptyset$ □

DEFINITION 4.2.2-8: Define $\text{r3}:\text{Sta}^*\to\mathcal{P}'(\text{Exp})$ to be doubly strict so that $\text{sta}^*\notin\{\bot,\top\}$ implies $\text{r3}(\text{sta}^*) =$

$$\{\text{exp}\mid\exists j\in\{1,\ldots,\#\text{sta}^*\}:\ \text{gen3}(\text{sta}^*\!\downarrow\! j,\text{exp})\wedge$$
$$\forall k\in\{j+1,\ldots,\#\text{sta}^*\}:\ \text{pre3}(\text{sta}^*\!\downarrow\!(k-1),\text{sta}^*\!\downarrow\! k,\text{exp})$$

where $\text{gen3}(\text{sta},\text{exp}) = \text{gen2}(\text{sta},\text{exp})$ and
$\text{pre3}(\text{sta1},\text{sta2},\text{exp}) = (\text{sta1.wit}\notin\{\bot,\top\})\wedge(\text{sta2.wit}\notin\{\bot,\top\})\wedge$
$\forall\text{ide}\in\mathcal{I}[\![\text{exp}]\!]:\ \text{sta1.env}[\![\text{ide}]\!] = \text{sta2.env}[\![\text{ide}]\!]$ □

Of course also r3 can be defined inductively: $\text{r3}(\text{sta}^*\S\langle\text{sta}\rangle) =$
$[\ \text{r3}(\text{sta}^*)\cap\{\text{exp}\mid\text{pre3}(\text{sta}^*.\text{last},\text{sta},\text{exp})\}\ ]\cup\{\text{exp}\mid\text{gen3}(\text{sta},\text{exp})\}$
where we have assumed $\text{sta}^*\notin\{\bot,\top\}$.

LEMMA 4.2.2-9: uabs-r2 ⊑ uabs-r3 []
Proof is shown in appendix 4. []

For later usage it is convenient to extract the following lemma used in the proof of Lemma 4.2.2-9:

LEMMA 4.2.2-10: ∀sta1,sta2∈Sta ∀exp∈Exp:
pre3(sta1,sta2,exp) => pre2(sta1,sta2,exp) []
Proof is shown in appendix 4. []

4.2.3 Available Expression Interpretation

Table 4.2-E contains the ae-interpretation. It has S = $\mathcal{P}$(Exp) and <u>exp</u> is intended to describe a set of available expressions. We explain the primitive functions below. Combinators * and ⊕ and functions Pattach and Pg are used when we perform continuation removal for ae (Lemma 4.2.3-4).

The effect of g upon <u>exp</u> is described by a transfer function (λ<u>exp</u>.GENg(par) ∪ [PREg(par) ∩ <u>exp</u>]) applied to <u>exp</u>. Here GENg(par) is the set of expressions generated and PREg(par) is the set of expressions preserved. Mostly PREg(par) = Exp and GENg(par) = ∅.

One exception is that ∀ide∈Ide: PREassign⟦ide⟧ = {exp |ide∉$\mathcal{I}$⟦exp⟧}. That is, we only preserve the expressions that we know to be unaffected by the value of ide.

The other exception is that ∀exp∈Exp: GENapply⟦ope⟧⟦exp⟧ = {exp}. We expect any application apply⟦ope⟧ to generate the expression exp = exp1 ope exp2 "corresponding to" operator ope in the parse-tree. What this expression is cannot be deduced from ope. The solution we have chosen is to supply ae-apply and GENapply with an extra parameter being that expression. To do so the semantic function $\mathcal{E}$ (and P$\mathcal{E}$) must be changed as shown in Table 4.2-F.

We would like to know that ae is an interpretation because then Theorem 2.2-4 assures that ae can be used together with the semantic functions of Table 2.2-C. However, ae-apply has an extra parameter (in Exp), so strictly speaking ae is not an interpretation; further, Table 4.2-F modifies Table 2.2-C so that the proof of Theorem 2.2-4 no longer pertains. It is straight-forward, but tedious, to be precise and correct these minor deviations. We choose to ignore these issues and state:

LEMMA 4.2.3-1: Table 4.2-E specifies an approximate interpretation. []

Also note that the use of a continuation style semantics is advantageous to a direct style semantics because there is an "implicit ordering" upon the evaluation of subexpressions. This is not necessarily the case for a direct style semantics. Even when subexpressions "have no side-effects" this is desirable for "available expressions" analysis.

Semantic Characterization of ae

We then give a semantic characterization of ae. Because of the extra parameter to apply we cannot show his-sts ⊑<uabs-r3,conc-r3> ae. Instead we prove:

THEOREM 4.2.3-2: ∀pro ∀<u>inp</u>: $\mathcal{P}$his-sts⟦pro⟧<u>inp</u> ⊑ (R conc-r3)($\mathcal{P}$ae⟦pro⟧<u>inp</u>) []
Proof is shown in appendix 4. []

```
TABLE 4.2-E(a) --- INTERPRETATION ae
======================================

Domains (of ae)
---------------
   c ∈ C = S -c> A
 inp ∈ I = 𝒫(Inp)
   a ∈ A = Pla -c> S
 exp ∈ S = 𝒫'(Exp)
       R = S X A

Constants
---------
  ae-wrong ∈ C
  ae-wrong = ⊥ = λexp.λpla.Exp

  ae-finish ∈ C
  ae-finish = ⊥ = λexpλpla.Exp

Combinators (of ae)
-------------------
  * ∈ [S -c> R] X [S -c> R] -c> [S -c> R]
  ss1 * ss2 = λexp.< ss1(ss2(exp)↓1)↓1
                   , ss1(ss2(exp)↓1)↓2 ⊔ ss2(exp)↓2 >

  ⊕ ∈ C X [S -c> R] -c> C
  c ⊕ ss = λexp. c(ss(exp)↓1) ⊔ (ss exp)↓2

Pseudo-Semantic Functions
-------------------------
  ae-𝓑 ∈ Bas° -c> Val
  ae-𝓑 = std-𝓑

  ae-𝒪 ∈ Ope° -c> (Val X Val -c> Val)
  ae-𝒪 = std-𝒪

Auxiliary Functions
-------------------
  ae-setup ∈ C -c> I -c> A
  ae-setup c inp =
    c ∅

  ae-cond ∈ C X C -c> C
    ae-Dt-cond ∈ S -c> S
    ae-Df-cond ∈ S -c> S

  ae-cond(c1,c2) exp =
    c1(ae-Dt-cond(exp)) ⊔ c2(ae-Df-cond(exp))
  ae-Dt-cond(exp) =
    exp
  ae-Df-cond(exp) =
    exp

  ae-attach ∈ Pla -c> C -c> C
    ae-Pattach ∈ Pla -c> S -c> R
  ae-attach(pla)(c)(exp) =
    (c exp) ⊔ ⊥[exp/pla]
  ae-Pattach(pla)(exp) =
    <exp, ⊥[exp/pla]>
```

To conduct this proof it is convenient to perform continuation removal for $\mathcal{E}$ae, thus defining P$\mathcal{E}$ae. As a companion to Lemmas 4.2.1-1 and 4.2.1-2 we have:

LEMMA 4.2.3-3: Tables 4.2-A (modified by Table 4.2-F) and 4.2-E define a function P$\mathcal{E}$ae. It is of functionality as shown and no functions is supplied with an argument of the wrong type. []

```
TABLE 4.2-E(b) --- INTERPRETATION ae
====================================

Primitive Functions
-------------------
  ae-g ∈ Par -t> C -c> C
    ae-Dg ∈ Par -t> S -c> S
    ae-Pg ∈ Par -t> S -c> R
    PREg ∈ Par -t> S
    GENg ∈ Par -t> S
  ae-g par c exp =
    c( ae-Dg(par)(exp))
  ae-Dg par exp =
    GENg(par) ∪ [ PREg(par) ∩ exp ]
  ae-Pg par exp =
    < ae-Dg(par)(exp), ⊥ >

  PREapply⟦ope⟧⟦exp⟧ = Exp
  GENapply⟦ope⟧⟦exp⟧ = {exp}

  PREassign⟦ide⟧ = {exp | ide∉𝓕⟦exp⟧ }
  GENassign⟦ide⟧ = ∅

  PREcontent⟦ide⟧ = Exp
  GENcontent⟦ide⟧ = ∅

  PREpush⟦bas⟧ = Exp
  GENpush⟦bas⟧ = ∅

  PREread = Exp
  GENread = ∅

  PREwrite = Exp
  GENwrite = ∅
```

```
TABLE 4.2-F --- CHANGES TO 𝒞ae AND P𝒞ae
=======================================

𝒞⟦ exp1 ope exp2 ⟧ occ c =
     attach<occ,"(exp">;
     𝒞⟦exp1⟧ occ§<1>;
     𝒞⟦exp2⟧ occ§<3>;
     apply⟦ope⟧⟦exp1 ope exp2⟧;
     attach<occ,"exp)">;c

P𝒞⟦ exp1 ope exp2 ⟧ occ =
     Pattach<occ,"exp)"> *
     Papply⟦ope⟧⟦exp1 ope exp2⟧ *
     P𝒞⟦exp2⟧ occ§<3> *
     P𝒞⟦exp1⟧ occ§<1> *
     Pattach<occ,"(exp">
```

LEMMA 4.2.3-4: 𝒞ae⟦exp⟧occ c = c ⊕ P𝒞ae⟦exp⟧occ [ ]
Proof is similar to that of Lemma 4.2.1-2. [ ]

We need the following predicates:
P-S:(his-sts-S X ae-S) -> B defined by P-S(his-sts-s,ae-s) =
his-sts-s ⊑ conc-r3(ae-s) ∧ ∀sta*∈his-sts-s: sta*.last.wit∉{⊥,⊤}
P-C:(his-sts-C X ae-C) -> B defined by P-C(his-sts-c,ae-c) =
P-S(his-sts-s,ae-s) => his-sts-c(his-sts-s) ⊑ (R conc-r3)(ae-c(ae-s))
The key ingredient in the proof of Theorem 4.2.3-2 is

LEMMA 4.2.3-5: P-C(his-sts-c,ae-c) =>
P-C(𝒞his-sts⟦exp⟧occ his-sts-c, 𝒞ae⟦exp⟧occ ae-c) [ ]
Proof is shown in appendix 4. [ ]

LEMMA 4.2.3-6: For any primitive funcion g (except apply) we have
P-C(his-sts-c,ae-c) => P-C(his-sts-g(par)(his-sts-c), ae-g(par)(ae-c)) []
Proof is shown in appendix 4. []

Since <uabs-r3,conc-r3> is a pair of adjoined functions and uabs-r1 $\sqsubseteq$ uabs-r2 $\sqsubseteq$ uabs-r3 a corollary to Theorem 4.2.3-2 is:

COROLLARY 4.2.3-7: $\forall$pro $\forall$inp:
(R uabs-r1) ($\mathcal{P}$his-sts$[\![$pro$]\!]$inp) $\sqsubseteq$
(R uabs-r2) ($\mathcal{P}$his-sts$[\![$pro$]\!]$inp) $\sqsubseteq$
(R uabs-r3) ($\mathcal{P}$his-sts$[\![$pro$]\!]$inp) $\sqsubseteq$
$\mathcal{P}$ae$[\![$pro$]\!]$inp []

4.3 Comparison with other Approaches

In this section we compare our development of available expressions with the literature. We compare it with the traditional formulation of available expressions, with the common subexpression elimination of [Don79], and with work by Cousot&Cousot [CoC77a],[CoC79] and Wegbreit [Weg75].

Traditional formulation of available expressions

Our ae-interpretation computes available expressions by means of a transfer function of the form λexp.generated $\cup$ (preserved $\cap$ exp), which is as in the traditional formulation of available expressions. This indicates that the concepts underlying our formulation of available expressions closely match the traditional concepts.
This must be complemented by the results of chapter 5 which show a connection between the "solutions".
The semantic characterization of available expressions is not found in the traditional method. Also, our approach unites local and global analysis.

"Common subexpression elimination" of [Don79]

In section 7.2 of [Don79] Donzeau-Gouge applies her framework to the "common subexpression elimination" of [Kil73]. This data flow analysis is different from available expressions, but there is some similarity in purpose: "determination of common subexpressions ... includes the determination of available expressions..."[Don79].
Global common subexpression elimination [Kil73] associates partitioned sets of expressions with arcs. For any two expressions to occur in the same partition the following must hold: It must be known that the values they evaluated to at their latest computation are the same; and after their latest computation their values must not have changed. See [Kil73] for details and how to compute this information.

In the discussion of [Don79] below we use our notation. When performing common subexpression elimination in the framework of [Don79] it is natural to map a place (<occ,"exp)">) to the set of expressions that are in the same partition (at this place) as pro at occ. We believe that the non-standard semantics of [Don79,section 7.2] does perform this task. This belief is supported by (parts of) the motivation presented in [Don79]. But [Don79], probably by mistake, does not prove this. Her theorem asserts something different, but (the proof of) Lemma 8 appears to be in error [Nie80]. The possible remedies do not appear to be immediate.

The non-standard semantics of [Don79,7.2] does not use the concepts "generate" or "preserve", nor does it compute common subexpressions by a method akin to that of [Kil73]. Instead she gives a Herbrand interpretation: the state maps expressions of the program to symbolic trees. The place <occ,"exp)"> is mapped to the set of expressions such that the current state maps each expression (of that set) to the symbolic tree that pro at occ is mapped to.

Because [Don79] uses concepts different from those of [Kil73] it is not obvious how the concepts are related. This is contrary to our ae that does use the traditional concepts. We therefore feel that our development gives a semantic characterization better than [Don79,section7.2] of the traditional data flow analysis which it intends to model.

Comparison with Cousot&Cousot and Wegbreit

Both [CoC77a,p.241] and [CoC79,p.278] formulate abstract interpretations for "available expressions". Also [Weg75,p.276] formulates "available expressions". All three formulations use the notions of "preserve" and "generate".

None of the three papers give a semantic characterization of the formulation. Wegbreit [Weg75] does not even mention the issue. In [CoC77a,p.242] there is the following discussion: "One can distinguish between syntactic and semantic abstract interpretations of a program. Syntactic interpretations are proved to be correct by reference to the program syntax (e.g. ... available expressions ...). By contrast semantic abstract interpretations must be proved to be consistent with the formal semantics of the language (e.g. constant propagation)" [CoC77a]. We do not consider it reasonable to characterize "available expressions" as "syntactic", because then the information hardly can be used for anything. From our experience we may suggest that one cannot prove "available expressions" correct with respect to the usual semantics, but that one has to use a more "concrete" semantics that records the computational history.

In [CoC79,p.278] the "available expressions" interpretation is developed from a "set of traces" semantics. However, the transfer function, essentially λexp.generated $\cup$ (preserved $\cap$ exp), is not justified: the definitions of preserved and generated are neither given nor related to the semantics.

It appears that our treatment of available expressions is the first to give some semantic characterization of "available expressions analysis". It is not crucial that we work from a denotational semantics.

CHAPTER 5

The MOP Solution

The purpose of this chapter is to relate the data flow information specified by Table 2.2-C together with an induced interpretation (ind) to the solutions considered in traditional data flow analysis. In section 5.1 we define semantic functions $A\mathcal{P}$, $A\mathcal{D}$, $A\mathcal{C}$ and $A\mathcal{E}$ such that $A\mathcal{P}$ind$[\![$pro$]\!]$ specifies the transfer functions of program pro. These functions were called Bf<...,...> in section 2.3. We also state some properties fulfilled by the semantic functions.

In section 5.2 we show a certain relationship between $\mathcal{P}$ind$[\![$pro$]\!]$ and $A\mathcal{P}$ind$[\![$pro$]\!]$. This relationship can be formulated as: $\mathcal{P}$ind$[\![$pro$]\!]$ specifies the MOP solution of traditional data flow analysis. One possible conclusion is that our approach is valid <u>because</u> it specifies the MOP solution. We believe that this is not a reasonable conclusion because the fact that the MOP solution is wanted is only based on intuitive arguments. The converse conclusion is that it is correct to want the MOP solution <u>because</u> it agrees with the information specified by our approach. This is also not a reasonable conclusion, e.g. because there is a certain arbitrariness in going from std to col. The reasonable conclusion probably is that two different approaches, both intuitively correct, turn out to give the same result, thus raising our confidence in both of them.

In this chapter we only consider interpretation ind and especially in proofs we often omit the prefix ind-. The same development can be performed for his-ind and ae with only occasional changes in the notation. Also recall that sts and his-sts are special cases of ind and his-ind (respectively).

5.1 The Transfer Functions

One way to obtain the transfer functions pertaining to some program pro is to do as in "traditional" data flow analysis: First the program is syntactically transformed into a graph, where nodes are sequences of simple assignment statements and arcs represent flow of control (as mentioned in section 2.3). Transfer functions are then constructed and associated with the nodes. These two stages are usually not justified semantically: the syntactic transformation is considered intuitively obvious and the construction of the transfer functions is done without reference to any formal semantics. An exception is [CoC77a] where the transfer functions are justified with respect to an operational semantics.

For our purposes it is simpler to define one non-standard semantics (Table 5.1-A and an induced interpretation ind) that traverses the program and constructs the transfer functions. Hopefully, it is intuitively clear that the transfer functions constructed by $A\mathcal{P}$ind$[\![$pro$]\!]$ essentially correspond to those usually constructed in traditional data flow analysis; because only

TABLE 5.1-A(a) --- SEMANTIC FUNCTIONS COLLECTING ARCS

```
Domains
-------
     S              specified by the interpretation
     D = S -i> S
     U = Pla X D
 Trans = (Pla X Pla) X D
     F = U -t> U X 𝒫(Trans)

Auxiliary Functions
-------------------
  Arecord ∈ D -t> F
  Arecord(d-new)<pla,d-old> =
    <<pla,d-new∘d-old>,∅>

  Aattach ∈ Pla -t> F
  Aattach(pla-new)<pla-old,d> =
    <<pla-new,λs.s>, {<<pla-old,pla-new>,d>}>

  * ∈ F X F -t> F
  (f * g) u = < f(g(u)↓1)↓1, f(g(u)↓1)↓2 ∪ g(u)↓2>

  WITH ∈ F X F -t> F
  (f WITH g) u = < g(u)↓1, f(u)↓2 ∪ g(u)↓2>

  ALSO ∈ F X 𝒫(Trans) -t> F
  (f ALSO trans) u = < f(u)↓1, f(u)↓2 ∪ trans>

Semantic Functions
------------------
A𝒫 ∈ Pro -t> 𝒫(Trans)

  A𝒫⟦ BEG dcl IN cmd END ⟧ =
    [ A𝒞⟦cmd⟧ <2> *
      A𝒟⟦dcl⟧ <1> ] < <<>,"(pro">, λs.s> ↓2

A𝒟 ∈ Dcl -t> Occ -t> F

  A𝒟⟦ dcl1 ; dcl2 ⟧ occ =
    Aattach<occ,"dcl)"> *
    A𝒟⟦dcl2⟧ occ§<2> *
    A𝒟⟦dcl1⟧ occ§<1> *
    Aattach<occ,"(dcl">

  A𝒟⟦ DCL ide := bas ⟧ occ =
    Aattach<occ,"dcl)"> *
    Arecord(Dassign⟦ide⟧) *
    Arecord(Dpush⟦bas⟧) *
    Aattach<occ,"(dcl">

Aℰ ∈ Exp -t> Occ -t> F

  Aℰ⟦ exp1 ope exp2 ⟧ occ =
    Aattach<occ,"exp)"> *
    Arecord(Dapply⟦ope⟧) *
    Aℰ⟦exp2⟧ occ§<3> *
    Aℰ⟦exp1⟧ occ§<1> *
    Aattach<occ,"(exp">

  Aℰ⟦ ide ⟧ occ =
    Aattach<occ,"exp)"> *
    Arecord(Dcontent⟦ide⟧) *
    Aattach<occ,"(exp">

  Aℰ⟦ bas ⟧ occ =
    Aattach<occ,"exp)"> *
    Arecord(Dpush⟦bas⟧) *
    Aattach<occ,"(exp">
```

then Theorem 5.2-12 can be interpreted to say: 𝒫ind⟦pro⟧ specifies the MOP solution. We ignore the fact that A𝒫ind⟦pro⟧ considers smaller basic blocks than is usually done {a}.

```
TABLE 5.1-A(b) --- SEMANTIC FUNCTIONS COLLECTING ARCS
=====================================================

A𝒞 ∈ Cmd -t> Occ -t> F

  A𝒞⟦ cmd1 ; cmd2 ⟧ occ =
    Aattach<occ,"cmd)"> *
    A𝒞⟦cmd2⟧ occ§<2> *
    A𝒞⟦cmd1⟧ occ§<1> *
    Aattach<occ,"(cmd">

  A𝒞⟦ ide := exp ⟧ occ =
    Aattach<occ,"cmd)"> *
    Arecord(Dassign⟦ide⟧) *
    A𝒞⟦exp⟧ occ§<2> *
    Aattach<occ,"(cmd">

  A𝒞⟦ IF exp THEN cmd1 ELSE cmd2 FI ⟧ occ =
    ([ Aattach<occ,"cmd)"> *
       A𝒞⟦cmd2⟧ occ§<3> *
       Arecord(Df-cond) ] WITH
     [ Aattach<occ,"cmd)"> *
       A𝒞⟦cmd1⟧ occ§<2> *
       Arecord(Dt-cond) ] ) *
    A𝒞⟦exp⟧ occ§<1> *
    Aattach<occ,"(cmd">

  A𝒞⟦ WHILE exp DO cmd OD ⟧ occ =
    ([ ( A𝒞⟦cmd⟧ occ§<2> * Arecord(Dt-cond)) WITH
       ( Aattach<occ,"cmd)"> * Arecord(Df-cond)) ] *
     A𝒞⟦exp⟧ occ§<1> *
     Aattach<occ,"(cmd"> )
    ALSO {< <<occ§<2>,"cmd)">, <occ§<1>,"(exp">>, λs.s>}

  A𝒞⟦ WRITE exp ⟧ occ =
    Aattach<occ,"cmd)"> *
    Arecord(Dwrite) *
    A𝒞⟦exp⟧ occ§<1> *
    Aattach<occ,"(cmd">

  A𝒞⟦ READ ide ⟧ occ=
    Aattach<occ,"cmd)"> *
    Arecord(Dassign⟦ide⟧) *
    Arecord(Dread) *
    Aattach<occ,"(cmd">
```

In the sequel we explain the non-standard semantics as specified by Table 5.1-A and some interpretation ind.

To understand Table 5.1-A it may be helpful to mentally view a program pro as a graph (see section 2.3). This amounts to identifying arcs with places and letting a node be represented by a pair of places. The start arc corresponds to < <>,"(pro">. This leads to one way of understanding the semantics, even if this view sometimes may appear artificial.

{a} One way to remedy this is to remove some attach functions from Table 2.2-C and the corresponding Aattach functions from Table 5.1-A. A disadvantage of this is that approximations are then performed inside basic blocks. This can be avoided by using ind' instead of ind where ind' is mainly as sts but where attach (and similarly Aattach) is changed so as to apply the abstraction and concretization functions.

The intention with A𝒫ind⟦pro⟧ then is to specify
{<<pla1,pla2>,Bf<pla1,pla2>> | <pla1,pla2> is a path in pro going through one node}
A tuple <<pla1,pla2>,Bf<pla1,pla2>> will be called a transition and is an element of Trans = (Pla X Pla) X (S -i> S) = (Pla X Pla) X D where D = S -i> S is the domain of transfer functions. Thus the functionality of A𝒫ind is Pro -t> 𝒫(Trans).

More or less the intentions with A𝒞ind⟦cmd⟧, A𝒟ind⟦dcl⟧ and Aℰind⟦exp⟧ are as above. In the sequel we consider only A𝒞ind⟦cmd⟧.

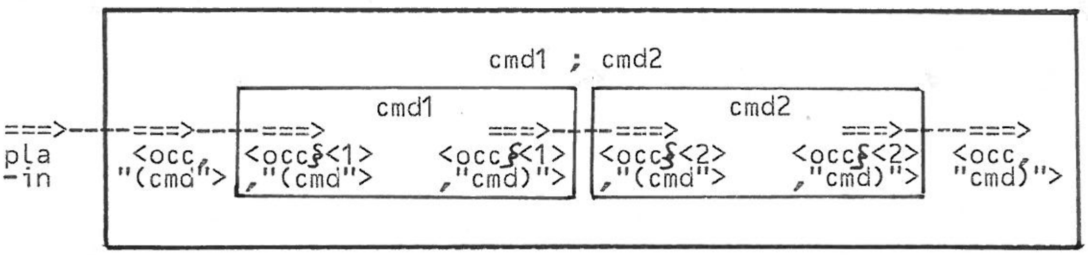
We supply A𝒞ind⟦cmd⟧ with an occurrence because places are defined in terms of occurrences and because we want distinct syntactic constructs to generate distinct places. The functionality of A𝒞ind⟦cmd⟧(occ) is more complicated than might have been expected:

A𝒞ind⟦cmd⟧ occ ∈ F   where F = U -t> U X 𝒫(Trans)
                     and U = Pla X D

Below we informally explain why we supply A𝒞ind⟦cmd⟧(occ) with an argument <pla,d> ∈ U.

That a node is represented by a pair of places implies that "the first place" encountered in cmd (namely <occ,"(cmd">) is part of a representation <pla,<occ,"(cmd">> of some node. To generate a transition for the transfer function (d) for that node we supply <pla,d> to A𝒞ind⟦cmd⟧(occ). Another way to explain the need to supply d to A𝒞ind⟦cmd⟧(occ) is that our approach unifies local and global analysis. When performing local analysis one "traces through" the internal of a node and must record the partial transfer function (d) encountered so far. Then when reaching a place <occ,"(cmd"> the transition <<pla,<occ,"(cmd">>,d> will be produced. - A reading of <pla,d> is: "since place pla was left the partial transfer function d has been recorded".

The need for A𝒞ind⟦cmd⟧(occ)<pla,d> to produce an element of U is best illustrated when cmd is cmd1;cmd2. The clause for A𝒞ind⟦ cmd1 ; cmd2 ⟧ can be depicted as shown on the figure below. Here ===> represents an arc (place) and ---- can be thought of as a node. The element of U to be supplied to A𝒞ind⟦cmd2⟧(occ§<2>) must be produced by A𝒞ind⟦cmd1⟧(occ§<1>).



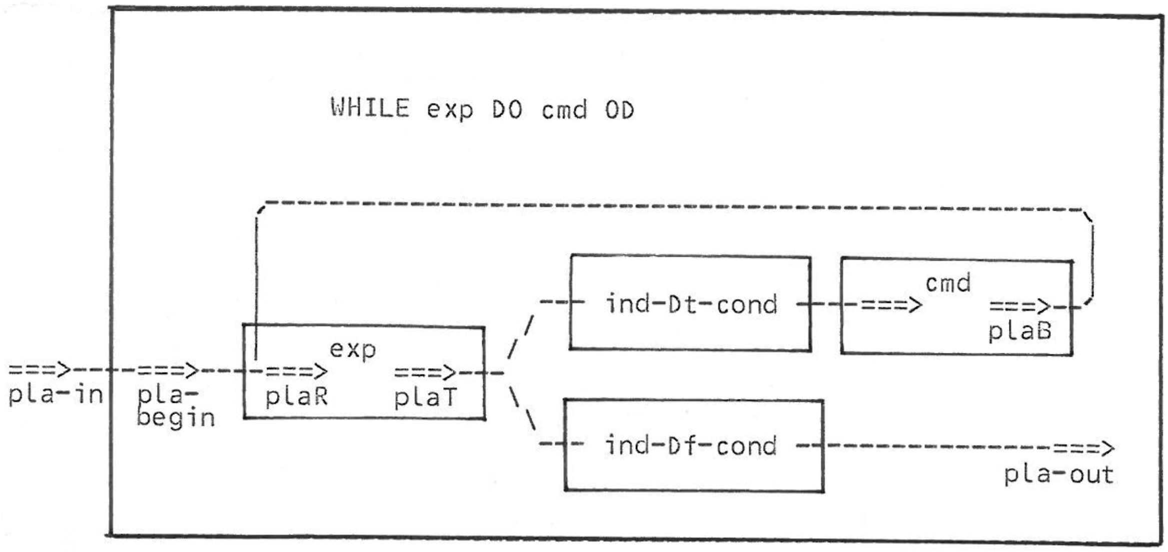
The transitions generated by A𝒞ind⟦cmd⟧(occ)<pla-in,d-in> include:

< <pla-in,<occ,"(cmd">> ,d-in>
A𝒞ind⟦cmd1⟧ occ§<1> < <occ,"(cmd">, λs.s> ↓2
A𝒞ind⟦cmd2⟧ occ§<2> < <occ§<1>,"cmd)">, λs.s> ↓2
< < <occ§<2>,"cmd)">, <occ,"cmd)"> >   , λs.s>

That this is so follows from Lemma 5.1-4.

To understand the clause for A𝒞ind⟦ WHILE exp DO cmd OD ⟧(occ) it may be helpful to consider the figure below. The abbreviations of places used will also be used in the proofs of Lemmas 5.2-8 and 5.2-11. Abbreviate

```
pla-begin = <occ,"(cmd">
plaR      = <occ§<1>,"(exp">             ("Repeat")
plaT      = <occ§<1>,"exp)">             ("Test")
plaB      = <occ§<2>,"cmd)">             ("Back")
pla-out   = <occ,"cmd)">
```



The boxes labelled ind-Df-cond and ind-Dt-cond represent the effect of these functions, i.e. to choose the false or true branch {a}.

The transitions generated by A𝒞ind⟦ WHILE exp DO cmd OD ⟧(occ)<pla-in,d-in> include:

```
< <pla-in,pla-begin>, d-in>
A𝓔ind⟦exp⟧ occ§<1> <pla-begin, λs.s> ↓2
A𝒞ind⟦cmd⟧ occ§<2> <plaT, ind-Dt-cond> ↓2
< <plaB,plaR>, λs.s>
< <plaT,pla-out>, ind-Df-cond>
```

This follows from Lemma 5.1-4. As will become clear in section 5.2 it is the transition < <plaB,plaR> ,λs.s> which accounts for the "iterative" nature of a WHILE construct.

Similarly to the situation in section 2.2 we have:

LEMMA 5.1-1: Table 5.1-A and interpretation ind define functions A𝒫ind, A𝒟ind, A𝒞ind and A𝓔ind. They are of functionalities as shown and no function is supplied with an argument of the wrong type. []

We now state some properties of the non-standard semantics (Lemmas 5.1-4 and 5.1-5). We first define some concepts which are explained afterwards.

{a} In data flow analysis it is often assumed that Dt-cond = Df-cond = λs.s. This is not the case for ind but one can, of course, define a more approximate interpretation apr such that this holds.

DEFINITION 5.1-2:

Define xpld: Occ $\to \mathcal{P}$(Pla) (for "explode") by

$xpld(occ) = \{\langle occ', q'\rangle \mid occ' = occ\S\langle\ldots\rangle\}$.

Here $occ' = occ\S\langle\ldots\rangle$ is a shorthand for $\exists occ''\in Occ-\{\bot,\top\}: occ' = occ\S occ''$.

Define out: F $\to$ Pla by $out(Ag) = Ag\langle \langle\rangle,"?"\rangle, \lambda s.s\rangle \downarrow 1 \downarrow 1$.

Define local: F $\to \mathcal{P}$(Pla) by

$local(Ag) = \{pla2 \mid \langle\langle pla1,pla2\rangle,d\rangle \in Ag\langle \langle\rangle,"?"\rangle, \lambda s.s\rangle \downarrow 2\}$. []

DEFINITION 5.1-3:

Define the predicate P-F: F $\to$ B by P-F(Ag) =

i) $\exists pla\text{-}out\ \forall\langle pla\text{-}in,d\text{-}in\rangle\in U: Ag\langle pla\text{-}in,d\text{-}in\rangle\downarrow 1 = \langle pla\text{-}out,\lambda s.s\rangle$

ii) $\exists \underline{pla}\in\mathcal{P}(Pla)\ \forall\langle pla\text{-}in,d\text{-}in\rangle\in U$:

$out(Ag) \in \underline{pla}\ \wedge$

$\{pla1 \mid \langle\langle pla1,pla2\rangle,d\rangle \in Ag\langle pla\text{-}in,d\text{-}in\rangle\downarrow 2\} \subseteq (\underline{pla}-\{out(Ag)\})\cup\{pla\text{-}in\}\ \wedge$

$\{pla2 \mid \langle\langle pla1,pla2\rangle,d\rangle \in Ag\langle pla\text{-}in,d\text{-}in\rangle\downarrow 2\} = \underline{pla}$

iii) $\forall pla\text{-}in1\notin local(Ag)\ \forall pla\text{-}in2\in Pla\ \forall d\text{-}in1,d\text{-}in2\in D$:

LET $\underline{trans1} = Ag\langle pla\text{-}in1,d\text{-}in1\rangle\downarrow 2$ IN

LET $\underline{trans2} = Ag\langle pla\text{-}in2,d\text{-}in1\circ d\text{-}in2\rangle\downarrow 2$ IN

$\underline{trans2} = \{\langle\langle pla\text{-}in2,pla\rangle,d\circ d\text{-}in2\rangle \mid \langle\langle pla\text{-}in1,pla\rangle,d\rangle\in \underline{trans1}\}$

$\quad\cup \{\langle\langle pla1,pla2\rangle,d\rangle \mid pla1\neq pla\text{-}in1 \wedge \langle\langle pla1,pla2\rangle,d\rangle\in\underline{trans1}\}$ []

Let $Ag = A\mathcal{C}ind[\![cmd]\!](occ)$ and assume P-F(Ag). Condition "i)" intuitively says that regardless from where (pla-in) cmd is entered it must be left by the arc pla-out. Clearly pla-out = out(Ag).

Condition "ii)" intuitively says that there is a set $\underline{pla}$ of places that "occur in cmd", one of which is out(Ag). When cmd is entered from pla-in any transition $\langle\langle pla1,pla2\rangle,d\rangle$ generated has $pla1\in(\underline{pla}-\{out(Ag)\})\cup\{pla\text{-}in\}$ and $pla2\in\underline{pla}$. Clearly $\underline{pla}$ = local(Ag).

Condition "iii)" says what use cmd makes of the place and partial transfer function it is entered with. When $d\text{-}in2 = \lambda s.s$ the condition expresses what difference it makes to come from pla-in2 rather than pla-in1. When pla-in1 = pla-in2 it expresses how the partial transfer function influences the set of transitions produced.

We use xpld to express conditions like $local(Ag) \subseteq xpld(occ)$. Intuitively this says that any place occurring in cmd has $pla\downarrow 1$ to be an extension of occ.

Lemmas 5.1-4 and 5.1-5 state properties of the non-standard semantics.

LEMMA 5.1-4:

a) $\forall cmd$: $P\text{-}F(A\mathcal{C}ind[\![cmd]\!]occ)$
$\quad\wedge\ local(A\mathcal{C}ind[\![cmd]\!]occ) \subseteq xpld(occ)$
$\quad\wedge\ out(A\mathcal{C}ind[\![cmd]\!]occ) = \langle occ,"cmd)"\rangle$

b) $\forall exp$: $P\text{-}F(A\mathcal{E}ind[\![exp]\!]occ)$
$\quad\wedge\ local(A\mathcal{E}ind[\![exp]\!]occ) \subseteq xpld(occ)$
$\quad\wedge\ out(A\mathcal{E}ind[\![exp]\!]occ) = \langle occ,"exp)"\rangle$

c) $\forall exp$: $\forall pla\text{-}in\notin local(A\mathcal{E}ind[\![exp]\!]occ)$: $\forall d\text{-}in\in D$:
$\quad\{\langle\langle pla1,pla2\rangle,d\rangle \mid pla1=pla\text{-}in\ \wedge$
$\qquad\qquad \langle\langle pla1,pla2\rangle,d\rangle\in [A\mathcal{E}ind[\![exp]\!](occ)\langle pla\text{-}in,d\text{-}in\rangle\downarrow 2]\}$
$\quad = \{\langle\langle pla\text{-}in, \langle occ,"(exp"\rangle\rangle,d\text{-}in\rangle\}$

d) $\forall dcl$: $P\text{-}F(A\mathcal{D}ind[\![dcl]\!]occ)$
$\quad\wedge\ local(A\mathcal{D}ind[\![dcl]\!]occ) \subseteq xpld(occ)$
$\quad\wedge\ out(A\mathcal{D}ind[\![dcl]\!]occ) = \langle occ,"dcl)"\rangle$ []

Proof is by an (omitted) structural induction that uses the results of Lemma 5.1-5. []

Analogues of "c)" also hold for commands and declarations but this will not be needed. We explain the results of Lemma 5.1-5 below.

LEMMA 5.1-5:

1) P-F(Aattach pla) $\wedge$ local(Aattach pla) = {pla} $\wedge$ out(Aattach pla) = pla
2) P-F(Ag) =>
   P-F(Ag*Arecord(d)) $\wedge$ local(Ag*Arecord(d)) = local(Ag) $\wedge$
   out(Ag*Arecord(d)) = out(Ag)
3) P-F(Ag1) $\wedge$ P-F(Ag2) $\wedge$ out(Ag2) $\notin$ local(Ag1) =>
   P-F(Ag2*Ag1) $\wedge$ local(Ag2*Ag1) = local(Ag2) $\cup$ local(Ag1)
   $\wedge$ out(Ag2*Ag1) = out(Ag2)
4) P-F(Ag1) $\wedge$ P-F(Ag2) $\wedge$ out(Ag2) $\notin$ [local(Ag1)-{out(Ag1)}] =>
   P-F(Ag1 WITH Ag2) $\wedge$ local(Ag1 WITH Ag2) = local(Ag1) $\cup$ local(Ag2)
   $\wedge$ out(Ag1 WITH Ag2) = out(Ag2)
5) P-F(Ag) $\wedge$ pla1 $\in$ [local(Ag)-{out(Ag)}] =>
   P-F(Ag ALSO {<<pla1,pla2>,d>}) $\wedge$
   local(Ag ALSO {<<pla1,pla2>,d>}) = local(Ag) $\cup$ {pla2}
   out(Ag ALSO {<<pla1,pla2>,d>}) = out(Ag) []

Proof is omitted. []

The intuitive content of "3)" is that P-F is "preserved under sequencing". The condition out(Ag2) $\notin$ local(Ag1) is fulfilled when local(Ag1) $\cap$ local(Ag2) = $\emptyset$ as is usually the case. Combinator * is as in Table 5.1-A.

One way that out(Ag2) $\notin$ [local(Ag1)-{out(Ag1)}] may hold in "4)" is when out(Ag2) = out(Ag1) as is the case for the conditional. Case "5)" is only needed when considering the WHILE loop.

From A𝒫ind⟦pro⟧ it is possible to define a <u>partial</u> function Bf (from Pla X Pla to D) such that Bf<pla1,pla2> = d <=> <<pla1,pla2>,d> $\in$ A𝒫ind⟦pro⟧. This Bf corresponds to the Bf of section 2.3, and thus complements our earlier explanation about the intention with A𝒫ind⟦pro⟧.

5.2 Equivalence of the Solutions

In this section we show the desired connection between the data flow information specified by our approach (𝒫ind⟦pro⟧) and the MOP solution. To do so we specify what we understand by the MOP solution. For this we need the function Close:

DEFINITION 5.2-1: For S = ind-S and Trans as in Table 5.1-A we define
Close: $\mathcal{P}$(Trans) X Pla X $\mathcal{P}$(S) -> (Pla -t> $\mathcal{P}$(S)) by
Close($\underline{trans}$, pla-in, $\underline{s}$) pla-out =
{dn(...(d1(s))) | s$\in\underline{s}$ $\wedge$ n$\geq$1 $\wedge$
$\exists$<<pla[0],pla[1]>,d1>,...,<<pla[n-1],pla[n]>,dn>$\in$ $\underline{trans}$
so that pla-in = pla[0] and pla-out = pla[n] } []

OBSERVATION 5.2-2: $\forall$pla $\forall\underline{s}$: $\lambda\underline{trans}$.Close($\underline{trans}$, pla, $\underline{s}$) is isotone. []

Recall that R(f)(g) = λl.f(g(l)) (subsection 2.4.2) and let start = <<>,"(pro"> and init$\in$S. Note that (R $\sqcup$) (Close(A𝒫ind⟦pro⟧, start, {init})) closely corresponds to the formulation of the MOP solution in section 2.3: λarc.$\sqcup${Bf<start,...,arc> init | <start,...,arc> is a path }. This is because Bf and A𝒫ind⟦pro⟧ are related as stated in section 5.1. A difference that we will ignore throughout is that the two formulations disagree on start.

To prove (and state) the relationship between $\mathcal{P}$ind⟦pro⟧ and A$\mathcal{P}$ind⟦pro⟧ we define the predicate P-G (explained below):

DEFINITION 5.2-3:
Define EQ: (Pla -t> S) X (Pla -t> S) -> B by
cl1 EQ cl2 = $\forall$pla: [pure[Pla](pla) => cl1(pla) = cl2(pla)].
For C = ind-C define P-G: (C -c> C) X F -> B by
P-G(g,Ag) = P-F(Ag) $\wedge$
[$\forall$pla-in$\notin$local(Ag): $\forall c \in$C: $\forall \underline{s} \in \mathcal{P}$(S):
$\bigsqcup${g(c)(s) |s$\in\underline{s}$} EQ $\bigsqcup${c(s) |s$\in$cl(out(Ag))} $\sqcup$ (R $\bigsqcup$) cl
where cl = Close(Ag<pla-in,λs.s>$\downarrow$2, pla-in, $\underline{s}$)] []

The use of EQ instead of = is motivated later (in the proof of Lemma 5.2-5). Note that the first two $\bigsqcup$ in P-G are joins of Pla -t> S whereas the third is the join of S.

Suppose c = ind-finish = $\perp$ and $\underline{s}$ = {init} in the definition of P-G. Then P-G(g,Ag) says that (g;c)init equals the MOP solution (R $\bigsqcup$)(cl). When c $\neq \perp$ the expression $\bigsqcup${c(s) |s$\in$cl(out(Ag))} is needed to represent the effect of "the rest of the program". To understand it note that cl(out(Ag)) is intended to be (when $\underline{s}$ = {init}):
{Bf<pla-in,...,out(Ag)> init | <pla-in,...,out(Ag)> is a path }
We need to write $\bigsqcup${c(s) | s$\in$cl(out(Ag))} rather than c($\bigsqcup$(cl(out(Ag)))) because we do not assume that ind is a "distributive framework".

A partial result in showing that $\mathcal{P}$ind⟦pro⟧ is the MOP solution is:

LEMMA 5.2-4:
pure[Occ](occ) => P-G($\mathcal{D}$ind⟦dcl⟧occ, A$\mathcal{D}$ind⟦dcl⟧occ)
pure[Occ](occ) => P-G($\mathcal{E}$ind⟦exp⟧occ, A$\mathcal{E}$ind⟦exp⟧occ) []
Proof We omit the proof, which is by structural induction. Use is made of Lemmas 5.2-5, 5.2-6 and 5.2-7. []

The proof of the following lemma shows why we use EQ rather than =.

LEMMA 5.2-5: pure[Pla](pla) => P-G(ind-attach pla, ind-Aattach pla) []
Proof is shown in appendix 5. []

LEMMA 5.2-6: P-G(g2,Ag2) $\wedge$ g1 = λc.c$\circ$Dg1 => P-G(λc.g1;g2;c, Ag2*Arecord(Dg1))

LEMMA 5.2-7: P-G(g1,Ag1) $\wedge$ P-G(g2,Ag2) $\wedge$ local(Ag1)$\cap$local(Ag2) = $\emptyset$ =>
P-G(λc.g1;g2;c, Ag2*Ag1) []
Proof is shown in appendix 5. []

One way to paraphrase the condition local(Ag1)$\cap$local(Ag2) = $\emptyset$ is: "the set of places occurring in g1 is disjoint from the set of places occurring in g2".

To intuitively and informally explain how this condition is used in the proof one may perceive f$\in$F as a "flowchart". One traces through f by (non-deterministically) following a path <pla0,...,pla[n]> where
$\forall$i$\in${1,...,n} $\exists$di: <<pla[i-1],pla[i]>,di> $\in$ f<pla0,λs.s> $\downarrow$2.

Then the condition local(Ag1)$\cap$local(Ag2) = $\emptyset$ is used in the proof to express the situation in which g2 may be "entered" from g1 and to assert that once g2 has "been entered" from g1 it is impossible to "enter" g1 again.

Another partial result in showing that $\mathcal{P}$ind⟦pro⟧ is the MOP solution is:

LEMMA 5.2-8: pure[Occ](occ) => P-G($\mathcal{C}$ind⟦cmd⟧occ, A$\mathcal{C}$ind⟦cmd⟧occ) []

Proof is shown in appendix 5. □

It is more difficult to establish this Lemma than Lemma 5.2-4. The proof uses Lemmas 5.2-9 and 5.2-11 below. Lemma 5.2-9 is useful for handling the conditional.

```
LEMMA 5.2-9:  P-G(g1,Ag1) ∧ P-G(g2,Ag2) ∧ out(Ag1)=out(Ag2) ∧
 local(Ag1)∩local(Ag2)={out(Ag2)} =>
 P-G(λc.cond(g1;c, g2;c),
     [Ag1*Arecord(ind-Dt-cond)] WITH [Ag2*Arecord(ind-Df-cond)])          □
 Proof is shown in appendix 5.                                            □
```

The condition local(Ag1)∩local(Ag2)={out(Ag2)}={out(Ag1)} is used in the proof to assert that once g1 has been "entered" it is impossible to "enter" g2, and vice versa.

To factor out the complexities of handling the WHILE construct we state the "technical" Lemma 5.2-11. It employs the function Luk which essentially is as Close but constrains the use of some transitions.

```
DEFINITION 5.2-10:  Define
 Luk: 𝒫(Trans) X Pla X 𝒫(S) X Integer X 𝒫(Trans) -> ( Pla -t> 𝒫(S) ) by
 Luk( trans1, pla-in, s, k, trans2) pla-out =
 {dn(...(d1(s))) | s∈s ∧ n≥1 ∧
        ∃ <<pla[0],pla[1]>,d1>,...,<<pla[n-1],pla[n]>,dn>∈trans1∪trans2
        so that pla[0]=pla-in ∧ pla[n]=pla-out
        and |{i | <<pla[i-1],pla[i]>,di> ∈ trans2}| = k }                  □
```

Here |{...}| is the cardinality of the set {...}.

```
LEMMA 5.2-11:
 Abbreviate:
    pla-begin = <occ,"(cmd">                    plaR = <occ§<1>,"(exp">
    plaT = <occ§<1>,"exp)">                     plaB = <occ§<2>,"cmd)">
    pla-out = <occ,"cmd)">

    (A)g1  = (A)𝔈ind⟦exp⟧ occ§<1>
    (A)g2  = (A)𝔊ind⟦cmd⟧ occ§<2>

    g[c]c' = g1; cond(g2;c', attach(pla-out);c)
    Ag = ([Ag2*Arecord(Dt-cond)]
          WITH [Aattach(pla-out)*Arecord(Df-cond)])
         * Ag1

    a[k+1] = Luk( Ag<pla-begin,λs.s>↓2, pla-begin, s, k, {<<plaB,plaR>,λs.s>})
 Assume:
    pure[Occ](occ) ∧ P-G(g1,Ag1) ∧ P-G(g2,Ag2)
 Then ∀k≥1:
    ⊔{(g[c])^K(⊥)(s) | s∈s} EQ ⊔{c(s) | s∈([ak∪...∪a1] pla-out) } ∪
                                (R ⊔)[ak∪...∪a1]                          □
 Proof is shown in appendix 5.                                            □
```

The abbreviations of places are in accordance with the figure for the WHILE loop in section 5.1. Lemma 5.2-11 intuitively says that ⊔{(g[c])^K(⊥)(s) | s∈s} approximates the effect of the WHILE construct (and the rest of the program) by allowing the WHILE loop to be iterated at most k-1 times. Here one iteration means to "follow" the <<plaB,plaR>,λs.s>

transition once.

From Lemmas 5.2-4, 5.2-7 and 5.2-8 it is quite easy to show that $\mathcal{P}ind[\![pro]\!]$ specifies the MOP solution:

THEOREM 5.2-12: $\forall pro \in Pro: \forall \underline{inp} \in \mathcal{P}(Inp): \mathcal{P}ind[\![pro]\!]\underline{inp}$ EQ

$$\begin{aligned}
&\lambda pla.\bigsqcup\{\, dn(\ldots(d1(uabs\{\, \langle \lambda ide.\text{"nil"}\ inVal,\ inp,\ \langle\rangle,\ \langle\rangle\rangle \mid inp \in \underline{inp}\})))\ \mid \\
&\qquad n \geq 1 \wedge \exists \langle\langle pla0, pla1\rangle, d1\rangle, \ldots, \langle\langle pla[n-1], pla[n]\rangle, dn\rangle \in A\mathcal{P}ind[\![pro]\!] \\
&\qquad \wedge pla[0] = \langle\langle\rangle, \text{"(pro"}\rangle \wedge pla[n] = pla\}
\end{aligned}$$ []

Proof is shown in appendix 5. []

As mentioned earlier this development also holds for his-ind and ae as well as for sts and his-sts.

CHAPTER 6

Program Transformations

In this chapter (section 6.1) we investigate how data flow information specified by our approach can be used to validate a class of program transformations that includes a "constant folding" example [AhU78,p.409]. In section 6.2 we briefly relate our approach to other work.

A property of a denotational semantics is that it is always possible to replace one syntactic construct with another that has the same denotation {a}. But this is not sufficient to validate the program transformations considered e.g. in optimizing compilers. Consider constant folding that replaces an expression (e.g. i+3) by a constant (e.g. 5) provided the expression can only be "reached" with a constant value of the identifiers occurring in it (e.g. i=2). But the denotations of the expression (i+3) and the constant (5) are usually not the same.

By a "computational context" we understand a set of states that are possible at some place, i.e. $\mathcal{P}sts[\![pro]\!]\underline{inp}\langle occ,\text{"(..."}\rangle$. To be able to validate program transformations it is necessary only to require that the two syntactic constructs produce the same result in the computational context in which they occur. In the example above it is sufficient that any state in the computational context maps i to 2. Our method is one way of validating program transformations by taking computational contexts into account.

6.1 The Method

We formalize (in subsection 6.1.1) the notion of "a labelled parsetree" and formally define some operations upon labelled parsetrees; these operations are not considered in Denotational Semantics. Then (in subsection 6.1.2) we formulate the semantic functions in the new notation and prove some properties of continuation removal. Our subsequent development is mostly in terms of the continuation removed functions.

In subsection 6.1.3 we prove some "intuitively obvious" properties about the data flow information specified. The properties are needed in subsection 6.1.4 where we prove sufficient "local" conditions for when a program is semantically equivalent to a transformed program. These conditions take the computational context into account and are adequate to give an easily tested condition for the validity of "constant folding" as well as many other program transformations.

{a} When talking about "denotation" and "semantically equivalent" we assume this is with respect to interpretation std.

6.1.1 Labelled Parse-trees

In this subsection we give a formal definition of a labelled parse-tree. We do so in order to give formal definitions of "the subtree at occurrence lab" and "the tree obtained by replacing the subtree at lab by ...". The definitions are adapted from [Ros73].

We need the "names" of the productions of Table 2.2-A:

```
Prod = { <"pro","BEG dcl IN cmd END">,
         <"dcl","dcl1 ; dcl2">, <"dcl","DCL ide := bas">,
         <"cmd","cmd1 ; cmd2">, <"cmd","ide := exp">,
           <"cmd","IF exp THEN cmd1 ELSE cmd2 FI">,
           <"cmd","WHILE exp DO cmd OD">,
           <"cmd","WRITE exp">, <"cmd","READ ide">,
         <"exp","exp1 ope exp2">, <"exp","ide">, <"exp","bas"> }
       ∪ {<"ide",ide> | ide∈Ide}
       ∪ {<"bas",bas> | bas∈Bas}
       ∪ {<"ope",ope> | ope∈Ope}
```

We assume that no two elements of Prod have the same second component. It is convenient to define "names" for the syntactic categories by Cat = {prod↓1 | prod∈Prod} and similarly "names" for the right-hand sides by Str = {prod↓2 | prod∈Prod}. Finally, we need a set of labels. We use Lab = {occ∈Occ | pure[Occ](occ)} rather than Occ to avoid some special cases below (e.g. so that root below is always defined).

We now define the labelled parse trees by defining ∀lab∈Lab: ∀cat∈Cat: the sets Tree(lab,cat) ⊆ ℘(LabXProd). Intuitively, t∈Tree(lab,cat) is a labelled parse-tree of category cat whose root is labelled lab. Formally, it is a set containing for each node a tuple (label and production). The trees are defined inductively as shown by Table 6.1-A, where we "imitate" the productions of Table 2.2-A. It is convenient to define Tree = ⋃{Tree(lab,cat) | lab∈Lab ∧ cat∈Cat}.

Below we state some properties fulfilled by the trees. They are intuitively obvious and we omit the proofs. We also define some notation.

OBSERVATION 6.1.1-1:

```
t∈Tree(lab,cat) => ∃!str: <lab,<cat,str>>∈t ∧ ∀lab',cat',str':
  [ <lab',<cat',str'>> ∈ t-{<lab,<cat,str>>} => lab≠lab' ∧ lab'=lab§<...>]
t∈Tree => ∃!<lab,<cat,str>>: <lab,<cat,str>>∈t ∧ t∈Tree(lab,cat)          []
```

Similarly to the abbreviations made in chapter 4 we let (for x∈LabXProd) x.lab mean x↓1, x.cat mean x↓2↓1 and x.str mean x↓2↓2. Also lab1≠lab2§<...> means ∀lab3: lab1≠lab2§lab3.

DEFINITION 6.1.1-2:

```
root: Tree -> LabXProd is defined by
root(t) = <lab,<cat,str>> iff [t∈Tree(lab,cat) ∧ <lab,<cat,str>>∈t]

at: TreeXLab -> ℘(LabXProd) is defined by
t at lab = {<lab',<cat,str>> | lab'=lab§<...> ∧ <lab',<cat,str>>∈t}

dom: Tree -> ℘(Lab) is defined by
dom(t) = {lab | <lab,<cat,str>>∈t}
```

TABLE 6.1-A --- DEFINITION OF PARSE-TREES
===

```
t1∈Tree(lab§<1>,"dcl") ∧ t2∈Tree(lab§<2>,"cmd") =>
    {<lab,<"pro","BEG dcl IN cmd END">>}∪t1∪t2 ∈ Tree(lab,"pro")
---
t1∈Tree(lab§<1>,"dcl") ∧ t2∈Tree(lab§<2>,"dcl") =>
    {<lab,<"dcl","dcl1;dcl2">>}∪t1∪t2 ∈ Tree(lab,"dcl")

t1∈Tree(lab§<1>,"ide") ∧ t2∈Tree(lab§<2>,"bas") =>
    {<lab,<"dcl","DCL ide:=bas">>}∪t1∪t2 ∈ Tree(lab,"dcl")
---
t1∈Tree(lab§<1>,"cmd") ∧ t2∈Tree(lab§<2>,"cmd") =>
    {<lab,<"cmd","cmd1;cmd2">>}∪t1∪t2 ∈ Tree(lab,"cmd")

t1∈Tree(lab§<1>,"ide") ∧ t2∈Tree(lab§<2>,"exp") =>
    {<lab,<"cmd","ide:=exp">>}∪t1∪t2 ∈ Tree(lab,"cmd")

t1∈Tree(lab§<1>,"exp") ∧ t2∈Tree(lab§<2>,"cmd") ∧ t3∈Tree(lab§<3>,"cmd") =>
    {<lab,<"cmd","IF exp THEN cmd1 ELSE cmd2 FI">>}∪t1∪t2∪t3 ∈ Tree(lab,"cmd")

t1∈Tree(lab§<1>,"exp") ∧ t2∈Tree(lab§<2>,"cmd") =>
    {<lab,<"cmd","WHILE exp DO cmd OD">>}∪t1∪t2 ∈ Tree(lab,"cmd")

t1∈Tree(lab§<1>,"exp") =>
    {<lab,<"cmd","WRITE exp">>}∪t1 ∈ Tree(lab,"cmd")

t1∈Tree(lab§<1>,"ide") =>
    {<lab,<"cmd","READ ide">>}∪t1 ∈ Tree(lab,"cmd")
---
t1∈Tree(lab§<1>,"exp") ∧ t2∈Tree(lab§<2>,"ope") ∧ t3∈Tree(lab§<3>,"exp") =>
    {<lab,<"exp","exp1 ope exp2">>}∪t1∪t2∪t3 ∈ Tree(lab,"exp")

t1∈Tree(lab§<1>,"ide") =>
    {<lab,<"exp","ide">>}∪t1 ∈ Tree(lab,"exp")

t1∈Tree(lab§<1>,"bas") =>
    {<lab,<"exp","bas">>}∪t1 ∈ Tree(lab,"exp")
---
ide∈Ide =>
    {<lab,<"ide",ide>>} ∈ Tree(lab,"ide")

bas∈Bas =>
    {<lab,<"bas",bas>>} ∈ Tree(lab,"bas")

ope∈Ope =>
    {<lab,<"ope",ope>>} ∈ Tree(lab,"ope")
```

```
...(...<-...): TreeXLabXTree -> ℘(LabXProd) is subtree replacement and is
defined by
t1(lab<-t2) = {<lab',<cat',str'>>∈t1 | lab'≠lab§<...>}
              ∪ {<lab§lab',<cat',str'>> | <lab',<cat',str'>>∈t2}

soni: Tree->℘(LabXProd) is defined by
soni(t) = t at (root(t).lab§<i>)                                       []
```

OBSERVATION 6.1.1-3:

```
tree∈Tree ∧ lab∈dom(tree) => tree at lab ∈ Tree
tree1∈Tree ∧ (∃str: <lab,<cat,str>>∈tree1) ∧ tree2∈Tree(<>,cat) =>
      tree1(lab<-tree2)∈Tree                                           []
```

TABLE 6.1-B --- SEMANTIC FUNCTIONS 𝒫 and 𝒯
==

```
𝒯 ∈ Tree -t> C -c> C
𝒯(tree)(c) =
  CASE root(tree).str OF
  "BEG dcl IN cmd END":
    attach<root(tree).lab,"(">;
    𝒯(son1(tree));
    𝒯(son2(tree));
    attach<root(tree).lab,")">;c
  "dcl1 ; dcl2":
    attach<root(tree).lab,"(">;
    𝒯(son1(tree));
    𝒯(son2(tree));
    attach<root(tree).lab,")">;c
  "DCL ide:=bas":
    attach<root(tree).lab,"(">;
    push( root(son2(tree)).str );
    assign( root(son1(tree)).str );
    attach<root(tree).lab,")">;c
  "exp1 ope exp2":
    attach<root(tree).lab,"(">;
    𝒯(son1(tree));
    𝒯(son3(tree));
    apply( root(son2(tree)).str );
    attach<root(tree).lab,")">;c
  "ide":
    attach<root(tree).lab,"(">;
    content( root(son1(tree)).str );
    attach<root(tree).lab,")">;c
  "bas":
    attach<root(tree).lab,"(">;
    push( root(son1(tree)).str );
    attach<root(tree).lab,")">;c
  "cmd1 ; cmd2":
    attach<root(tree).lab,"(">;
    𝒯(son1(tree));
    𝒯(son2(tree));
    attach<root(tree).lab,")">;c
  "ide := exp":
    attach<root(tree).lab,"(">;
    𝒯(son2(tree));
    assign( root(son1(tree)).str );
    attach<root(tree).lab,")">;c
  "IF exp THEN cmd1 ELSE cmd2 FI":
    attach<root(tree).lab,"(">;
    𝒯(son1(tree));
    cond( 𝒯(son2(tree));
            attach<root(tree).lab,")">;c
        , 𝒯(son3(tree));
            attach<root(tree).lab,")">;c)
  "WHILE exp DO cmd OD":
    attach<root(tree).lab,"(">;
    FIX( λc'. 𝒯(son1(tree));
              cond( 𝒯(son2(tree));c'
                  , attach<root(tree).lab,")">;c))
  "WRITE exp":
    attach<root(tree).lab,"(">;
    𝒯(son1(tree));
    write;
    attach<root(tree).lab,")">;c
  "READ ide":
    attach<root(tree).lab,"(">;
    read;
    assign( root(son1(tree)).str );
    attach<root(tree).lab,")">;c
  OTHERWISE:
    ⊥
  ESAC
```

```
𝒫 ∈ Tree(<>,"pro") -t> I -t> A
𝒫(tree) =
  setup( 𝒯(tree);finish )
```

TABLE 6.1-C --- SEMANTIC FUNCTION P𝒯
==================================

```
P𝒯 ∈ Tree -t> S -c> R
P𝒯(tree) =
  CASE root(tree).str OF
  "BEG dcl IN cmd END":
    Pattach<root(tree).lab,")"> *
    P𝒯(son2(tree)) *
    P𝒯(son1(tree)) *
    Pattach<root(tree).lab,"(">
  "dcl1 ; dcl2":
    Pattach<root(tree).lab,")"> *
    P𝒯(son2(tree)) *
    P𝒯(son1(tree)) *
    Pattach<root(tree).lab,"(">
  "DCL ide:=bas":
    Pattach<root(tree).lab,")"> *
    Passign( root(son1(tree)).str ) *
    Ppush( root(son2(tree)).str ) *
    Pattach<root(tree).lab,"(">
  "exp1 ope exp2":
    Pattach<root(tree).lab,")"> *
    Papply( root(son2(tree)).str ) *
    P𝒯(son3(tree)) *
    P𝒯(son1(tree)) *
    Pattach<root(tree).lab,"(">
  "ide":
    Pattach<root(tree).lab,")"> *
    Pcontent( root(son1(tree)).str ) *
    Pattach<root(tree).lab,"(">
  "bas":
    Pattach<root(tree).lab,")"> *
    Ppush( root(son1(tree)).str) *
    Pattach<root(tree).lab,"(">
  "cmd1 ; cmd2":
    Pattach<root(tree).lab,")"> *
    P𝒯(son2(tree)) *
    P𝒯(son1(tree)) *
    Pattach<root(tree).lab,"(">
  "ide := exp":
    Pattach<root(tree).lab,")"> *
    Passign( root(son1(tree)).str ) *
    P𝒯(son2(tree)) *
    Pattach<root(tree).lab,"(">
  "IF exp THEN cmd1 ELSE cmd2 FI":
    Pcond( Pattach<root(tree).lab,")"> *
           P𝒯(son2(tree))
         , Pattach<root(tree).lab,")"> *
           P𝒯(son3(tree)) ) *
    P𝒯(son1(tree)) *
    Pattach<root(tree).lab,"(">
  "WHILE exp DO cmd OD":
    FIX[ λss. Pcond( ss * P𝒯(son2(tree))
                   , Pattach<root(tree).lab,")"> ) *
              P𝒯(son1(tree)) ] *
    Pattach<root(tree).lab,"(">
  "WRITE exp":
    Pattach<root(tree).lab,")"> *
    Pwrite *
    P𝒯(son1(tree)) *
    Pattach<root(tree).lab,"(">
  "READ ide":
    Pattach<root(tree).lab,")"> *
    Passign( root(son1(tree)).str ) *
    Pread *
    Pattach<root(tree).lab,"(">
  OTHERWISE:
    ⊥
  ESAC
```

6.1.2 Semantic Functions

In this subsection we formulate the semantic functions with respect to the new "notation" (Table 6.1-B). Also we perform continuation removal for the entire language (Table 6.1-C).

There is a close correspondence between Tables 6.1-B and 2.2-C as well as some minor differences. One is that we have combined $\mathcal{D}$, $\mathcal{C}$ and $\mathcal{E}$ into the single function $\mathcal{T}$. Also (in attach) we elide the "dcl", "cmd" and "exp" parts of the places. The changes have been performed to simplify later formulations but are otherwise unimportant. In the definition of $\mathcal{T}$ we use a CASE construct that is hopefully self-explanatory.

Similarly to Theorem 2.2-4 and Lemma 4.2.1-1 we have (omitting the proofs):

LEMMA 6.1.2-1: Table 6.1-B and an interpretation int define functions $\mathcal{P}$int and $\mathcal{T}$int. They are of functionalities as shown and no function is supplied with an argument of the wrong type. []

LEMMA 6.1.2-2: Table 6.1-C and std (as augmented by 4.2-B) define a function P$\mathcal{T}$std. Similarly, Table 6.1-C and sts (as augmented by 4.2-C) define a function P$\mathcal{T}$sts. They are of functionalities as shown and no function is supplied with an argument of the wrong type. []

We now prove the "correctness" of our formulation of continuation removal, i.e. we relate $\mathcal{T}$ and P$\mathcal{T}$.

LEMMA 6.1.2-3:
$\forall$tree$\in$Tree: $\forall$std-c$\in$std-C: $\mathcal{T}$std(tree)(std-c) = std-c$\oplus$[P$\mathcal{T}$std(tree)] []
Proof is shown in appendix 6. []

LEMMA 6.1.2-4: $\forall$tree$\in$Tree: $\forall$sts-c$\in$sts-C: sts-c a complete-$\sqcup$-morphism => $\mathcal{T}$sts(tree)(sts-c) = sts-c$\oplus$[P$\mathcal{T}$sts(tree)] []
Proof essentially follows the same pattern as in Lemma 6.1.2-3 (except that case 5 is omitted). []

6.1.3 Properties of the Data-Flow Information

We now prove some properties of P$\mathcal{T}$(tree) that will be needed to validate the program transformation in subsection 6.1.4. The properties are expressed by Lemmas 6.1.3-2, 6.1.3-5 and 6.1.3-7 below.

We first show a relationship between $\lambda\underline{sta}$.P$\mathcal{T}$sts(tree)$\underline{sta}\downarrow$1 and P$\mathcal{T}$std(tree). The result bears some relationship to Lemma 4.2.1-4.

DEFINITION 6.1.3-1: Define P-G : [(std-S -c> std-R) X (sts-S -c> sts-R)] -> B by P-G(std-ss,sts-ss) = $\forall\underline{sta}\in\mathcal{P}$(Sta):
(sts-ss($\underline{sta}$))$\downarrow$1 = { (std-ss(sta)|Sta) | sta$\in\underline{sta}$ $\wedge$ [std-ss(sta)ESta]=true}
[]

LEMMA 6.1.3-2: $\forall$tree$\in$Tree: P-G(P$\mathcal{T}$std(tree), P$\mathcal{T}$sts(tree)) []
Proof is shown in appendix 6. []

COROLLARY 6.1.3-3: $\lambda\underline{sta}$. P$\mathcal{T}$sts(tree)$\underline{sta}$ $\downarrow$1 is a complete-$\sqcup$-morphism. []

We now show some simple properties of P𝒯sts(tree).

DEFINITION 6.1.3-4: Define predicates Pa:Tree->B, Pb:Tree->B and Pc:Tree->B by

```
Pa(tree) = ∀sta∈℘(Sta): (P𝒯sts(tree)sta ↓2)<root(tree).lab,"("> = sta
Pb(tree) = ∀sta: (P𝒯sts(tree)sta ↓2)<root(tree).lab,")"> = P𝒯sts(tree)sta ↓1
Pc(tree) = ∀sta∈℘(Sta): ∀lab∈Lab: ∀q∈Q:
             [ pure[Pla]<lab,q> ∧ lab∉dom(tree) =>
                 (P𝒯sts(tree)sta ↓2)<lab,q> = ∅ ]                           []
```

Predicates Pa, Pb and Pc are used to prove predicates Pd and Pe below. Predicate Pc constrains the places for which P𝒯sts(tree) can specify non-empty data flow information; this is needed when we perform proofs by structural induction.

LEMMA 6.1.3-5: ∀tree∈Tree: Pa(tree) ∧ Pb(tree) ∧ Pc(tree) []

Proof is shown in appendix 6. []

We now consider some interesting properties of P𝒯sts(tree). Together with Lemma 6.1.3-2 they are the fundament for validating program transformations (Theorem 6.1.4-2).

DEFINITION 6.1.3-6: Define predicates Pd: TreeXLab -> B and Pe: TreeXLab -> B by

```
Pd(tree,lab) = ∀sta∈℘(Sta):
                 LET a = P𝒯sts(tree)sta ↓2 IN
                 a<lab,")"> = (P𝒯sts(tree at lab) (a<lab,"(">))↓1
Pe(tree,lab) = ∀sta∈℘(Sta):
                 LET a = P𝒯sts(tree)sta ↓2 IN
                 LET str = root(tree at lab).str IN
           i)    str ∈ {"dcl1 ; dcl2", "cmd1 ; cmd2", "BEG dcl IN cmd END"} =>
                 a<lab§<1>,"("> = a<lab,"("> ∧
                 a<lab§<2>,"("> = a<lab§<1>,")">
           ii)   str = "exp1 ope exp2" =>
                 a<lab§<1>,"("> = a<lab,"("> ∧
                 a<lab§<3>,"("> = a<lab§<1>,")">
           iii)  str ∈ {"ide := exp", "WRITE exp"} =>
                 a<lab§<1>,"("> = a<lab,"(">
           iv)   str = "IF exp THEN cmd1 ELSE cmd2 FI" =>
                 a<lab§<1>,"("> = a<lab,"("> ∧
                 a<lab§<2>,"("> = sts-Dt-cond( a<lab§<1>,")"> )
                 a<lab§<3>,"("> = sts-Df-cond( a<lab§<1>,")"> )
           v)    str = "WHILE exp DO cmd OD" =>
                 LET F = [λsta. P𝒯sts( tree at lab§<2> ) sta ↓1 ] ∘
                         sts-Dt-cond ∘
                         [λsta. P𝒯sts( tree at lab§<1> ) sta ↓1 ] IN
                 a<lab§<1>,"("> = ⋃{ Fⁿ( a<lab,"("> ) | n≥0 } ∧
                 a<lab§<2>,"("> = sts-Dt-cond( a<lab§<1>,")"> )
```

Also define predicates Pd': Tree -> B and Pe': Tree -> B by

```
Pd'(tree) = ∀lab∈dom(tree):
   [ root( tree at lab ).cat ∈ {"dcl", "cmd", "exp", "pro"} => Pd(tree,lab) ]
Pe'(tree) = ∀lab∈dom(tree):
   [ root( tree at lab ).cat ∈ {"dcl", "cmd", "exp", "pro"} => Pe(tree,lab) ]
                                                                              []
```

In the sequel we do not consider parse-trees of category "ide", "ope" or "bas". This is expressed by the definitions of Pd' and Pe' above and simplifies the development.

LEMMA 6.1.3-7: ∀tree∈Tree: Pd'(tree) ∧ Pe'(tree) □
Proof is shown in appendix 6. □

6.1.4 Validity of Program Transformations

The key result of this subsection is Theorem 6.1.4-2 that gives sufficient conditions for when tree1 and tree1(lab <- tree2) are semantically equivalent. The major positive virtue is that we do *not* need to require that tree1 *at* lab and tree2 are semantically equivalent, but only that they produce the same results in the computational contexts in which they occur. This is expressed by P2 below.

DEFINITION 6.1.4-1: Define predicates P1: TreeXTreeXLabX𝒫(Sta) -> B and P2: TreeXTreeXLabX𝒫(Sta) -> B by

```
P1(t1,t2,lab,sta) =
  ∃t∈Tree(<>,"pro"):
  ∃t1',t2'∈Tree(<>,"pro")∪Tree(<>,"dcl")∪Tree(<>,"cmd")∪Tree(<>,"exp"):
  t1 = t(lab <- t1') ∧ t2 = t(lab <- t2') ∧ P2(t1,t2,lab,sta)
P2(t1,t2,lab,sta) =
  ∀sta∈(P𝒮sts(t1)sta ↓2) <lab,"(">:
  P𝒮std( t1 at lab ) sta = P𝒮std( t2 at lab) sta                      □
```

THEOREM 6.1.4-2: ∀t1,t2∈Tree: ∀lab∈Lab: ∀sta∈𝒫(Sta):
P1(t1,t2,lab,sta) => P1(t1,t2,<>,sta) □
Proof is shown in appendix 6. □

We now rephrase Theorem 6.1.4-2 so as to avoid continuation removal where possible:

THEOREM 6.1.4-3:

```
    t1∈Tree ∧ t2∈Tree ∧ lab∈Lab ∧ inp∈𝒫(Inp)∧
    ∃t∈Tree(<>,"pro")
    ∃t1',t2'∈ Tree(<>,"pro") ∪Tree(<>,"dcl") ∪Tree(<>,"cmd") ∪Tree(<>,"exp"):
    t1 = t(lab <- t1') ∧ t2 = t(lab <- t2') ∧
    ∀sta∈𝒫sts(t1)inp<lab,"(">:
    P𝒮std( t1 at lab )(sta) = P𝒮std( t2 at lab )(sta)
 => ∀inp∈inp: 𝒫std(t1)inp = 𝒫std(t2)inp                                  □
```

Proof is shown in appendix 6. □

Below we formulate a special case of Theorem 6.1.4-3 that expresses sufficient conditions for constant folding to be valid. It has been formulated so as to make it clear that each test can be verified automatically; especially if the test concVa(...)⊆... is replaced by ...⊑dabsVa(...) for dabsVa and concVa semi-down-adjoined (2.4.3). The proof is tedious (and long) and is omitted because it gives no insight.

COROLLARY 6.1.4-4:

```
Let  bas∈Bas
     tree∈Tree(<>,"pro")
     lab∈dom(tree) such that root(tree at lab).cat = "exp"
     tree' = tree(lab <- { <<>,<"exp","bas">>, <<1>,<"bas",bas>>} )
     <uabs,conc> be the pair of semi-adjoined functions defined in section
```

3.2 in terms of the pair <uabsVa,concVa> of semi-adjoined functions
If concVa(𝒫ind<uabs,conc>(tree)inp <lab,")"> ↓2↓1) ⊆ {std-ℬ⟦bas⟧}
then ∀inp∈inp: 𝒫std(tree)inp = 𝒫std(tree')inp □

When program transformations are performed in practice it is often important to know how the data flow information for the transformed program (tree') can be cheaply obtained from that of the original program (tree). Our approach makes it possible to show such relationships, e.g. (in the notation of Corollary 6.1.4-4) that pla↓1≠lab§<...> ∧ pure[Pla](pla) => 𝒫ind(tree)inp(pla) = 𝒫ind(tree')inp(pla). But we do not investigate such results.

6.2 Comparison with Other Approaches

Below, we briefly relate the approach of this chapter to other approaches considering the validity of program transformations [HuL78][Ger75]. The comments will be informal because both the aims and the semantic foundations of the papers differ from those of our development.

Comparison with [HuL78]

The aim of [HuL78] is to consider "[program] transformations based on control structure equivalence", e.g. recursion removal. The semantic foundation is different from Denotational Semantics as described in [Sto77] and [MiS76].

Consider programs t1 = t(lab <- t1') and t2 = t(lab <- t2'). The paper considers easily tested conditions for asserting that t1 and t2 are equivalent, i.e. P𝒯std(t1) = P𝒯std(t2). Clearly one can view t2 as obtained from t1 by a program transformation that replaces t1' by t2'. All the conditions are sufficient to deduce P𝒯std(t1') = P𝒯std(t2'), i.e. that the two syntactic constructs produce the same result for all states. Thus the paper does not consider the possibility of P𝒯std(t1) = P𝒯std(t2) when P𝒯std(t1')sta = P𝒯std(t2')sta only holds for the states (sta) of the computational context in which t1' and t2' occur. A consequence is that the method of [HuL78] is inadequate for validating program transformations like constant folding.

Comparison with [Ger75]

The aim of [Ger75] is to consider correctness of programs (and program transformations) with respect to an input predicate Pin and an output predicate Pout. That two programs t1 and t2 are both correct with respect to Pin and Pout does not, in general, imply that they produce the same results for inputs satisfying Pin. But in many situations (including our treatment of constant folding) it can be arranged that this is the case.

The semantic foundation is the "inductive assertions method" formulated using attribute grammars.

Assume that t1 = t(lab <- t1') and t2 = t(lab <- t2'). Programs must describe the predicates Pin and Pout as well one predicate for each WHILE loop. A predicate belonging to a WHILE loop specifies an approximate set of states expected to include the states possible at that point. Predicates Pin and Pout are denoted t1.Pin, etc. and we assume that t1.Pin = t2.Pin ∧ t1.Pout = t2.Pout.

The approach of [Ger75] is to define (in effect) P𝒯a specifying "forward attributes" and P𝒯fvc specifying "forward verification conditions".

Essentially P𝒯fa(t) is P𝒯sts(t){sta|t.Pin(sta)}↓2 so that P𝒯fa(t)<lab,"("> describes a set of states. But P𝒯fa(t)(pla) exploits the predicates described in the WHILE loops of t in order to express an (approximate) set of states that includes those holding at pla. This is in contrast with our approach where abstract interpretation is powerful enough to express this set of states without exploiting such predicates.

That P𝒯fa(t)(pla) exploits the predicates belonging to the WHILE loops of t has as a consequence that these predicates must be verified. The logical formula P𝒯fvc(t) formulates a correctness condition in terms of P𝒯fa(t). When P𝒯fvc(t) is true it is said that t is correct. - Very roughly P𝒯fvc(t) corresponds to our Pe'(t), but it is not always true (contrary to Pe'(t)) because the predicates in the program need not be satisfied.

The paper does not explicitly consider an analogue of P𝒯std. It is merely stated that P𝒯std must be so that P𝒯fvc(t) $\wedge$ t.Pin(sta) implies t.Pout(P𝒯std(t)sta) and that the predicates belonging to the WHILE loops are satisfied.

In [Ger75,p.64] there is a schematic transformation that includes constant folding. Sufficient conditions to deduce correctness of t2 are

a) t1 is correct, and

b) $\forall$sta $\in$ P𝒯fa(t1)<lab,"("> that P𝒯std(t1')(sta) = P𝒯std(t2')(sta)

As mentioned earlier, correctness of t1 and t2 can be made to imply that P𝒯std(t1) and P𝒯std(t2) are equivalent for all states satisfying t1.Pin. Thus the approach of [Ger75] can be used to validate constant folding.

The main virtue of [Ger75] is that a notion of computational context is present (see "b)" above) so that e.g. constant folding can be validated. But it is difficult to automate the method. One reason is that P𝒯fvc(t1) must be proven and that the formula often is "structurally complex, although not necessarily deep" [Ger75,p.55]. Another is that P𝒯fa requires that each WHILE loop contains a predicate in order to describe (approximately) the set of states possible inside WHILE loops.

We therefore feel that our approach is better suited to verify program transformations than that of [Ger75].

CHAPTER 7

Conclusion

There are (at least) two ways of viewing the development of this paper. The first view emphasizes the <u>formulation</u> of data flow analyses: history-insensitive analyses (chapter 3) and history-sensitive analyses (chapter 4). This view corresponds to that of [Don79]. But the theorems stated do not formally relate the data flow information $\mathcal{P}$ind⟦pro⟧ to the semantic meaning $\mathcal{P}$std⟦pro⟧. This is because of the loose connection between $\mathcal{P}$col and $\mathcal{P}$std (or $\mathcal{P}$tra and $\mathcal{P}$std). Similar remarks apply to $\mathcal{P}$his-ind and $\mathcal{P}$ae.

We have taken a second view that emphasizes the <u>formulation</u> of data flow analyses as well as a <u>relationship</u> between the data flow information and the semantics. An indication of the usefulness of $\mathcal{P}$ind even if it could not be related to $\mathcal{P}$std is achieved by relating the data flow information specified by our approach to the solutions considered in traditional data flow analysis (chapter 5). Similar remarks apply to $\mathcal{P}$his-ind and $\mathcal{P}$ae. The validation of program transformations is one way of relating $\mathcal{P}$ind to $\mathcal{P}$std (chapter 6). - We have been unable to think of others.

Our approach to formulating data flow analyses is based on expressing abstract interpretation in a denotational setting.

We hope that our motivation and analysis of the concepts from abstract interpretation gives more insight than previous motivations (section 2.4). In particular, the concept "semi-adjoined" is closer than "adjoined" to the informal approximation ideas that are considered in "traditional data flow analysis" {a}. We expect it to be possible to weaken our assumption that only complete lattices are considered. It is interesting if the isotony assumption of "semi-adjoined functions" can be weakened in such a way that Theorem 3.1-10 still holds.

The dual concepts "semi-down-adjoined" and "down-adjoined" are useful when considering program transformations, as indicated by the remarks leading up to Corollary 6.1.4-4.

The formulation of Abstract Interpretation within Denotational Semantics gives a more high-level formulation than the usual (chapter 3). That the entire development is based on semi-adjoined functions shows the usability of the concept "semi-adjoined".

From a practical point of view the material of sub-section 2.4.2 shows that pairs of semi-adjoined functions can be specified in a systematic way. This is useful because the concept of "induced interpretation" makes it

---

{a} Let uabs be an abstraction function and conc a concretization function. A consequence of requiring uabs and conc to be adjoined is that conc $\circ$ uabs must be extensive, isotone and idempotent. When uabs and conc are semi-adjoined then conc $\circ$ uabs need not be idempotent. The "informal approximation ideas" amount to conc $\circ$ uabs being extensive.

possible to specify a data flow analysis simply by giving a pair of semi-adjoined functions (between $\mathcal{P}$(Sta) and some complete lattice). This is demonstrated by the "constant propagation" example of section 3.2.

There is a problem inherent in our approach. The applicability is limited because we are unable to define reflexive powersets. Thereby language constructs as labels and procedures cannot be handled.

We hope to investigate whether the literature on power-domains contains tools that can extend the applicability of our method. Reflexive powersets amounts to defining a domain that contains its own powerset. Since this is impossible (Cantor's Theorem [Hal60]) we must exclude some elements similarly to Scott's exclusion of non-continuous functions when solving X = X -> X. In section 3.1 we mentioned that modelling the powerset by ...-c>T" excluded elements that we wanted to be there {a}.

Intuitively, in our approach we use two kinds of partial orders. Some are interpreted as in Denotational Semantics, i.e. ⊑ means "less defined than" in the sense of Scott [Sto77]. Others, e.g. those of $\mathcal{P}$(...), can maybe more naturally be thought of as "logically implies", because $\mathcal{P}$(X) is isomorphic to the set (X -t> T") of predicates on X. - These two ways of considering ⊑ are different. This gives an intuitive explanation for why e.g. the continuations of col are not continuous.

The development of this paper was based on a single table of semantic equations so that the entire development must be redone for another table of semantic equations. It should be possible to avoid this by specifying a class of semantic tables (e.g. by a grammar or an algebra) such that Theorems 3.1-6 and 3.1-10 hold in this more general case.

"Available expressions" is a forward, history-sensitive data flow analysis. We share with [CoC77a] the belief that it cannot be given any semantic characterization with respect to the static semantics (using sts). This is contrary to what holds for "constant folding". Instead we obtained a semantic characterization with respect to a more "concrete" static semantics (using his-sts in chapter 4). We believe that this extends the applicability of abstract interpretation, and that it is not crucial (but maybe helpful) that we work from a denotational semantics {b}.

Probably neither sts nor his-sts is adequate for giving a semantic characterization of the backward analysis "live variables". Presumably the role of $\mathcal{P}$his-sts will be played by a semantic function $\mathcal{P}$fut-sts that associates (sets of) continuations with places. Whether it is necessary to work with sets of continuations rather than just continuations is difficult to predict.

A set of live identifiers can be obtained from a continuation c by e.g.

r(c) = { ide∈Ide | ∃<env,inp,out,wit>∈Sta ∃val1,val2∈Val:
        c<env[val1/ide],inp,out,wit> ≠ c<env[val2/ide],inp,out,wit> }

As in section 4.2 it probably is natural to develop a semantic notion of liveness and approximate it by the syntactic notion.

{a} Recall that T" is the complete lattice ({yes,no},⊑) with no ⊑ yes.

{b} In the flow-chart view one would have to consider local analysis as well as global analysis. Alternatively, the basic blocks must contain only one operator (in one expression).

The constant propagation example of section 3.2 is similar to that of [Don79,section 7.1]. Two other data flow analyses formulated in [Don79] are determination of common subexpressions [Don79,section 7.2] and determination of invariant expressions [Don79,section 7.3]. Our approach cannot directly model any of these. One reason is that both formulations have the domain of inputs to be Q^* where $q \in Q$ identifies a symbolic input value. It would be interesting to investigate how to model this by means of abstract interpretation.

We indicate the usefulness of our approach in two ways. One is to compare the information specified by our approach to the solutions of traditional data flow analysis. The other is to validate program transformations.

To relate our approach to that of traditional data flow analysis we constructed a "traditional" data flow analysis problem from a program (chapter 5). One shortcoming of this construction is that it considers smaller basic blocks than usual. We then showed that the MOP solution to the constructed problem equals the information specified by our approach. Informally stated: our approach specifies the MOP solution.

It would be interesting to obtain a formulation that specifies the MFP solution. This could be used to formulate the work of [Ros80] in our approach, i.e. to specify elimination methods computing a solution between MFP and MOP. We conjecture that a kind of MFP solution is specified by (an augmented) interpretation ind and direct-style semantic functions corresponding to those of Table 6.1-C {a}. We expect the solution to correspond to Kildall's MFP solution [Hec77,p.173] rather than the MFP solution of section 2.3 (Kam&Ullman's MFP solution [Hec77,p.178]). If this conjecture holds it is an argument in favour of basing our approach upon continuation style semantics.

In chapters 4 and 6 we have performed continuation removal because of difficulties in conducting proofs without performing continuation removal. An exception is the use of $P\mathscr{E}$std in chapter 4 which is probably necessary to consider a semantic notion of availability.
The need to perform continuation removal does not seem to limit the applicability of our approach. This is because language constructs that makes it difficult to perform continuation removal (e.g. jumps) probably also leads to reflexive domains involving powersets, which cannot be handled by our approach.

The main reason for validating program transformations (in chapter 6) has been to remedy the undesired loose connection between $\mathscr{P}$ind (as well as $\mathscr{P}$sts and $\mathscr{P}$col) and $\mathscr{P}$std. But program transformations are also important in their own right. - We believe that to formalize the notion of correctness of data flow information one has to consider program transformations.
Presumably the approach is of wide applicability: A sufficient condition for two syntactic constructs to be replacable by one another is that they produce the same results in the computational context in which they are

----

{a} It is crucial that ind is augmented in the same way as sts is, e.g. that we continue to use ind-S rather than $\mathcal{P}$(ind-S). If $\mathcal{P}$(ind-S) is used it is possible to obtain a direct style semantics that specifies the MOP solution. This remark relates to [CoC79,Theorem 9.2.0.1] and to some extent to the functionality of Close in chapter 5.

placed. Constant folding is but one of the many program transformations that can be validated in this way.

We expect it to be considerably more difficult to validate program transformations exploiting "available expressions" information. It is probably reasonably straight-forward to exploit "live variables" information.

In summary, the main accomplishments of this paper are:

- To weaken some of the assumptions usually stated in abstract interpretation. To present a different motivation for the usual assumptions.

- To express abstract interpretation in a denotational setting. To show that a denotational semantics specifying a data flow analysis can be obtained by defining a pair of semi-adjoined functions in a rather systematic way.

- To show how "available expressions" can be characterized semantically by means of abstract interpretation. Previous treatments have only formulated "available expressions".

- To show that the information specified by our approach essentially is the MOP solution.

- To formulate correctness conditions for program transformations. These assert that for two syntactic constructs to be replacable by one another it is sufficient that they produce the same results in the computational context in which they are placed.

References

[AhU78] A.V.Aho & J.D.Ullman: "Principles of Compiler Design", Addison-Wesley, 1978.

[CoC77a] P.Cousot & R.Cousot: "Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints", Conf. Record of the 4'th ACM Symposium on Principles of Programming Languages, 1977.

[CoC77b] P.Cousot & R.Cousot: "Static determination of dynamic properties of recursive procedures", IFIP WG.2.2 Working Conf. on Formal Description of Programming Concepts, St-Andrews, Canada, North-Holland Pub. Co., 1977.

[CoC77c] P.Cousot & R.Cousot: "Automatic synthesis of optimal invariant assertions: mathematical foundations", Proc. of ACM Symp. on Artificial Intelligence & Programming Languages, SIGPLAN Notices 12, 8 (1977).

[CoC79] P.Cousot & R.Cousot: "Systematic design of program analysis frameworks", Conf. Record of the 6'th ACM Symposium on Principles of Programming Languages, 1979.

[CoH78] P.Cousot & N.Halbwachs: "Automatic discovery of linear restraints among variables of a program", Conf. Record of the 5'th ACM Symposium on Principles of Programming Languages, 1978.

[Cou79] P.Cousot: "Semantic Foundations of Program Analysis", to appear in [JoM81].

[Don78] V.Donzeau-Gouge: "Utilisation de la Semantique Denotationelle Pour l'Etude d'Interpretations Non-Standard", Rapport de Recherche No 273, INRIA, Rocquencourt, Le Chesnay, France.

[Don79] V.Donzeau-Gouge: "Denotational Definition of Properties of Program Computations", to appear in [JoM81]. See also Rapport de Recherche No 349, INRIA, Rocquencourt, Le Chesnay, France.

[Ger75] S.L.Gerhart: "Correctness-Preserving Program Transformations", Conf. Record of the 2'nd ACM Symposium on Principles of Programming Languages, 1975.

[Gor79] M.J.C.Gordon: "The Denotational Description of Programming Languages: An Introduction", Springer Verlag, (1979).

[Grä71] G.Grätzer: "Lattice Theory - First Concepts and Distributive Lattices", W.H.Freeman and Company, San Francisco, 1971.

[Hal60] P.R.Halmos: "Naive Set Theory", D. van Nostrand Company (1960).

[Hec77] M.S.Hecht: "Flow Analysis of Computer Programs", North-Holland, New York, 1977.

[HuL78] G.Huet & B.Lang: "Proving and Applying Program Transformations Expressed with Second-Order Patterns", Acta Informatica 11, 1978.

References

[JoM78] N.D.Jones & S.S.Muchnick: "Complexity of Flow Analysis, Inductive Assertions and A Language Due to Dijkstra", to appear in [JoM81]. See also TR-78-4, Department of Computer Science, The University of Kansas, Lawrence, Kansas, USA.

[JoM81] N.D.Jones & S.S.Muchnick: "Program Flow Analysis: Theory and Applications", Prentice Hall, to appear in 1981.

[Jon80] N.D.Jones, personal communication, (1980).

[KaU77] J.B.Kam & J.D.Ullman: "Monotone Data Flow Analysis Frameworks", Acta Informatica 7, 1977.

[Kil73] G.A.Kildall: "A unified approach to global program optimization", Conf. Record of ACM Symposium on Principles of Programming Languages, 1973.

[Mos79] P.D.Mosses: "SIS - Semantics Implementation System: Reference Manual and User Guide", MD-30, DAIMI, Computer Science Department, Aarhus University, Denmark.

[MiS76] R.Milne & C.Strachey: "A theory of programming language semantics", Chapman and Hall, London, 1976.

[Nie79] F.Nielson: "Compiler Writing Using Denotational Semantics", TR-10, DAIMI, Computer Science Department, Aarhus University, Denmark.

[Nie80] F.Nielson, letter to V.Donzeau-Gouge, (1980).

[Ros73] B.K.Rosen: "Tree-Manipulating Systems and Church-Rosser Theorems", Journal of the ACM 20, 1973.

[Ros77] B.K.Rosen: "High-Level Data Flow Analysis", Communications of the ACM, 20, 10, (1977).

[Ros79] B.K.Rosen: "Degrees of Availability as an Introduction to the General Theory of Data Flow Analysis", to appear in [JoM81].

[Ros80] B.K.Rosen: "Monoids for Rapid Data Flow Analysis", SIAM J. Comput., Vol.9, No.1, Feb. 1980.

[San73] J.G.Sanderson: "The Lambda Calculus, Lattice Theory and Reflexive Domains", Mathematical Institute Lecture Notes, University of Oxford (1973).

[Sin72] M.Sintzoff: "Calculating Properties of Programs by Valuations on Specific Models", Proceedings of an ACM Conference on Proving Assertions about Programs, SIGPLAN Notices vol. 7, no. 1 (1972).

[Sto77] J.E.Stoy: "Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory", MIT Press, Cambridge, Massachusetts, USA, 1977.

References

[Tar55] A.Tarski: "A Lattice-Theoretical Fixpoint Theorem and its Applications", Pacific Journal of Mathematics, 5 (1955).

[Ten76] R.D.Tennent: "The Denotational Semantics of Programming Languages", Communications of the ACM, 19, 8 (1976).

[Ull75] J.D.Ullman: "A Survey of Data Flow Analysis Techniques", Second USA-JAPAN Computer Conference, 1975.

[War42] M.Ward: "The closure operators of a lattice", Annals of Mathematics, 2'nd Series, vol.43 (1942).

[Weg75] B.Wegbreit: "Property Extraction in Well-Founded Property Sets", IEEE Transactions on Software Engineering, vol. SE-1, no. 3, 1975.

APPENDIX 1

The development of subsection 2.4.3 is not the only way to move the test $l \sqsubseteq lp$ to M. Assume that L is a power-set, e.g. L = $\mathcal{P}$(INT) where INT is the integers, and that COMP(l) = {i| i$\notin$l}. Further assume that uabs and conc are semi-adjoined between L and M and that conc is a strict $\sqcap$-morphism. Then a different way of testing $l \sqsubseteq lp$ avoiding the use of dabs is to test uabs(l) $\sqcap$ uabs(COMP(lp)) = $\perp$ in M [Jon80]. To see this:

uabs(l) $\sqcap$ uabs(COMP (lp)) = $\perp$ =>
(conc $\circ$ uabs)(l) $\sqcap$ (conc $\circ$ uabs)(COMP(lp)) = $\emptyset$ =>
l $\sqcap$ (COMP(lp)) = $\emptyset$ =>
l $\sqsubseteq$ lp

But sometimes this method is too coarse as may be seen from the following example: Let M = $\mathcal{P}${a,b} and conc{a,b} = INT, conc{a} = {0}, conc{b} = {i |i>0} and conc($\emptyset$) = $\emptyset$. Also uabs = λl'.$\sqcap${m |conc(m) $\sqsupseteq$ l'} so that uabs and conc are adjoined. When l = lp = {i |i>0} then uabs(l) $\sqcap$ uabs(COMP(lp)) $\neq$ $\emptyset$. In our approach we could have used dabs(lp) = {b} so that uabs(l) $\sqsubseteq$ dabs(lp).

To investigate the connection between our method and that of [Jon80] we assume that uabs and conc are adjoined and that M (as well as L) is a power-set and that conc is strict. It is easy to see that the method of [Jon80] amounts to dabs(l') = COMP(uabs(COMP(l'))) and testing uabs(l) $\sqsubseteq$ dabs(lp). This is a safe test because <dabs,conc> is semi-down-adjoined: Clearly dabs is isotone and dabs(l') $\sqsupseteq$ m <=> uabs(COMP(l')) $\sqcap$ m = $\emptyset$ => (conc $\circ$ uabs)(COMP(l')) $\sqcap$ conc(m) = $\emptyset$ => COMP(l') $\sqcap$ conc(m) = $\emptyset$ => l' $\sqsupseteq$ conc(m) so that conc $\circ$ dabs is reductive.

But <dabs,conc> need not be quasi-down-adjoined as follows from the previous example, where dabs(lp) = COMP(uabs(COMP(lp))) =$\emptyset$ even though conc{b} $\sqsubseteq$ lp. If we also assume that conc(COMP(m)) = COMP(conc(m)) then we can show that <dabs, conc> is a pair of down-adjoined functions:

dabs(l')$\sqsupseteq$m <=> COMP(uabs(COMP(l')))$\sqsupseteq$m <=> uabs(COMP(l')) $\sqsubseteq$ COMP(m) <=>
COMP(l') $\sqsubseteq$ conc(COMP(m)) <=> COMP(l') $\sqsubseteq$ COMP(conc(m)) <=> l' $\sqsupseteq$ conc(m)

By Lemma 2.4.3-11 this definition of dabs is "adequate" under the stated assumptions: L and M power-sets, uabs and conc adjoined, $\forall$m: conc(COMP(m)) = COMP(conc(m)) (which implies that conc is strict).

APPENDIX 2

PROOF OF 2.1.2-4:

1) (f complete-$\sqcup$-morphism) => (f continuous) $\wedge$ (f $\sqcup$-morphism) $\wedge$ ($f(\bot)=\bot$) is obvious from the definitions.

2) If $l1 \sqsubseteq l2$ then $\{l1,l2\}$ is a directed, finite, and non-empty set. If f is continuous or a $\sqcup$-morphism then $l1 \sqsubseteq l2$ implies $f(l1) \sqsubseteq f(l1) \sqcup f(l2) = f(l1 \sqcup l2) = f(l2)$ so f is isotone.

3) Define f:T->T by $f(\bot)$ = f(false) = f(true) = $\bot$ and $f(\top)=\top$. Then f is continuous but not a $\sqcup$-morphism.

4) Define $[0,1]=\{x \mid x \text{ real} \wedge x \geq 0 \wedge x \leq 1\}$ and $[0,1[=\{x \in [0,1] \mid x \neq 1\}$. Then $([0,1],\leq)$ is a complete lattice and $[0,1[$ is a directed set. Define T"=({no,yes},$\sqsubseteq$) with $t1 \sqsubseteq t2$ <=> ($t1 \neq$ yes $\vee$ $t2 \neq$ no) and f:[0,1]->T" by f(x)=(x<1->no,yes). Then f is a $\sqcup$-morphism but not continuous: $\bigsqcup\{ f(x) \mid x \in [0,1[\}$ = no $\neq$ yes = f(1) = $f(\bigsqcup [0,1[)$.

5) Assume f is continuous, a $\sqcup$-morphism and $f\bot=\bot$. Let $\underline{l} \subseteq L$. Define $\underline{l}' = \{\bigsqcup \underline{l}'' \mid \underline{l}'' \subseteq \underline{l} \wedge \underline{l}''$ finite$\}$ so $\underline{l}'$ is a directed set (using Lemma 2.1.1-7) with $\bigsqcup \underline{l}' = \bigsqcup \underline{l}$ (using Lemma 2.1.1-7). Then

$f(\bigsqcup \underline{l}) = f(\bigsqcup \underline{l}')$
$= \bigsqcup\{f(l) \mid l \in \underline{l}'\}$ since f is continuous
$= \bigsqcup\{f(\bigsqcup \underline{l}'') \mid \underline{l}'' \subseteq \underline{l} \wedge \underline{l}''$ finite$\}$
$= \bigsqcup\{\bigsqcup\{ f(l) \mid l \in \underline{l}''\} \mid \underline{l}'' \subseteq \underline{l} \wedge \underline{l}''$ finite $\wedge$ $\underline{l}'' \neq \emptyset\}$
since $f\bot=\bot$ and f is a $\sqcup$-morphism
$= \bigsqcup\{f(l) \mid l \in \underline{l}\}$ by Lemma 2.1.1-7. []

PROOF OF 2.1.3-2:
[San73,p.27-28] proves (for n=2) that L1X...XLn is a complete lattice and that $\bigsqcup$ is as shown. Dual computations give the form of $\sqcap$. To see that $\downarrow i$ is continuous: Let $\underline{l} \subseteq$ L1X...XLn be directed. Then
$(\bigsqcup \underline{l})\downarrow i = \langle \ldots, \bigsqcup\{l'\downarrow i \mid l' \in \underline{l}\}, \ldots\rangle \downarrow i = \bigsqcup\{l'\downarrow i \mid l' \in \underline{l}\}$.

Many of the following lemmas are (like this one) proved in the literature (e.g. [Sto77], [San73], [MiS76]). The proofs therefore are omitted. []

PROOF OF 2.1.3-13:
To see that L-t>M, L-i>M and L-c>M are partially ordered sets is easy (for anti-symmetry use the axiom of extensionality [Sto77,p.56]). It is straight-forward to show that L-t>M and L-i>M are complete lattices with $\sqcup$ and $\sqcap$ as shown. Also one can show that L-c>M is a complete lattice with $\sqcup$ as shown (using Lemma 2.1.1-7 twice) and $\sqcap \underline{f} = \bigsqcup\{g \mid g \in$ L-c>M $\wedge \forall f' \in \underline{f}: f' \sqsupseteq g\}$.

To see that $\bigsqcup\{g \mid g \in$ L-c>M $\wedge \forall f' \in \underline{f}: f' \sqsupseteq g\}$ need not be $\lambda l. \sqcap\{f(l) \mid f \in \underline{f}\}$ define L = $\{\bot,\top\} \cup \{$ <"a",i>, <"b",i> $\mid i>0\}$ with $l1 \sqsubseteq l2$ <=> ($l1=\bot$ $\vee$ $l2=\top$ $\vee$ ($l1\downarrow 1=l2\downarrow 1$ $\wedge$ $l1\downarrow 2 \leq l2\downarrow 2$)). Define f:L->L by $f(\bot)=\bot$, f<"a",i>=f<"b",i>=<"a",i> and $f(\top)=\top$. Define g:L->L by $g(\bot)=\bot$, g<"a",i>=g<"b",i>=<"b",i> and $g(\top)=\top$. Obviously, f and g are continuous. But h = $\lambda l.f(l) \sqcap g(l)$ = $\lambda l.l \neq \top$-> $\bot$, $\top$ is not continuous:
h($\bigsqcup\{$<"a",i> $\mid i>0\}$) = $\top \neq \bot$ = $\bigsqcup\{$h<"a",i> $\mid i>0\}$). []

PROOF OF 2.2-4:
The proof is by structural induction [Sto77]. Define the predicates:

P-Dcl:Dcl->B by P-Dcl(dcl) = 𝒟int⟦dcl⟧ is well-defined (and in the clause for 𝒟int⟦dcl⟧ no function is supplied with an argument of the wrong type) and 𝒟int⟦dcl⟧ is of functionality as shown.

P-Exp, P-Cmd and P-Pro are defined similarly.

Clearly the predicates are well-defined, since they are not defined recursively in terms of each other [Sto77]. We could be more precise about "argument of the wrong type" but there is little point in doing so.

Structural Induction on Dcl:
The case 'DCL ide := bas': First note that (the last) attach is supplied with arguments (<occ,"dcl)"> and c) of the right types and that attach<occ,"dcl)">;c is well-defined. Then note that assign is supplied with arguments of the right types and that assign⟦ide⟧;attach<occ,"dcl)">;c is well-defined. Then note that push is supplied with arguments of the right types and that push⟦bas⟧;assign⟦ide⟧;attach<occ,"dcl)">;c is well-defined. Finally note that (the first) attach is supplied with arguments of the right types and that 𝒟int⟦DCL ide:=bas⟧ occ c is well-defined. Furthermore no function in the clause for 𝒟int⟦DCL ide:=bas⟧ occ c is supplied with an argument of the wrong type. Also Lemma 2.1.2-8 yields
𝒟int⟦DCL ide:=bas⟧ ∈ Occ -c> C -c> C.
The case 'dcl1;dcl2' is similar.

Structural Induction on Exp: As for Dcl above.

Structural Induction on Cmd: The case 'IF exp THEN cmd1 ELSE cmd2 FI'. Let g1,g2 ∈ C-c>C and cond ∈ C×C-c>C. Then h=λc. cond(g1;c, g2;c) is easily seen to be continuous. From this P-Cmd(IF exp THEN cmd1 ELSE cmd2 FI) follows.
The case 'WHILE exp DO cmd OD': Let g∈C-c>C and cond∈C×C-c>C and h[n] = λc.(λc'. cond(g;c', c))^n⊥. Then one can show ∀n≥0: h[n]∈C-c>C so that λc.FIX(λc'.cond(g;c',c)) = λc.⊔{ h[n];c |n≥0} = ⊔{ h[n] | n≥0 } ∈ C -c> C because of Lemma 2.1.3-13(3). From this follows that P-Cmd(WHILE exp DO cmd OD).
The remaining cases are straight-forward.

Structural Induction on Pro: This is by hypothesis and requirements upon setup. []

PROOF OF 2.2-5:
a) As there are no reflexive domains in Tables 2.2-B and 2.2-D the domains (including C, I, A and S of Table 2.2-D) obviously exist and are complete lattices.

b) Obviously wrong∈C and finish∈C.

c&d) We only prove assign ∈ Ide -t> C -c> C as the remaining proofs are more or less similar to this proof.

1) We show Vassign ∈ Ide -t> Sta -c> T.
It is convenient to write Vassign⟦ide⟧sta= (#sta↓4)<<1 ->false,true. By lemmas of section 2.1 and our assumption that << is continuous it easily follows that Vassign⟦ide⟧ ∈ Sta-c>T.

2) We can show Bassign $\in$ Ide -t> Sta -c> Sta similarly to above.

3) We now show assign $\in$ Ide -t> C -c> C.
Obviously assign $\in$ Ide -t> C -t> Sta -t> A. It is easy to show assign $\in$ Ide -t> C -c> Sta -t> A and using "1)" and "2)" and results like 2.1.2-8 it is straight-forward to show assign $\in$ Ide -t> C -c> Sta -c> A.
□

PROOF OF 2.4.2-1:
Let <-> be => if "semi-adjoined" and <=> if "adjoined". Then
$\forall l1 \in L1\ \forall l[n+1] \in L[n+1]$:

$\quad$ uabs[n](...(uabs1(l1)) $\sqsubseteq$ l[n+1]
$\quad$ <-> uabs[n-1](...(uabs1(l1))) $\sqsubseteq$ conc[n](l[n+1])
$\quad$ ...
$\quad$ <-> l1 $\sqsubseteq$ conc1(...(conc[n](l[n+1])))

In the case of "adjoined" the result obviously follows (by Definition 2.4.1-6). In the case of "semi-adjoined" the result follows by Observation 2.4.1-5 because uabs[n]$\circ$...$\circ$uabs1 and conc1$\circ$...$\circ$conc[n] clearly are isotone.

PROOF OF 2.4.2-3:
$\forall \underline{l} \in \mathcal{P}(L1 \times \ldots \times Ln)\ \forall m \in \mathcal{P}(L1) \times \ldots \times \mathcal{P}(Ln)$:

$\text{uabs}\times(\underline{l}) \sqsubseteq m \iff \forall i \in \{1,\ldots,n\}: \{l\downarrow i \mid l \in \underline{l}\} \subseteq m\downarrow i$
$\qquad \iff \underline{l} \subseteq \{\langle l1,\ldots,ln\rangle \mid \forall i \in \{1,\ldots,n\}: li \in m\downarrow i\}$
$\qquad \iff \underline{l} \subseteq \text{conc}\times(m)$ □

PROOF OF 2.4.2-9:
We have $\forall \underline{l} \in \mathcal{P}(L1+\ldots+Ln)\ \forall m \in \mathcal{P}(L1)+\ldots+\mathcal{P}(Ln)$:

1) If $\underline{l} = \emptyset$:
$\quad \text{uabs+}(\underline{l}) \sqsubseteq m \iff \bot \sqsubseteq m \iff \text{true} \iff \emptyset \subseteq \text{conc+}(m) \iff l \subseteq \text{conc+}(m)$

2) If $\forall l \in \underline{l}\ \exists li \in Li: l = li \text{ in}L1+\ldots+Ln$ and $\underline{l} \neq \emptyset$ then:
$\quad \text{uabs+}(\underline{l}) \sqsubseteq m \iff \{ li \mid li \text{ in}L1+\ldots+Ln \in \underline{l} \} \text{ in}\mathcal{P}(L1)+\ldots+\mathcal{P}(Ln) \sqsubseteq m$
$\qquad \iff (m=\top) \vee (m\downarrow 1 = i \wedge \{li \mid li \text{ in}L1+\ldots+Ln \in \underline{l}\} \subseteq m\downarrow 2)$
$\qquad$ (this was by Definition 2.1.3-4)
$\qquad \iff (m=\top) \vee (m\downarrow 1 = i \wedge \underline{l} \subseteq \{ li \text{ in}L1+\ldots+Ln \mid li \in m\downarrow 2\})$
$\quad$ if $m=\top$ then $\iff \text{true} \iff \underline{l} \subseteq L1+\ldots+Ln \iff \underline{l} \subseteq \text{conc+}(m)$
$\quad$ if $m=\bot$ then $\iff \text{false} \iff \underline{l} \subseteq \emptyset \iff \underline{l} \subseteq \text{conc+}(m)$ (as $\underline{l}\neq\emptyset$)
$\quad$ if $m = \underline{l}j \text{ in}\mathcal{P}(L1)+\ldots+\mathcal{P}(Ln)$ where $\underline{l}j \subseteq Lj$ then
$\qquad \iff j=i \wedge \underline{l} \subseteq \text{conc+}(m) \iff \underline{l} \subseteq \text{conc+}(m)$ (as $\underline{l}\neq\emptyset$)

3) Otherwise:
$\quad \text{uabs+}(\underline{l}) \sqsubseteq m \iff \top \sqsubseteq m$
$\quad$ if $m=\top$ then $\iff \text{true} \iff \underline{l} \subseteq \text{conc+}(m)$
$\quad$ if $m=\bot$ then $\iff \text{false} \iff \underline{l} \subseteq \text{conc+}(m)$ (as $\underline{l}\neq\emptyset$)
$\quad$ if $m = \underline{l}j \text{ in}\mathcal{P}(L1)+\ldots+\mathcal{P}(Ln)$ then $\iff \text{false} \iff \underline{l} \subseteq \text{conc+}(m)$
$\qquad$ since $\text{NON}(\exists i \forall l \in \underline{l} \exists li \in Li: l = li \text{ in}L1+\ldots+Ln)$
$\qquad$ and $\forall l \in \text{conc+}(m) \exists lj \in Lj: l = lj \text{ in}L1+\ldots+Ln$ □

PROOF OF 2.4.2-11:
Let <-> be <=> if <uabs,conc> is a pair of adjoined functions and => if <uabs,conc> is a pair of semi-adjoined functions. Then for $l \in L1+\ldots+Li+\ldots+Ln$ and $m \in L1+\ldots+Mi+\ldots+Ln$:

If $l=\bot$:
$\quad (\text{Si uabs } l) \sqsubseteq m \iff \bot \sqsubseteq m \iff \bot \sqsubseteq (\text{Si conc } m) \iff l \sqsubseteq (\text{Si conc } m)$

If $l = lj \text{ in}L1+\ldots+Ln$ and $j \neq i$:
$\quad (\text{Si uabs } l) \sqsubseteq m$
$\quad \iff lj \text{ in}L1+\ldots+Mi+\ldots+Ln \sqsubseteq m$
$\quad \iff (m=\top) \vee (\exists lj': m = lj' \text{ in}L1+\ldots+Mi+\ldots+Ln \wedge lj \sqsubseteq lj')$

$\Leftrightarrow l \sqsubseteq (Si\ conc\ m) \vee (\exists lj': m = lj'\ inL1+...+Mi+...+Ln \wedge l \sqsubseteq (Si\ conc\ m))$

$\Leftrightarrow l \sqsubseteq (Si\ conc\ m)$

If $l = li$ in $L1+...+Ln$:

$(Si\ uabs\ l) \sqsubseteq m \Leftrightarrow (\ uabs(li)\ inL1+...+Mi+...+Ln\ ) \sqsubseteq m$

$\Leftrightarrow (m=\tau) \vee (\exists mi: m = mi\ inL1+...+Mi+...+Ln \wedge uabs(li) \sqsubseteq mi)$

$\Leftrightarrow (m=\tau) \vee (\exists mi: m = mi\ inL1+...+Mi+...+Ln \wedge li \sqsubseteq conc(mi))$

$\Leftrightarrow (l \sqsubseteq (Si\ conc\ m)) \vee (\exists mi: m = mi\ inL1+...+Mi+...+Ln \wedge l \sqsubseteq (Si\ conc\ m))$

$\Leftrightarrow l \sqsubseteq (Si\ conc\ m)$

If $l=\tau$:

$(Si\ uabs\ l) \sqsubseteq m \Leftrightarrow \tau \sqsubseteq m \Leftrightarrow \tau \sqsubseteq (Si\ conc\ m) \Leftrightarrow l \sqsubseteq (Si\ conc\ m)$

In the case of "adjoined" the result obviously follows (by 2.4.1-6). In the case of "semi-adjoined" the result follows by 2.4.1-5 because (Si uabs) and (Si conc) clearly are isotone. []

PROOF OF 2.4.3-3:

Suppose $\langle uabs, conc\rangle$ is a pair of adjoined functions between L and M.
Clearly $uabs = \lambda l.\sqcap\{m \mid uabs(l) \sqsubseteq m\} = \lambda l.\sqcap\{m \mid l \sqsubseteq conc(m)\}$.
Now $uabs(\sqcup \underline{L}) = \sqcap\{m \mid \sqcup\underline{L} \sqsubseteq conc(m)\} = \sqcap\{m \mid \forall l \in \underline{L}: l \sqsubseteq conc(m))\}$
$= \sqcap\{m \mid \forall l \in \underline{L}: uabs(l) \sqsubseteq m\} = \sqcap\{m \mid \sqcup\{uabs(l) \mid l \in \underline{L}\} \sqsubseteq m\}$
$= \sqcup\{uabs(l) \mid l \in \underline{L}\}$
so that uabs is a complete-$\sqcup$-morphism and hence isotone (2.1.2-4).

Similarly, $conc = \lambda m.\sqcup\{l \mid uabs(l) \sqsubseteq m\}$ is a complete-$\sqcap$-morphism. It is easy to see that conc is isotone.

So $\langle uabs, conc\rangle$ is pair of semi-adjoined functions. To see that it is a pair of quasi-adjoined functions: $conc \circ uabs = \lambda l.\sqcap conc(\ \{m \mid l \sqsubseteq conc(m)\}) = \lambda l.\ \sqcap\{conc(m) \mid l \sqsubseteq conc(m)\}$. []

PROOF OF 2.4.3-10:

Since (semi-,quasi-) down-adjoined is the dual concept of (semi-,quasi-) adjoined the proof is dual to that of 2.4.3-3. []

PROOF OF 2.4.3-11:

1) We show: $(\exists m,mp: m \sqsubseteq mp \wedge l \sqsubseteq conc(m) \wedge conc(mp) \sqsubseteq lp) \Leftrightarrow uabs(l) \sqsubseteq dabs(lp)$

$\Leftarrow$: Obvious with $m = uabs(l) \wedge mp = dabs(lp)$

$\Rightarrow$: Assume $m \sqsubseteq mp \wedge l \sqsubseteq conc(m) \wedge conc(mp) \sqsubseteq lp$. Since uabs and conc are quasi-adjoined we get

$conc(uabs(l)) = \sqcap\{conc(m') \mid conc(m') \sqsupseteq l\} \sqsubseteq conc(m)$

and from dabs and conc quasi-down-adjoined we get

$conc(dabs(lp)) = \sqcup\{conc(m') \mid conc(m') \sqsubseteq lp\} \sqsupseteq conc(mp)$

so

$l \sqsubseteq conc(uabs(l)) \sqsubseteq conc(m) \sqsubseteq conc(mp) \sqsubseteq conc(dabs(lp)) \sqsubseteq lp$

If uabs and conc are adjoined then $l \sqsubseteq conc(dabs(lp))$ gives $uabs(l) \sqsubseteq dabs(lp)$. If dabs and conc are down-adjoined then $conc(uabs(l)) \sqsubseteq lp$ gives $uabs(l) \sqsubseteq dabs(lp)$.

2) We show $conc(uabs(l)) \sqsubseteq conc(dabs(lp)) \Leftrightarrow uabs(l) \sqsubseteq dabs(lp)$.

$\Leftarrow$: Trivial

$\Rightarrow$: If uabs and conc are adjoined: $conc(uabs(l)) \sqsubseteq conc(dabs(lp)) \Rightarrow l \sqsubseteq conc(dabs(lp)) \Rightarrow uabs(l) \sqsubseteq dabs(lp)$. Similarly if dabs and conc are down-adjoined: $conc(uabs(l)) \sqsubseteq conc(dabs(lp)) \Rightarrow conc(uabs(l)) \sqsubseteq lp \Rightarrow$

uabs(l) $\sqsubseteq$ dabs(lp). □

PROOF OF 2.4.3-12:
We know that conc $\circ$ uabs is isotone and extensive and it is easy to show that conc $\circ$ uabs is idempotent (in fact conc $\circ$ uabs $\circ$ conc = conc); so conc $\circ$ uabs is an upper closure operator. Obviously {conc(uabs(l)) | l$\in$L} $\subseteq$ {conc(m) | m$\in$M}. Conversely, let conc(m) $\in$ {conc(m') | m'$\in$M}. From conc $\circ$ uabs $\circ$ conc = conc follows conc(m) = conc(uabs(conc(m))) $\in$ {conc(uabs(l)) | l$\in$L}.

To show that the upper closure operator is unique, we let uco1 and uco2 be two upper closure operators with {uco1(l) | l$\in$L} = {uco2(l) | l$\in$L} = {conc(m) | m$\in$M}. Then for any l$\in$L: l$\sqsubseteq$uco1(l) => uco2(l) $\sqsubseteq$ uco2(uco1(l)) = uco1(l) since uco2 is idempotent and uco1(l) $\in$ {uco2(l') | l'$\in$L} so that uco2(l) $\sqsubseteq$ uco1(l). Conversely uco2(l) $\sqsupseteq$ uco1(l). This shows uco1 = uco2, i.e. uniqueness. □

PROOF OF 2.4.3-13:
Define NUM = {0,1,...} and POS = {1,2,...} and let L=$\mathcal{P}$(NUM). Define uco:L->L by uco(l) = IF l is finite THEN l ELSE NUM. Then uco is an upper closure operator. Define $\underline{l}$$\subseteq$L as { l$\subseteq$NUM |(l is finite)$\wedge$ 0$\notin$l}. Then $\underline{l}$ is directed and $\sqcup\underline{l}$ = POS. But uco($\sqcup\underline{l}$) = uco(POS) = NUM $\neq$ POS = $\sqcup\underline{l}$ = $\sqcup${ uco(l) | l$\in\underline{l}$}. So uco is not continuous. □

APPENDIX 3

PROOF OF 3.1-2:

1) Since there are no reflexive domains the domains (including S, I, A and C) obviously exist and are complete lattices.

2) Obviously, wrong $\in$ C and finish $\in$ C.

3) We only prove g $\in$ Par -t> C -c> C for primitive functions as the proofs are tedious and the cases of setup, attach, and cond more or less are like this proof.

   a) Dg $\in$ Par -t> $\mathcal{P}$(Sta) -c> $\mathcal{P}$(Sta)
   Clearly Dg(par) is well-defined. It is a complete-$\sqcup$-morphism because for any $\underline{s} \in \mathcal{P}(S) = \mathcal{P}(\mathcal{P}(Sta))$: Dg(par)($\sqcup\underline{s}$)

   $= \{$std-Bg(par)(sta) | std-Vg(par)(sta)=true $\wedge$ sta$\in(\sqcup\underline{s})\}$

   $= \{$std-Bg(par)(sta) | std-Vg(par)(sta)=true $\wedge$ $\exists\underline{sta}\in\underline{s}$: sta$\in\underline{sta}\}$
   (this was by $\sqcup$ being $\cup$)

   $= \sqcup\{\{$std-Bg(par)(sta) | std-Vg(par)(sta)=true $\wedge$ sta$\in\underline{sta}\}$ | $\underline{sta}\in\underline{s}\}$
   (this was by $\sqcup$ being $\cup$)

   $= \sqcup\{$ Dg(par)($\underline{sta}$) | $\underline{sta}\in\underline{s}\}$

   b) Clearly g(par) is well-defined. If $\underline{c}\subseteq$C then

   g(par)($\sqcup\underline{c}$) = ($\sqcup\underline{c}$)$\circ$(Dg(par))
   $= \sqcup\{$c $\circ$ Dg(par) | c$\in\underline{c}\}$
   $= \sqcup\{$g(par)(c) | c$\in\underline{c}\}$

   So g(par) is a complete-$\sqcup$-morphism and hence g $\in$ Par -t> C -c> ($\mathcal{P}$(Sta) -t> Pla -t> $\mathcal{P}$(Sta)). Since Dg(par) and c$\in$C are continuous Lemma 2.1.2-8 yields
   g $\in$ Par -t> C -c> ($\mathcal{P}$(Sta) -c> Pla -t> $\mathcal{P}$(Sta)). Finally,
   $\forall$c$\in$C $\forall\underline{sta}\in\mathcal{P}$(Sta): c($\underline{sta}$) $\in$ Pla -c> $\mathcal{P}$(Sta) so that g $\in$ Par -t> C -c> C$\square$

PROOF OF 3.1-6:

To prove the theorem we need the predicates
P-S: $\mathcal{P}$(Sta)->B where P-S($\underline{sta}$) = $\forall$sta$\in\underline{sta}$:topfree[Sta](sta)
P-C:(col-C$\times$sts-C)->B where P-C(col-c,sts-c) = $\forall\underline{sta}$:
  [P-S($\underline{sta}$) => sts-c($\underline{sta}$) = $\sqcup\{$col-c(sta) | sta$\in\underline{sta}\}$]
The proof is by structural induction (case 5 below), but we first show some auxiliary results:

1) For any primitive function g $\in$ Par -t> C -c> C we show that if P-C(col-c,sts-c) then P-C(col-g(par)(col-c), sts-g(par)(sts-c)).

   i) We first show topfree[Sta](sta) => std-Vg(par)(sta)$\neq\top$. We only consider g=assign as the remaining cases are similar. Assume topfree[Sta]<env,inp,out,wit>. Then #wit$\neq\top$ by property of # and (#wit<<1)$\neq\top$ by property of <<. As false$\neq\top$ and true$\neq\top$ and $\perp\neq\top$ the result follows.

   ii) We then show P-S($\underline{sta}$) => P-S(sts-Dg(par)($\underline{sta}$)). It suffices to show topfree[Sta](sta) $\wedge$ std-Vg(par)(sta)=true =>

topfree[Sta](std-Bg(par)(sta)). Let sta = <env,inp,out,wit> and assume std-Vg(par)(sta) = true as well as topfree[Sta](sta).

a) If g = assign: Clearly topfree[Wit](<val>§wit) => topfree[Wit](wit). Also topfree[Val](val) ∧ topfree[Env](env) ∧ ide∉{⊥,⊤} yield topfree[Env](env[val/ide]). To see this assume topfree[Ide°](ide'). Then env[val/ide]ide'∈{⊥,val,env⟦ide'⟧} because Assumption 2.2-3 and == continuous implies that (ide==ide')∈ {⊥,true,false}. This shows topfree[Sta](std-Bassign⟦ide⟧sta).

b) The cases content, read and write are similar to the above case (but simpler). The cases apply and push use Assumption 3.1-4.

iii) Finally we show the result for primitive functions. Assume P-S($\underline{\text{sta}}$) and P-C(col-c,sts-c). Then
sts-g(par)(sts-c)($\underline{\text{sta}}$)
 = sts-c(sts-Dg(par)($\underline{\text{sta}}$)) (and by "ii" and P-C(col-c,sts-c):)
 = ⊔{(col-c)(col-Bg(par)(sta)) |sta∈$\underline{\text{sta}}$ ∧ std-Vg(par)(sta)=true}
 = ⊔{col-g(par)(col-c)(sta) | sta∈$\underline{\text{sta}}$ ∧ std-Vg(par)(sta)=true}
 = ⊔{col-g(par)(col-c)(sta) | sta∈$\underline{\text{sta}}$} (which was by "i" and l⊔⊥=l)

2) Similarly to "1)" we can prove P-C(col-c1,sts-c1)∧ P-C(col-c2,sts-c2) => P-C(col-cond(col-c1,col-c2), sts-cond(sts-c1,sts-c2)). In step "i)" we show that if topfree[Sta](sta) then
std-Vcond(sta)=true => col-Scond(sta)≠⊤ and col-Vcond(sta)≠⊤

3) Similarly to "1)" we can prove P-C(col-c,sts-c) =>
P-C(col-attach(pla)(col-c), sts-attach(pla)(sts-c))

4) Similarly to "1)" we can prove
[P-C(col-c,sts-c)∧ ∀inp∈$\underline{\text{inp}}$: topfree[Inp]inp] =>
sts-setup(sts-c)$\underline{\text{inp}}$ = ⊔{col-setup(col-c)(inp) |inp∈$\underline{\text{inp}}$}.

5) We now perform the structural induction. We define the predicates P-Cmd:Cmd->B by P-Cmd(cmd)=
  [P-C(col-c,sts-c) => P-C(𝒞col⟦cmd⟧occ col-c, 𝒞sts⟦cmd⟧occ sts-c)]
and P-Exp:Exp->B and P-Dcl:Dcl->B similarly.

Structural Induction on Dcl: This is straight-forward using "1)" to "4)".

Structural Induction on Exp: As above.

Structural Induction on Cmd: Most cases are as above.

Case WHILE exp DO cmd OD. Assume P-C(col-c,sts-c). Let g=λc'.ℰ⟦exp⟧occ§<1>;cond(𝒞⟦cmd⟧occ§<2>;c', attach<occ,"cmd)">;c). Then by "1)" to "4)" and induction hypotheses it easily follows that P-C(col-c',sts-c') => P-C(col-g(col-c'), sts-g(sts-c')). Since P-C(⊥,⊥) a proof by induction shows ∀n≥0: P-C($(\text{col-g})^n\bot$, $(\text{sts-g})^n\bot$).

To deduce P-C(FIX(col-g), FIX(sts-g)) we assume P-S($\underline{\text{sta}}$). Then
FIX(sts-g)$\underline{\text{sta}}$
 = ⊔{ $(\text{sts-g})^n\bot$ $\underline{\text{sta}}$ |n≥0}
 = ⊔{ ⊔{ $(\text{col-g})^n\bot$ sta |sta∈$\underline{\text{sta}}$} |n≥0} (and by Lemma 2.1.1-7 twice:)
 = ⊔{ ⊔{ $(\text{col-g})^n\bot$ sta |n≥0} |sta∈$\underline{\text{sta}}$}
 = ⊔{ FIX(col-g) sta |sta∈$\underline{\text{sta}}$}
This shows P-C(FIX(col-g), FIX(sts-g)) from which

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

Structural Induction on Pro: Surely P-C(col-finish, sts-finish) so that the above inductions give P-C($\mathcal{D}$col⟦dcl⟧<1>;$\mathcal{C}$col⟦cmd⟧<2>;col-finish, $\mathcal{D}$sts⟦dcl⟧<1>;$\mathcal{C}$sts⟦cmd⟧<2>;sts-finish). Then the result follows by "4)".
□

PROOF OF 3.1-10:
The proof is by structural induction. We define the following predicates:
P-C:(apr1-C$\times$apr2-C)->B by P-C(apr1-c,apr2-c) =
(apr1-c $\circ$ conc $\sqsubseteq$ (R conc) $\circ$ apr2-c)
P-Cmd:Cmd->B by P-Cmd(cmd) = P-C(apr1-c,apr2-c) =>
  P-C($\mathcal{C}$apr1⟦cmd⟧occ apr1-c, $\mathcal{C}$apr2⟦cmd⟧occ apr2-c)
and P-Exp:Exp->B and P-Dcl:Dcl->B similarly.

Structural induction on Dcl: The proof is straight-forward using apr1 $\sqsubseteq$<uabs,conc> apr2.

Structural induction on Exp: As above.

Structural Induction on Cmd: The cases mostly are as above. Consider the case WHILE exp DO cmd OD. Let
g[c] = λc'.$\mathcal{E}$⟦exp⟧occ§<1>;cond($\mathcal{C}$⟦cmd⟧occ§<2>;c', attach<occ,"cmd)">;c).
Assume P-C(apr1-c,apr2-c). Then P-C(apr1-c',apr2-c') =>
P-C(apr1-g[apr1-c] apr1-c', apr2-g[apr2-c] apr2-c'). Since P-C($\perp$,$\perp$) a proof by induction shows $\forall n \geq 0$: P-C((apr1-g[apr1-c])$^n\perp$, (apr2-g[apr2-c])$^n\perp$). To show P-C(FIX(apr1-g[apr1-c]), FIX(apr2-g[apr2-c])) we calculate:
FIX(apr1-g[apr1-c])$\circ$conc
 = $\sqcup${(apr1-g[apr1-c])$^n\perp$ $\circ$ conc |n$\geq$o}
 $\sqsubseteq$ $\sqcup${(R conc)$\circ$ (apr2-g[apr2-c])$^n\perp$ |n$\geq$0}
 $\sqsubseteq$ (R conc)$\circ$ $\sqcup${(apr2-g[apr2-c])$^n\perp$ |n$\geq$0}
   (which was by isotony of R conc)
 = (R conc) $\circ$ FIX(apr2-g[apr2-c]).

Structural induction on Pro: That the result holds is directly from
P-C($\mathcal{D}$apr1⟦dcl⟧<1>;$\mathcal{C}$apr1⟦cmd⟧<2>;apr1-finish,
  $\mathcal{D}$apr2⟦dcl⟧<1>;$\mathcal{C}$apr2⟦cmd⟧<2>;apr2-finish)
and apr1 $\sqsubseteq$<uabs,conc> apr2.
□

PROOF OF 3.1-12:
We must show the conditions of Definition 3.1-9: Let P-C(sts-c,ind-c) =
(sts-c $\circ$ conc $\sqsubseteq$ (R conc) $\circ$ind-c).

a) P-C(sts-wrong,ind-wrong) and P-C(sts-finish,ind-finish) are immediate.

b) Assume P-C(sts-c1,ind-c1) and P-C(sts-c2,ind-c2). Then
 sts-cond(sts-c1,sts-c2) $\circ$ conc
  $\sqsubseteq$ λs.sts-c1(conc(uabs(sts-Dt-cond(conc(s))))) $\sqcup$
    sts-c2(conc(uabs(sts-Df-cond(conc(s)))))
     (which was because sts-c1 and sts-c2 isotone and
     conc$\circ$uabs extensive)
  $\sqsubseteq$ λs.(R conc)(ind-c1(ind-Dt-cond(s))) $\sqcup$
    (R conc)(ind-c2(ind-Df-cond(s)))
     (which was by assumptions)
  $\sqsubseteq$ (R conc) $\circ$ (ind-cond(ind-c1,ind-c2))
    (which was by R conc isotone)

This shows P-C(sts-cond(sts-c1,sts-c2), ind-cond(ind-c1,ind-c2)).

c&d&e) The proof goes essentially as in case "b)" above. In the proof of "c)" we use the fact $\forall s \in S$: $\perp$[conc(s)/pla] $\sqsubseteq$ (R conc)($\perp$[s/pla]). []

PROOF OF 3.1-13:
Define the predicates
P:(sts-C$\times$ind-C)->B by P(sts-c,ind-c)=[sts-c $\circ$ conc $\sqsubseteq$ (R conc) $\circ$ ind-c]
Q:(ind-C$\times$apr-C)->B by Q(ind-c,apr-c)=[ind-c $\sqsubseteq$ apr-c]

1) Assume ind<uabs,conc> $\sqsubseteq$<λs.s,λs.s> apr and show sts $\sqsubseteq$<uabs,conc> apr by proving the conditions of Definition 3.1-9.

a) Clearly P(sts-wrong,apr-wrong) and P(sts-finish,apr-finish).

b) Assume P(sts-c1,apr-c1) and P(sts-c2,apr-c2). Then by Lemma 3.1-12
sts-cond(sts-c1,sts-c2)$\circ$conc $\sqsubseteq$ (R conc)$\circ$(ind-cond(apr-c1,apr-c2))
But Q(apr-c1,apr-c1) $\wedge$ Q(apr-c2,apr-c2) so ind $\sqsubseteq$<λs.s,λs.s> apr implies
ind-cond(apr-c1,apr-c2) $\sqsubseteq$ apr-cond(apr-c1,apr-c2)
Also (R conc) is isotone. This implies
sts-cond(sts-c1,sts-c2)$\circ$conc $\sqsubseteq$ (R conc)$\circ$(apr-cond(apr-c1,apr-c2))
From this P(sts-cond(sts-c1,sts-c2), apr-cond(apr-c1, apr-c2) follows.

c&d&e) The proof goes as in "b)" above.

2) Assume sts $\sqsubseteq$<uabs,conc> apr and show ind<uabs,conc> $\sqsubseteq$<λs.s,λs.s> apr by proving the conditions of Definition 3.1-9.

a) Obviously Q(ind-wrong,apr-wrong) and Q(ind-finish,apr-finish).

b) Assume Q(ind-c1,apr-c1) and Q(ind-c2,apr-c2). Since apr-cond is isotone in both parameters we only need to show
ind-cond(ind-c1,ind-c2) $\sqsubseteq$ apr-cond(ind-c1,ind-c2)
to be able to deduce Q(ind-cond(ind-c1,ind-c2), apr-cond(apr-c1,apr-c2)).
Define sts-c1 = (R conc)$\circ$ ind-c1 $\circ$ uabs
sts-c2 = (R conc) $\circ$ ind-c2 $\circ$ uabs
Then P(sts-c1,ind-c1) and P(sts-c2,ind-c2) because uabs$\circ$conc is the identity (reductive is enough). Then (by hypothesis)
P(sts-cond(sts-c1,sts-c2), apr-cond(ind-c1,ind-c2))
i.e.
(sts-c1 $\circ$ sts-Dt-cond $\circ$ conc) $\sqcup$ (sts-c2 $\circ$ sts-Df-cond $\circ$ conc) $\sqsubseteq$
(R conc) $\circ$ apr-cond(ind-c1,ind-c2)
and because (R uabs) is isotone we get:
[(R uabs)$\circ$(R conc)$\circ$ind-c1$\circ$uabs$\circ$sts-Dt-cond$\circ$conc] $\sqcup$ [...] $\sqsubseteq$
(R uabs) $\circ$ (R conc) $\circ$ apr-cond(ind-c1,ind-c2)
When <uabs,conc> is exact this is equivalent to
ind-c1$\circ$ind-Dt-cond $\sqcup$ ind-c2$\circ$ind-Df-cond $\sqsubseteq$ apr-cond(ind-c1,ind-c2)
as was to be shown.

c&d&e) The proof goes as in "b)" above. []

PROOF OF 3.1-14:
That <conc$\circ$uabs,λ<u>sta</u>.<u>sta</u>> is a pair of semi-adjoined functions is immediate. Define ind1=ind<conc$\circ$uabs, λ<u>sta</u>.<u>sta</u>> and ind2=ind<uabs,conc>. We show the result by a structural induction that uses the predicates:
P-C:(ind1-C$\times$ind2-C)->B where P-C(ind1-c,ind2-c) =

[(R uabs)∘ind1-c∘conc∘uabs = ind2-c∘uabs]
P-Cmd:Cmd->B where P-Cmd(cmd) = [P-C(ind1-c,ind2-c) =>
P-C(𝒞ind1⟦cmd⟧occ ind1-c, 𝒞ind2⟦cmd⟧occ ind2-c)
and P-Dcl and P-Exp are defined similarly.

Before we approach the structural induction ("6" below) it is convenient to show some properties of primitive functions, auxiliary functions and constants.

1) Since uabs and conc are adjoined we have uabs(⊥)=⊥ so
P-C(ind1-wrong,ind2-wrong) and P-C(ind1-finish,ind2-finish).

2) We show P-C(ind1-c1,ind2-c1) ∧ P-C(ind1-c2,ind2-c2)
=> P-C(ind1-cond(ind1-c1,ind1-c2), ind2-cond(ind2-c1,ind2-c2))
Assume P-C(ind1-c1,ind2-c1) and P-C(ind1-c2,ind2-c2). Then
(R uabs)∘ind1-cond(ind1-c1,ind1-c2)∘ conc∘uabs
= (R uabs)∘ind1-c1∘conc∘uabs∘sts-Dt-cond∘conc∘uabs ⊔
(R uabs)∘ ind1-c2∘conc∘uabs∘sts-Df-cond∘conc∘uabs
(which was by uabs a (complete-)⊔-morphism)
= [ind2-c1∘ind2-Dt-cond ⊔ ind2-c2∘ind2-Df-cond]∘uabs
(which was by assumtions)
= ind2-cond(ind2-c1,ind2-c2)∘ uabs
Hence P-C(ind1-cond(ind1-c1,ind1-c2), ind2-cond(ind2-c1,ind2-c2)).

3) We show P-C(ind1-c,ind2-c) =>
P-C(ind1-attach(pla)(ind1-c), ind2-attach(pla)(ind2-c))
Assume P-C(ind1-c,ind2-c). Then
(R uabs)∘(ind1-attach(pla)∘ind1-c)∘ conc∘uabs
= λs.(R uabs)(ind1-c(conc(uabs(s)))) ⊔ (R uabs)(⊥[conc(uabs(s))/pla])
(which was because uabs a (complete-)⊔-morphism)
= λs. ind2-c(uabs(s)) ⊔ ⊥[(uabs∘conc∘uabs)(s)/pla]
(which was by assumption P-C(...,...) and uabs(⊤)=⊤ and uabs(⊥)=⊥)
= (ind2-attach(pla)(ind2-c))∘uabs
(which was by uabs∘conc∘uabs = uabs which is
a property of a pair <uabs,conc> of adjoined functions)
Hence P-C(ind1-attach(pla)(ind1-c), ind2-attach(pla)(ind2-c)).

4) The proof of P-C(ind1-c,ind2-c) => ∀inp:
(R uabs)(ind1-setup(ind1-c)inp) = ind2-setup(ind2-c)inp is easy.

5) The proof of P-C(ind1-c,ind2-c) =>
P-C(ind1-g(par)ind1-c, ind2-g(par)ind2-c) for a primitive function g goes as case "2)".

6) The structural induction is mostly straight-forward using the results of "1)" to "5)". We consider the case WHILE exp DO cmd OD. Define
g[c] = λc'.ℰ⟦exp⟧occ§<1>; cond(𝒞⟦cmd⟧occ§<2>;c', attach<occ,"cmd)">;c)
Then by hypotheses we may assume
P-C(ind1-c,ind2-c) ∧ P-C(ind1-c',ind2-c') =>
P-C(ind1-g[ind1-c]ind1-c', ind2-g[ind2-c]ind2-c').
Assume P-C(ind1-c,ind2-c). We now show
P-C(FIX(ind1-g[ind1-c]),FIX(ind2-g[ind2-c]))
Since P-C(⊥,⊥) a proof by induction yields ∀n≥0:
P-C([ind1-g[ind1-c]]^n⊥,[ind2-g[ind2-c]]^n⊥). Furthermore,
(R uabs)∘(⊔{(ind1-g[ind1-c])^n⊥ |n≥0}) ∘ conc ∘ uabs
= ⊔{ (R uabs)∘([ind1-g[ind1-c]]^n⊥)∘conc∘uabs |n≥0}

(this was by uabs a complete-$\sqcup$-morphism)

$= \bigsqcup\{ (\text{ind2-g}[\text{ind2-c}])^n \bot \circ \text{uabs} \mid n \geq 0\}$

(this was by $\forall n \geq 0$: P-C(...,...))

$= \text{FIX}(\text{ind2-g}[\text{ind2-c}]) \circ \text{uabs}$

This shows P-C(FIX(ind1-g[ind1-c]), FIX(ind2-g[ind2-c])). From this P-Cmd(WHILE exp DO cmd OD) easily follows. []

PROOF OF 3.1-15:

Abbreviate uco1=conc1$\circ$uabs1, uco2=conc2$\circ$uabs2, id=$\lambda \underline{sta}.\underline{sta}$, ind1=ind<uco1,id> and ind2=ind<uco2,id>.

We omit the proof of ind1 $\sqsubseteq_{\langle id,id\rangle}$ ind2, since it is straight-forward. From this result and Theorem 3.1-10 we have

$\forall \text{pro}:\forall \underline{inp}: \mathcal{P}\text{ind1}[\![\text{pro}]\!]\underline{inp} \subseteq \mathcal{P}\text{ind2}[\![\text{pro}]\!]\underline{inp}$

and by isotony of uco1 and uco1 $\sqsubseteq$ uco2

$\forall \text{pro}:\forall \underline{inp}: (R\ \text{uco1})(\mathcal{P}\text{ind1}[\![\text{pro}]\!]\underline{inp}) \sqsubseteq (R\ \text{uco2})(\mathcal{P}\text{ind2}[\![\text{pro}]\!]\underline{inp})$

By Lemma 3.1-14:

$\forall \text{pro}:\forall \underline{inp}: (R\ \text{conc1})(\mathcal{P}\text{ind}\langle \text{uabs1},\text{conc1}\rangle[\![\text{pro}]\!]\underline{inp}) \sqsubseteq$
$\quad (R\ \text{conc2})(\mathcal{P}\text{ind}\langle \text{uabs2},\text{conc2}\rangle[\![\text{pro}]\!]\underline{inp})$

[]

PROOF OF 3.1-16:

If <uabs,conc> is a pair of adjoined functions then conc $\circ$ uabs is an upper closure operator by Lemma 2.4.3-12.

Conversely, let uco:$\mathcal{P}$(Sta)->$\mathcal{P}$(Sta) be an upper closure operator. Let S = $\{\text{uco}(\underline{sta}) \mid \underline{sta} \in \mathcal{P}(\text{Sta})\}$ and id:S->S be λs.s. Then <uco,id> is a pair of adjoined functions, as can easily be shown. Furthermore uco($\top$)=$\top$ because uco is extensive. Below we show that S is a complete lattice.

Let $\sqsubseteq$, $\sqcup$ and $\sqcap$ be those of $\mathcal{P}$(Sta). Clearly (S,$\sqsubseteq$) is a partially ordered set. We show $\forall \underline{s} \subseteq S: \sqcap\underline{s} \in S$. This follows from

$\sqcap\underline{s} \sqsubseteq \text{uco}(\sqcap\underline{s}) \sqsubseteq \sqcap\{\text{uco}(s) \mid s \in \underline{s}\} = \sqcap\underline{s}$

(where we have used the properties of an upper closure operator) so that $\sqcap\underline{s}$ = uco($\sqcap\underline{s}$) $\in$ S. This implies that for arbitrary $\underline{s} \subseteq S$

$\forall s \in S: \sqcap\underline{s} \sqsupseteq s \iff \underline{s} \sqsupseteq s$

(because $\sqcap$ is meet of $\mathcal{P}$(Sta) and S$\subseteq$ $\mathcal{P}$(Sta)) so that $\sqcap$ is also meet of S (and exists).

We now show that $\forall \underline{s} \subseteq S: \exists s0 \in S: \forall s1 \in S: \underline{s} \sqsubseteq s1 \iff s0 \sqsubseteq s1$ because then S is a complete lattice. For fixed $\underline{s}$ set s0 = $\sqcap\{s2 \mid s2 \sqsupseteq \underline{s}\} \in S$. Then

$\underline{s} \sqsubseteq s1 \Rightarrow \{s2 \mid s2 \sqsupseteq \underline{s}\} \ni s1 \Rightarrow s0 \sqsubseteq s1$

and

$s0 \sqsubseteq s1 \Rightarrow \forall s3 \in \underline{s}: s3 \sqsubseteq \sqcap\{s4 \mid s4 \sqsupseteq s3\} \sqsubseteq s0 \sqsubseteq s1 \Rightarrow \underline{s} \sqsubseteq s1$ []

PROOF OF 3.1-17:

The part uco1$\sqsubseteq$uco2 => ind<uco1,id> $\sqsubseteq_{\langle id,id\rangle}$ ind<uco2,id> is to be proven as in the proof of Lemma 3.1-15 (where we omitted the proof).

For the converse implication abbreviate ind1 = ind<uco1,id> and ind2 = ind<uco2,id>. Define c$\in\mathcal{P}$(Sta)-i>Pla-c>$\mathcal{P}$(Sta) by c=$\lambda\underline{sta}.\lambda$pla.$\underline{sta}$. Note $\forall\underline{sta}$: (ind1-dummy;c)$\underline{sta}$ = λpla.uco1($\underline{sta}$) and $\forall\underline{sta}$: (ind2-dummy;c)$\underline{sta}$ = λpla.uco2($\underline{sta}$). So by ind1 $\sqsubseteq_{\langle id,id\rangle}$ ind2 we get $\forall\underline{sta}$: uco1($\underline{sta}$) $\sqsubseteq$ uco2($\underline{sta}$), i.e uco1 $\sqsubseteq$ uco2. []

PROOF OF 4.1-6:
The proof is by structural induction. Define the predicates:
P-S: $\mathcal{P}$(Sta*)->B by P-S(s)= $\forall$sta* $\in$ s: topfree[Sta*](sta*) $\wedge$ #sta* $\notin \{\perp, \top, 0\}$
P-C:(his-col-C $\times$ his-sts-C)->B by P-C(his-col-c,his-sts-c) =
 $\forall$s $\in \mathcal{P}$(Sta*): P-S(s) => his-sts-c(s) = $\bigsqcup\{$his-col-c(sta*) | sta* $\in$ s$\}$
P-Cmd:Cmd->B by P-Cmd(cmd) = [P-C(his-col-c,his-sts-c) =>
 P-C($\mathcal{E}$his-col$[\![$cmd$]\!]$occ his-col-c, $\mathcal{E}$his-sts$[\![$cmd$]\!]$ occ his-sts-c)
and P-Exp and P-Dcl similarly.

It is easy to modify the proof of Theorem 3.1-6 to hold in this setting; note that topfree[Sta*](sta*) $\wedge$ #sta* $\notin \{\perp, \top, 0\}$ => topfree[Sta](sta*.last) []

PROOF OF 4.1-9:
To show that uabs and conc are adjoined is by straight-forward calculations.
To show exactness: We know uabs(conc($\underline{\underline{\text{sta}}}$)) $\sqsubseteq$ $\underline{\underline{\text{sta}}}$. Let sta $\in$ $\underline{\underline{\text{sta}}}$. Then
<sta> $\in$ conc($\underline{\underline{\text{sta}}}$) so {<sta>} $\subseteq$ conc($\underline{\underline{\text{sta}}}$) so
sta $\in$ uabs({<sta>}) $\subseteq$ uabs(conc($\underline{\underline{\text{sta}}}$)), i.e. uabs(conc($\underline{\underline{\text{sta}}}$)) $\supseteq$ $\underline{\underline{\text{sta}}}$. []

PROOF OF 4.1-10:
Define P-C:(sts-C $\times$ his-ind-C)->B by P-C(sts-c,his-ind-c) = (sts-c=his-ind-c).
Then

a) P-C(sts-wrong,his-ind-wrong) and P-C(sts-finish,his-ind-finish)

b) Suppose P-C(sts-c1,his-ind-c1) and P-C(sts-c2,his-ind-c2). Then
his-ind-cond(his-ind-c1, his-ind-c2)
 = sts-c1 $\circ$ uabs $\circ$ his-sts-Dt-cond $\circ$ conc $\sqcup$ sts-c2 $\circ$ uabs $\circ$ his-sts-Df-cond $\circ$ conc
Now uabs(his-sts-Dt-cond(conc($\underline{\underline{\text{sta}}}$)))
= uabs {his-std-Bcond(sta*) | sta* $\in$ conc($\underline{\underline{\text{sta}}}$) $\wedge$ his-std-Vcond(sta*)=true
 $\wedge$ his-std-Scond(sta*)=true}
= {(sta* § <std-Bcond(sta*.last)>).last | sta*.last $\in$ $\underline{\underline{\text{sta}}}$
 $\wedge$ std-Vcond(sta*.last) = true
 $\wedge$ std-Scond(sta*.last) = true }
= {(sta* § <std-Bcond(sta*.last)>).last | sta* $\notin \{\perp, \top\}$
 $\wedge$ ...conditions as above... }

This was because sta* = $\perp$ => sta* § <...> = $\perp$ => (sta* § <...>).last = $\perp$
since $\perp$ ⩛ $\perp$ = $\perp$. But $\perp$ ⩛ $\perp$ $\in$ $\underline{\underline{\text{sta}}}$ => <$\perp$> $\in$ conc($\underline{\underline{\text{sta}}}$) and <$\perp$>.last = $\perp$
so that no elements were excluded from the set by requiring sta* $\neq \perp$.
Similarly for $\top$. So

= {std-Bcond(sta) | sta $\in$ $\underline{\underline{\text{sta}}}$ $\wedge$ std-Vcond(sta)=true $\wedge$ std-Scond(sta)=true}
= sts-Dt-cond($\underline{\underline{\text{sta}}}$)

Similarly for ...-Df-cond, so we have his-ind-cond(his-ind-c1,his-ind-c2)
= sts-c1 $\circ$ sts-Dt-cond $\sqcup$ sts-c2 $\circ$ sts-Df-cond
= sts-cond(sts-c1,sts-c2)

showing P-C(sts-cond(sts-c1,sts-c2), his-ind-cond(his-ind-c1,his-ind-c2)).

c) Clearly P-C(sts-c,his-ind-c) =>
```
  P-C(sts-attach(pla)(sts-c), his-ind-attach(pla)(his-ind-c))
```

d) Clearly P-C(sts-c,his-ind-c) =>
```
  sts-setup(sts-c)(inp) = his-ind-setup(his-ind-c)(inp)
  since uabs{<sta>|...} = {sta|...}
```

e) Along the same lines as in "b)" we show P-C(sts-c,his-ind-c) =>
```
  P-C(sts-g(par)(sts-c), his-ind-g(par)(his-ind-c)).
```

Similarly to the above result it is not difficult to show the two results stated in square brackets. []

PROOF OF 4.2.1-2:
We only prove the result for std since the proof for his-sts essentially is similar to that of std. Below we elide the prefix std-.

1) For g anyone of apply, content, push and for arbitrary c∈C we have:
```
  c⊕[Pg(par)]
  = λsta.Pg(par)(sta) ∈Sta -> c((Pg(par)(sta)) |Sta), "wrong" inA
  = λsta.Vg(par)(sta) -> c(Bg(par)(sta)), "wrong" inA
  = g(par)(c)
```

2) Similarly, for c∈C we have c⊕[Pattach pla] = attach(pla)(c).

3) We then show c⊕(ss1*ss2) = (c⊕ss1)⊕ss2 for any c∈C, ss1 ∈ Sta -c> R, ss2 ∈ Sta -c> R. We have c⊕(ss1*ss2)
```
  = λsta.(ss1*ss2)(sta) ∈Sta -> c((ss1*ss2)(sta) |Sta), "wrong" inA
  = λsta.[ ss2(sta) ∈Sta -> ss1(ss2(sta) |Sta), ss2(sta) ] ∈Sta ->
           c((ss1*ss2)(sta) |Sta), "wrong" inA
  = λsta. ss2(sta) ∈Sta -> [ss1(ss2(sta) |Sta) ∈Sta ->
           c((ss1*ss2)(sta) |Sta), "wrong" inA ], "wrong" inA
  = λsta. ss2(sta)∈Sta -> [ ss1(ss2(sta) |Sta) ∈Sta ->
           c(ss1(ss2(sta) |Sta) |Sta), "wrong" inA ],
           "wrong" inA
  = λsta. ss2(sta) ∈Sta -> [(c⊕ss1)(ss2(sta) |Sta)], "wrong" inA
  = (c⊕ss1)⊕ss2
```

4) The proof of the Lemma is by structural induction on Exp. This is straight-forward by "1)", "2)" and "3)". []

PROOF OF 4.2.1-4:
The proof is by structural induction on Exp. Define predicates P-S:Sta->B by P-S(sta) = sta.wit∉{⊥,⊤} and P-Exp:Exp->B by
```
P-Exp(exp) = ∀s∈℘(Sta*): [ (∀sta*∈s: P-S(sta*.last)) =>
   [ {sta*.last |sta*∈(Pζhis-sts⟦exp⟧occ s)↓1}
   = {(Pζstd⟦exp⟧occ (sta*.last)) |Sta | sta*∈s} ∧
   ∀sta*∈(Pζhis-sts⟦exp⟧occ s)↓1: P-S(sta*.last) ]]
```

1) <u>Case</u> <u>bas</u>: Assume ∀sta*∈s: P-S(sta*.last). Then
```
  Pζhis-sts⟦bas⟧occ s↓1
  = his-sts-Dpush⟦bas⟧ s
  = {his-std-Bpush⟦bas⟧ sta* |sta*∈s ∧ his-std-Vpush⟦bas⟧ sta* = true}
  = {sta*§<std-Bpush⟦bas⟧(sta*.last)> |sta*∈s}
  Since ⊥,⊤∉s we have
  {sta*.last |sta*∈(Pζhis-sts⟦bas⟧occ s)↓1}
```

= {std-Bpush⟦bas⟧(sta*.last) |sta*∈s}
= {(P℘std⟦bas⟧ occ (sta*.last)) |Sta | sta*∈ s}
and (e.g. from 4.2.1-3) sta*∈s => P-S(P℘std⟦bas⟧occ (sta*.last) |Sta).
Hence P-Exp(bas).

2) Case ide: This case is similar to the above case.

3) Case exp1 ope exp2: Assume ∀sta*∈s: P-S(sta*.last). Then
P℘his-sts⟦exp1 ope exp2⟧ occ s ↓1
= his-sts-Dapply⟦ope⟧ s2
where s2 = P℘his-sts⟦exp2⟧ occ§<3> s1 ↓1
s1 = P℘his-sts⟦exp1⟧ occ§<1> s ↓1
So that by P-Exp(exp1) and P-Exp(exp2)
{sta*.last | sta*∈s2}
= {P℘std⟦exp2⟧(occ§<3>)(sta*.last)|Sta | sta*∈s1}
= {P℘std⟦exp2⟧(occ§<3>)(P℘std⟦exp1⟧(occ§<1>)(sta*.last)|Sta)|Sta
| sta*∈ s}
and by several applications of Lemma 4.2.1-3
{ sta*.last | sta*∈P℘his-sts⟦exp1 ope exp2⟧ occ s ↓1}
= { sta*.last | sta*∈his-sts-Dapply⟦ope⟧ s2 }
and by ⊥,⊤∉s2
= { std-Bapply⟦ope⟧(sta*.last) | sta*∈ s2 ∧ std-Vapply⟦ope⟧(sta*.last)=true}
= { std-Papply⟦ope⟧sta|Sta | sta∈{sta*.last | sta*∈s2}}
= { (std-Papply⟦ope⟧ * P℘std⟦exp2⟧occ§<3> * P℘std⟦exp1⟧occ§<1>)
(sta*.last)|Sta | sta*∈s }
= { (P℘std⟦exp1 ope exp2⟧ occ (sta*.last)|Sta) | sta*∈s }

Also sta*∈s => P-S((P℘std⟦exp1 ope exp2⟧ occ (sta*.last)) |Sta) as follows
from Lemma 4.2.1-3. Hence P-Exp(exp1 ope exp2). []

PROOF OF 4.2.2-9:
Shown after the proof of 4.2.2-10. []

PROOF OF 4.2.2-10:
Lemma 4.2.1-3 shows pre3(sta1,sta2,exp) => sta1.wit∉{⊥,⊤} ∧ sta2.wit∉{⊥,⊤}
=> [(P℘std⟦exp⟧ sta1 ↓Sta)=true ∧ (P℘std⟦exp⟧ sta2 ↓Sta)=true ∧
#((P℘std⟦exp⟧sta1 |Sta).wit)∈{1,...}∧
#((P℘std⟦exp⟧sta2 |Sta).wit)∈{1,...}]
It therefore suffices to show ∀exp: P-Exp(exp) where P-Exp: Exp->B is
P-Exp(exp) = ∀sta1,sta2∈Sta: [pre3(sta1,sta2,exp) =>
[(P℘std⟦exp⟧sta1 |Sta).wit↓1 = (P℘std⟦exp⟧sta2 |Sta).wit↓1 ∧
pre3(P℘std⟦exp⟧sta1 |Sta, P℘std⟦exp⟧sta2 |Sta, exp)]]
The proof is by a structural induction:

1) Case bas: Assume pre3(<env1,inp1,out1,wit1>, <env2,inp2,out2,wit2>, bas).
Then P℘std⟦bas⟧<env[i],inp[i],out[i],wit[i]> |Sta
= std-Bpush⟦bas⟧<env[i],inp[i],out[i],wit[i]>
= <env[i],inp[i],out[i], <std-ℬ⟦bas⟧>§wit[i]>
From this P-Exp(bas) easily follows.

2) Case ide: As above.

3) Case exp = exp1 ope exp2: Let sta1, sta2 be such that
pre3(sta1,sta2,exp). Then P℘std⟦exp1 ope exp2⟧ sta[i]
= ((std-Papply⟦ope⟧ * P℘std⟦exp2⟧) * P℘std⟦exp1⟧) sta[i]
= (std-Papply⟦ope⟧ * P℘std⟦exp2⟧) sta[i]'

where Lemma 4.2.1-3 implies that there exists val[i]'∈Val such that
sta[i]' = <sta[i].env, sta[i].inp, sta[i].out, <val[i]'>§(sta[i].wit) >
Since pre3(sta1,sta2,exp) => pre3(sta1,sta2,exp2) we get from P-Exp(exp2)
that val1'=val2'. Obviously pre3(sta1',sta2',exp) because
sta[i]'.env = sta[i].env. So Pℰstd⟦exp1 ope exp2⟧ sta[i]
= std-Papply⟦ope⟧ sta[i]"
where Lemma 4.2.1-3 implies that there exists val[i]"∈Val so sta[i]" =
<sta[i].env, sta[i].inp, sta[i].out, <val[i]",val[i]'> §(sta[i].wit) >
As before, val1"=val2" and pre3(sta1",sta2",exp). Then
Pℰstd⟦exp1 ope exp2⟧ sta[i]
= sta[i]"' in(Sta+{"wrong"}) where
sta[i]"' = <sta[i].env, sta[i].inp, sta[i].out
, <std-𝒪⟦ope⟧<val[i]',val[i]">§(sta[i].wit) >
Then sta1"'.wit↓1 = sta2"'.wit↓1 and pre3(sta1"',sta2"',exp).
This shows P-Exp(exp1 ope exp2). []

PROOF OF 4.2.2-9:
To show uabs-r2 ⊆ uabs-r3 it suffices to show ∀sta*∈Sta*: r2(sta*) ⊇
r3(sta*). By the form of the definitions of r2 and r3 it is enough to show
∀sta1,sta2,exp: pre3(sta1,sta2,exp) => pre2(sta1,sta2,exp).
This follows from Lemma 4.2.2-10. []

PROOF OF 4.2.3-2:
After the proof of 4.2.3-6. []

PROOF OF 4.2.3-5:
Define the predicate P-Exp: Exp->B by P-Exp(exp) =
[P-S(his-sts-s,ae-s => [
P-S((Pℰhis-sts⟦exp⟧occ his-sts-s)↓1, (Pℰae⟦exp⟧occ ae-s)↓1) ∧
Pℰhis-sts⟦exp⟧occ his-sts-s ↓2 ⊑ (R conc-r3)(Pℰae⟦exp⟧ occ ae-s ↓2)]]

1) We must show P-C(his-sts-c,ae-c) =>
P-C(ℰhis-sts⟦exp⟧ occ his-sts-c, ℰae⟦exp⟧ occ ae-c). Assume
P-C(his-sts-c,ae-c) ∧ P-Exp(exp) ∧ P-S(his-sts-s,ae-s). Then
ℰhis-sts⟦exp⟧occ his-sts-c his-sts-s
= his-sts-c(Pℰhis-sts⟦exp⟧ occ his-sts-s ↓1) ⊔
Pℰhis-sts⟦exp⟧ occ his-sts-s ↓2
which was by Lemma 4.2.1-2
⊑ his-sts-c(Pℰhis-sts⟦exp⟧ occ his-sts-s ↓1) ⊔
(R conc-r3)(Pℰae⟦exp⟧ occ ae-s↓2)
which was by P-Exp(exp); by P-Exp(exp) ∧ P-C(his-sts-c,ae-c) we get
⊑ (R conc-r3)(ae-c(Pℰae⟦exp⟧ occ ae-s ↓1)) ⊔
(R conc-r3)(Pℰae⟦exp⟧ occ ae-s ↓2)
⊑ (R conc-r3)[ae-c(Pℰae⟦exp⟧ occ ae-s ↓1) ⊔ Pℰae⟦exp⟧ occ ae-s ↓2]
and by Lemma 4.2.3-4
= (R conc-r3)(ℰae⟦exp⟧ occ ae-c ae-s)

We then are left with showing P-Exp(exp). This is done by a structural induction on Exp (cases 2, 3 and 4 below).

2) Case bas: Assume P-S(his-sts-s,ae-s).

a) Pℰhis-sts⟦bas⟧ occ his-sts-s
= <his-sts-s', ⊥[his-sts-s/<occ,"(exp">] ⊔ ⊥[his-sts-s'/<occ,"exp)">] >
where
his-sts-s' = his-sts-Dpush⟦bas⟧ his-sts-s

= {his-std-Bpush⟦bas⟧ sta* | sta*∈his-sts-s}
(this was because his-std-Vpush⟦bas⟧ sta* = true)
= {sta*§< std-Bpush⟦bas⟧(sta*.last)> |sta*∈his-sts-s}

b) Pℰae⟦bas⟧ occ ae-s
= <ae-s', ⊥[ae-s/<occ,"(exp">] ⊔ ⊥[ae-s'/<occ,"exp)">]>
where
conc-r3(ae-s') = conc-r3(ae-s)
= {sta* | r3(sta*) ⊇ ae-s}

c) We now show his-sts-s' ⊆ conc-r3(ae-s'). Let sta*∈his-sts-s. Then
sta*∉{⊥,⊤} so r3(sta*§< std-Bpush⟦bas⟧(sta*.last) >)
⊇ r3(sta*) ∩ {exp |pre3(sta*.last, std-Bpush⟦bas⟧(sta*.last), exp)}
= r3(sta*)   (which was by obvious property of std-Bpush)
⊇ ae-s   (which was by using P-S(his-sts-s,ae-s))
This yields his-sts-s' ⊆ conc-r3(ae-s')

d) For any sta*∈his-sts-s' we have sta*.last.wit∉{⊥,⊤} as easily follows
from Lemma 4.2.1-3. This yields P-S(his-sts-s', ae-s') and P-Exp(bas).

3) Case ide: This case is similar to case 2.

4) Case exp1 ope exp2: Assume P-S(his-sts-s,ae-s). This case does not follow
the pattern of cases 2 and 3. We abbreviate:
F4 = Pattach<occ,"exp)">
F3 = F4 * Papply⟦ope⟧⟦exp1 ope exp2⟧
F2 = F3 * Pℰ⟦exp2⟧ occ§<3>
F1 = F2 * Pℰ⟦exp1⟧ occ§<1>
Then

a) Pℰhis-sts⟦exp1 ope exp2⟧ occ his-sts-s
= his-sts-F1(his-sts-s) ⊔ <⊥, ⊥[his-sts-s/<occ,"(exp">]>
= his-sts-F2(his-sts-s1) ⊔ <⊥, his-sts-a1 ⊔ ⊥[his-sts-s/<occ,"(exp">]>
where <his-sts-s1, his-sts-a1> = Pℰhis-sts⟦exp1⟧ occ§<1> his-sts-s
Similarly, Pℰae⟦exp1 ope exp2⟧ occ ae-s
= ae-F2(ae-s1) ⊔ <⊥, ae-a1 ⊔ ⊥[ae-s/<occ,"(exp">]>
By induction hypothesis P-Exp(exp1) we know
his-sts-a1 ⊆ (R conc-r3)(ae-a1)
P-S(his-sts-s1, ae-s1)
and by Lemma 4.2.1-4 we know
{sta*.last |sta*∈his-sts-s1}
= {Pℰstd⟦exp1⟧(sta*.last) |Sta |sta*∈his-sts-s}

b) Pℰhis-sts⟦exp1 ope exp2⟧ occ his-sts-s
= his-sts-F3(his-sts-s2) ⊔
<⊥, his-sts-a2 ⊔ his-sts-a1 ⊔ ⊥[his-sts-s/<occ,"(exp">] >
and Pℰae⟦exp1 ope exp2⟧ occ ae-s
= ae-F3(ae-s2) ⊔ <⊥, ae-a2 ⊔ ae-a1 ⊔ ⊥[ae-s/<occ,"(exp">]>
where we know
his-sts-a2 ⊆ (R conc-r3)(ae-a2)
P-S(his-sts-s2, ae-s2)
and by Lemma 4.2.1-4
{sta*.last | sta*∈his-sts-s2 }
= {Pℰstd⟦exp2⟧(Pℰstd⟦exp1⟧(sta*.last) |Sta)|Sta | sta*∈his-sts-s}

c) Pℰhis-sts⟦exp1 ope exp2⟧ occ his-sts-s

= his-sts-F4(his-sts-s3) ⊔ <⊥, his-sts-a2 ⊔ his-sts-a1 ⊔
⊥[his-sts-s/<occ,"(exp">]>
where his-sts-s3
= his-sts-Dapply⟦ope⟧(his-sts-s2)
= {sta*§< std-Bapply⟦ope⟧(sta*.last) > |sta*∈his-sts-s2}
because Lemma 4.2.1-3 shows that when sta*∈his-sts-s2 then sta*.last.wit
is of the form <val1,val2>§wit for wit∈{⊥,⊤} so that
std-Vapply⟦ope⟧(sta*.last) = true.
Similarly, P§ae⟦exp1 ope exp2⟧ occ ae-s
= ae-F4(ae-s3) ⊔ <⊥, ae-a2 ⊔ ae-a1 ⊔ ⊥[ae-s/<occ,"(exp">]>
where ae-s3 = ae-s ∪ {exp1 ope exp2}.

We want to show P-S(his-sts-s3,ae-s3). We must show 2 things:

i) ∀sta*∈his-sts-s3: sta*.last.wit∉{⊥,⊤}. This is straight-forward
because ∀sta*∈his-sts-s2: #(sta*.last.wit)∈{2,3,...}.

ii) ∀sta*∈his-sts-s2 that
r3(sta*§< std-Bapply⟦ope⟧(sta*.last) >) ⊇ (ae-s) ∪ {exp1 ope exp2}.
We have r3(sta*§< std-Bapply⟦ope⟧(sta*.last)>)
= [r3(sta*)
∩ {exp |pre3(sta*.last, std-Bapply⟦ope⟧(sta*.last), exp}]
∪ {exp |gen3(std-Bapply⟦ope⟧(sta*.last), exp)}
= r3(sta*) ∪ {exp| gen3(std-Bapply⟦ope⟧(sta*.last), exp)}
So it suffices to show gen3(std-Bapply⟦ope⟧(sta*.last), exp1 ope exp2).
We know by "4b)" ∃sta such that sta.wit∉{⊥,⊤} and
std-Bapply⟦ope⟧(sta*.last)
= std-Bapply⟦ope⟧(P§std⟦exp2⟧(P§std⟦exp1⟧ sta |Sta)|Sta)
which by Lemma 4.2.1-3 is seen to be P§std⟦exp1 ope exp2⟧ sta.
So we must show gen3(P§std(exp1 ope exp2⟧sta |Sta, exp1 ope exp2)
knowing sta.wit∉{⊥,⊤}. An equivalent formulation is to show
pre2(sta, P§std⟦exp1 ope exp2⟧ sta|Sta, exp1 ope exp2).
But we know pre3(sta, P§std⟦exp1 ope exp2⟧ sta|Sta, exp1 ope exp2)
(by 4.2.1-3) so Lemma 4.2.2-10 shows this.

We have shown P-S(his-sts-s3, ae-s3)

d) P§his-sts⟦exp1 ope exp2⟧ occ his-sts-s
= <his-sts-s3, ⊥[his-sts-s3/<occ,"exp)">] ⊔
his-sts-a2 ⊔ his-sts-a1 ⊔ ⊥[his-sts-s/<occ,"(exp">]>
and P§ae⟦exp1 ope exp2⟧ occ ae-s
= <ae-s3, ⊥[ae-s3/<occ,"exp)">] ⊔ ae-a2 ⊔ ae-a1 ⊔ ⊥[ae-s/<occ,"(exp">]>
It easily follows that P-Exp(exp1 ope exp2) holds. []

PROOF OF 4.2.3-6
Assume that P-C(his-sts-c,ae-c) and P-S(his-sts-s,ae-s) hold. We calculate
his-sts-g(par)(his-sts-c)(his-sts-s) = his-sts-c(his-sts-s')
where his-sts-s'
= {his-std-Bg(par)(sta*) |sta*∈his-sts-s ∧ his-std-Vg(par)(sta*)=true}
= {sta*§< std-Bg(par)(sta*.last)> |sta*∈his-sts-s ∧
std-Vg(par)(sta*.last)=true}
Similarly, ae-g(par)(ae-c)(ae-s) = ae-c(ae-s') where ae-s' = PREg(par) ∩ ae-s
because GENg(par) = ∅. It suffices to show P-S(his-sts-s',ae-s').

1) Clearly, ∀sta*∈his-sts-s': sta*.last.wit∉{⊥,⊤}

2) conc-r3(ae-s') = {sta* | r3(sta*) ⊇ PREg(par) ∩ ae-s }

Let sta*∈his-sts-s and assume std-Vg(par)(sta*.last) = true. Abbreviate sta' = std-Bg(par)(sta*.last). We now show r3(sta*§<sta'>) ⊇ PREg(par) ∩ ae-s. Since r3(sta*)⊇ae-s it is enough to show {exp |pre3(sta*.last,sta',exp)} ⊇ PREg(par). We show this for each primitive function (except apply):

g = assign: Let exp ∈ PREassign⟦ide⟧. Then ide∉𝓘⟦exp⟧ and since ide∉{⊥,⊤} and 𝓘⟦exp⟧ ∩ {⊥,⊤} = ∅ we have ∀ide'∈𝓘⟦exp⟧: (sta*.last.env)⟦ide'⟧ = (sta'.env)⟦ide'⟧. Hence pre3(sta*.last,sta',exp).

g ∈ {content, push, read, write}: Trivial since sta*.last.env = sta'.env[]

PROOF OF 4.2.3-2:
The proof is by structural induction. Define the predicates
P-Cmd: Cmd->B by P-Cmd(cmd) = [P-C(his-sts-c,ae-c) =>
P-C(𝒞his-sts⟦cmd⟧ occ his-sts-c, 𝒞ae⟦cmd⟧ occ ae-c)]
and P-Dcl and P-Exp similarly.

<u>Structural induction on Dcl</u>: Along the lines of Lemma 4.2.3-6 we can prove P-C(his-sts-c, ae-c) =>
P-C(his-sts-attach(pla)(his-sts-c), ae-attach(pla)(ae-c)). Then, in view of Lemma 4.2.3-6, the proof is straight-forward.

<u>Structural induction on Exp</u>: Use Lemma 4.2.3-5 instead.

<u>Structural induction on Cmd</u>: Along the lines of Lemma 4.2.3-6 we can prove P-C(his-sts-c1,ae-c1) ∧ P-C(his-sts-c2,ae-c2) =>
P-C(his-sts-cond(his-sts-c1, his-sts-c2), ae-cond(ae-c1, ae-c2)). This makes the structural induction easy, except for the WHILE loop.

For the WHILE loop: Assume P-C(his-sts-c,ae-c) and define
g=λc'. ℰ⟦exp⟧occ§<1>; cond(𝒞⟦cmd⟧occ§<2>;c', attach<occ,"cmd)">;c).
Clearly P-C(his-sts-c',ae-c') => P-C(his-sts-g(his-sts-c'), ae-g(ae-c')).
Also P-C(⊥,⊥) so that ∀n≥0: P-C(his-sts-g^n(⊥), ae-g^n(⊥)). Hence
∀n≥0: P-C(his-sts-g^n(⊥), FIX(ae-g)) and P-C(FIX(his-sts-g), FIX(ae-g)).
Then P-C(𝒞his-sts⟦WHILE exp DO cmd OD⟧occ his-sts-c,
𝒞ae⟦WHILE exp DO cmd OD⟧occ ae-c) easily follows.
Hence P-Cmd(WHILE exp DO cmd OD).

<u>Structural induction on Pro</u>: We must show P-C(his-sts-c,ae-c) =>
∀<u>inp</u>∈𝒫(Inp): his-sts-setup(his-sts-c)(<u>inp</u>)
⊆ (R conc-r3)(ae-setup(ae-c)<u>inp</u>)
It suffices to show P-S(his-sts-s, ae-s) where his-sts-s = {< <λide."nil" inVal, inp, <>, <>> > |inp∈<u>inp</u>} and ae-s = ∅. This is obvious. []

PROOF OF 5.2-5:

Part of the result follows from 5.1-5(1). For the remaining part, assume pure[Pla](pla) and assume $\text{pla-in} \notin \text{local(Aattach pla)} = \{\text{pla}\}$. Let $c \in C$ and $\underline{s} \subseteq S$ be given. Define

$\text{cl} = \text{Close(Aattach(pla)}\langle\text{pla-in}, \lambda s.s\rangle\psi 2, \text{pla-in}, \underline{s})$
$\quad = \lambda\text{pla}'.\ \{ s \mid s \in \underline{s} \wedge \text{pla}'=\text{pla}\}$

Then $(R \sqcup)\text{cl}$

$= \lambda\text{pla}'.\ \sqcup\{ s \mid s \in \underline{s} \wedge \text{pla}'=\text{pla}\}$
$= \lambda\text{pla}'.\ \sqcup(\cup\{ \{s \mid \text{pla}'=\text{pla}\} \mid s \in \underline{s}\})$
$= \lambda\text{pla}'.\ \sqcup\{ \sqcup\{s \mid \text{pla}'=\text{pla}\} \mid s \in \underline{s}\}$ (by 2.1.1-7)
$= \lambda\text{pla}'.\ \sqcup\{ \text{pla}'=\text{pla} \to s, \perp \mid s \in \underline{s}\}$
EQ $\lambda\text{pla}'.\ \sqcup\{ \text{pla}==\text{pla} \to s, \perp \mid s \in \underline{s}\}$
(which was by pure[Pla](pla) and Assumption 2.2-3)
$= \sqcup\{\perp[s/\text{pla}] \mid s \in \underline{s}\}$

so that $\sqcup\{\text{attach(pla)(c)(s)} \mid s \in \underline{s}\}$

$= \sqcup\{ c(s) \sqcup \perp[s/\text{pla}] \mid s \in \underline{s}\}$
$= \sqcup\{ c(s) \mid s \in \underline{s}\} \sqcup \sqcup\{\perp[s/\text{pla}] \mid s \in \underline{s}\}$
EQ $\sqcup\{ c(s) \mid s \in \text{cl(out(Aattach pla))} \} \sqcup (R \sqcup)\text{cl}$ []

PROOF OF 5.2-7:

Part of the result is by 5.1-5(3), because $\text{out(Ag2)} \in \text{local(Ag2)}$ implies $\text{out(Ag2)} \notin \text{local(Ag1)}$ when $\text{local(Ag1)} \cap \text{local(Ag2)} = \emptyset$. For the remaining part, assume $\text{P-G(g1,Ag1)} \wedge \text{P-G(g2,Ag2)} \wedge \text{local(Ag1)} \cap \text{local(Ag2)} = \emptyset$ and $\text{pla-in} \notin \text{local(Ag1)} \cup \text{local(Ag2)}$. Let $c \in C$ and $\underline{s} \subseteq S$ be given. Then

$\sqcup\{ (g1;g2;c)s \mid s \in \underline{s} \}$

EQ $\sqcup\{ (g2;c)s \mid s \in \text{cl1[out(Ag1)]} \} \sqcup (R \sqcup)\text{cl1}$
which was by hypothesis where
$\text{cl1} = \text{Close(Ag1}\langle\text{pla-in}, \lambda s.s\rangle\psi 2, \text{pla-in}, \underline{s})$

EQ $\sqcup\{ c(s) \mid s \in \text{cl2[out(Ag2)]} \} \sqcup (R \sqcup)\text{cl1} \sqcup (R \sqcup)\text{cl2}$
which was by hypothesis where
$\text{cl2} = \text{Close(Ag2}\langle\text{out(Ag1)}, \lambda s.s\rangle\psi 2, \text{out(Ag1)}, \text{cl1[out(Ag1)]})$

$= \sqcup\{ c(s) \mid s \in (\text{cl1} \sqcup \text{cl2})\text{[out(Ag2)]}\} \sqcup (R \sqcup)(\text{cl1} \sqcup \text{cl2})$
because $\text{cl1[out(Ag2)]} = \perp$ (from $\text{local(Ag1)} \cap \text{local(Ag2)} = \emptyset$)

Define $\text{cl} = \text{Close((Ag2*Ag1)}\langle\text{pla-in}, \lambda s.s\rangle\psi 2, \text{pla-in}, \underline{s})$
$\quad = \text{Close(Ag2}\langle\text{out(Ag1)}, \lambda s.s\rangle\psi 2 \sqcup \text{Ag1}\langle\text{pla-in}, \lambda s.s\rangle\psi 2, \text{pla-in}, \underline{s})$

Then it suffices to show $\text{cl} = \text{cl1} \sqcup \text{cl2}$, where $\sqcup$ is the binary join of $\text{Pla} \xrightarrow{t} \mathcal{P}(S)$.

To show $\text{cl} \sqsupseteq \text{cl1} \sqcup \text{cl2}$ we show $\text{cl} \sqsupseteq \text{cl1}$ and $\text{cl} \sqsupseteq \text{cl2}$. That $\text{cl} \sqsupseteq \text{cl1}$ is by Observation 5.2-2. That $\text{cl} \sqsupseteq \text{cl2}$ is by straight-forward calculations simpler than those below for the case $\text{cl} \sqsubseteq \text{cl1} \sqcup \text{cl2}$. So consider the converse inclusion. Let pla' be arbitrary and assume $s' \in \text{cl(pla')}$. Define trans(i) as a shorthand for $\langle\langle\text{pla}[i-1], \text{pla}[i]\rangle, d[i]\rangle$. Then

$\exists s \in \underline{s}$
$\exists \text{trans}(1), \ldots, \text{trans}(n) \in \text{Ag1}\langle\text{pla-in}, \lambda s.s\rangle\psi 2 \cup \text{Ag2}\langle\text{out(Ag1)}, \lambda s.s\rangle\psi 2$

so $\text{pla}[0] = \text{pla-in} \wedge \text{pla}[n] = \text{pla}' \wedge s' = dn(\ldots(d1(s)))$

Since $\text{pla-in} \notin \{\text{out(Ag1)}\} \cup \text{local(Ag2)}$ we know $\text{trans}(1) \in \text{Ag1}\langle\text{pla-in}, \lambda s.s\rangle\psi 2$. This implies the existence of

$k1 = \max\{j \mid trans(1),\ldots,trans(j) \in Ag1\langle pla\text{-}in, \lambda s.s\rangle\downarrow 2\}$
Then $d[k1](\ldots(d1(s))) \in cl1(pla[k1])$ and we are done if $k1 = n$. Otherwise $k1 < n$ so that
$k2 = \max\{j \mid trans(k1+1),\ldots,trans(j) \in Ag2\langle out(Ag1), \lambda s.s\rangle\downarrow 2\}$
exits. Then $pla[k1] \in Local(Ag1) \wedge pla[k1] \in Local(Ag2) \cup \{out(Ag1)\}$ so $pla[k1] = out(Ag1)$ showing $d[k2](\ldots(d[k1](\ldots(d1(s))) \in cl2\ pla[k2]$.

We are done if $k2 = n$. To see that $k2 = n$ assume $k2 < n$. Then $pla[k2] \in Local(Ag2) \wedge pla[k2] \in Local(Ag1) \cup \{pla\text{-}in\}$ which is a contradiction[]

PROOF OF 5.2-8:
Proof is shown after the proof of 5.2-11. []

PROOF OF 5.2-9:
Part of the result follows from 5.1-5(2,4). For the remaining part, assume $P\text{-}G(g1,Ag1) \wedge P\text{-}G(g2,Ag2) \wedge out(Ag1)=out(Ag2) \wedge Local(Ag1) \cap Local(Ag2)=\{out(Ag2)\}$ and $pla\text{-}in \notin Local(Ag1) \cup Local(Ag2)$. Let c and $\underline{s}$ be given and let $trans(i)$ be an abbreviation of $\langle\langle pla[i-1], pla[i]\rangle, d[i]\rangle$.
Then $\bigsqcup\{cond(g1;c,\ g2;c)s \mid s \in \underline{s}\}$
$= \bigsqcup\{\ (g1;c)(Dt\text{-}cond(s)) \mid s \in \underline{s}\} \sqcup \bigsqcup\{(g2;c)(Df\text{-}cond(s)) \mid s \in \underline{s}\}$
EQ $\bigsqcup\{\ c(s) \mid s \in cl1[out(Ag1)]\} \sqcup (R \sqcup)cl1 \sqcup$
$\bigsqcup\{\ c(s) \mid s \in cl2[out(Ag2)]\} \sqcup (R \sqcup)cl2$
which was by assumptions where
$cl1 = Close(\ Ag1\langle pla\text{-}in, \lambda s.s\rangle\downarrow 2,\ pla\text{-}in,\ \{Dt\text{-}cond(s) \mid s \in \underline{s}\})$
$= \lambda pla.\ \{dn(\ldots(d1(Dt\text{-}cond(s))) \mid s \in \underline{s} \wedge \exists trans(1),\ldots,trans(n) \in Ag1\langle pla\text{-}in, \lambda s.s\rangle\downarrow 2$ so $pla[0]=pla\text{-}in \wedge pla[n]=pla\}$
(and by iii of $P\text{-}F(Ag1)$ and $pla\text{-}in \notin Local(Ag1)$)
$= \lambda pla.\ \{dn(\ldots(d1(s))) \mid s \in \underline{s}\ \exists trans(1),\ldots,trans(n) \in Ag1\langle pla\text{-}in, Dt\text{-}cond\rangle\downarrow 2$ so $pla[0] = pla\text{-}in \wedge pla[n] = pla\}$
$= Close(\ Ag1\langle pla\text{-}in, Dt\text{-}cond\rangle\downarrow 2,\ pla\text{-}in,\ \underline{s})$
$= Close(\ [Ag1 * Arecord(Dt\text{-}cond)]\langle pla\text{-}in, \lambda s.s\rangle\downarrow 2,\ pla\text{-}in,\ \underline{s})$
$= Close(\ \underline{trans1},\ pla\text{-}in,\ \underline{s})$
where $\underline{trans1} = [Ag1 * Arecord(Dt\text{-}cond)]\langle pla\text{-}in, \lambda s.s\rangle\downarrow 2$
Similarly,
$cl2 = Close(\ Ag2\langle pla\text{-}in, \lambda s.s\rangle\downarrow 2,\ pla\text{-}in,\ \{Df\text{-}cond(s) \mid s \in \underline{s}\})$
$= Close(\ \underline{trans2},\ pla\text{-}in,\ \underline{s})$
where $\underline{trans2} = [Ag2 * Arecord(Df\text{-}cond)]\langle pla\text{-}in, \lambda s.s\rangle\downarrow 2$

Since $out(Ag1)=out(Ag2)$ we get $\bigsqcup\{\ cond(g1;c,\ g2;c)s \mid s \in \underline{s}\}$
$= \bigsqcup\{\ c(s) \mid s \in (cl1 \sqcup cl2)[out(Ag2)]\ \} \sqcup (R \sqcup)(cl1 \sqcup cl2)$
(by using Lemma 2.1.1-7). Define
$cl = Close(\ ([Ag1 * Arecord(Dt\text{-}cond)]\ WITH\ [Ag2 * Arecord(Df\text{-}cond)])\langle pla\text{-}in, \lambda s.s\rangle\downarrow 2,\ pla\text{-}in,\ \underline{s})$
$= Close(\ \underline{trans1} \cup \underline{trans2},\ pla\text{-}in,\ \underline{s})$

It suffices to show $cl = cl1 \sqcup cl2$. The $\sqsupseteq$ inclusion is obvious by 5.2-2 so consider the $\sqsubseteq$ inclusion. Let pla' be arbitrary and assume $s' \in cl(pla')$. Then
$\exists s \in \underline{s}$
$\exists trans(1),\ldots,trans(n) \in \underline{trans1} \cup \underline{trans2}$
so $pla[0] = pla\text{-}in \wedge pla[n] = pla' \wedge s' = dn(\ldots(d1(s)))$
Assume (without loss of generality) that $trans(1) \in \underline{trans1}$ so that $k = \max\{j \mid trans(1),\ldots,trans(j) \in \underline{trans1}\}$ is well-defined. Also $k=n$, because otherwise $pla[k] \in Local(Ag1) \wedge pla[k] \in (Local(Ag2)-\{out(Ag2)\}) \cup \{pla\text{-}in\}$ which is a contradiction. Clearly $s' \in Close(\underline{trans1},\ pla\text{-}in,\ \underline{s})\ pla' = cl1\ pla'$. []

PROOF OF 5.2-11:

Note by 5.1-4 that $\mathrm{out}(Ag1) = plaT \wedge \mathrm{local}(Ag1) \sqsubseteq \mathrm{xpld}(occ\S\langle 1\rangle)$ and $\mathrm{out}(Ag2) = plaB \wedge \mathrm{local}(Ag2) \sqsubseteq \mathrm{xpld}(occ\S\langle 2\rangle)$.

1) We first develop a formulation of $\bigsqcup\{\, g[c](c')(s) \mid s \in \underline{s}\}$. In the sequel $pla \in \{pla\text{-}begin, plaB\}$ intuitively is the arc along which $\underline{s}$ has been propagated. Let

$$
\begin{aligned}
a &= \textstyle\bigsqcup\{\, g[c](c')(s) \mid s \in \underline{s}\}\\
&\ \mathrm{EQ}\ \textstyle\bigsqcup\{\, \mathrm{cond}(g2;c',\ \mathrm{attach}(pla\text{-}out);c)s \mid s \in cl1\langle pla,\underline{s}\rangle plaT\} \sqcup\\
&\qquad (R \textstyle\bigsqcup)(cl1\langle pla,\underline{s}\rangle)\\
&\qquad\quad \text{which was by P-G}(g1,Ag1)\text{ where}\\
&\qquad\quad cl1\langle pla,\underline{s}\rangle = \mathrm{Close}(\ Ag1\langle pla, \lambda s.s\rangle\psi 2,\ pla,\ \underline{s})\\
&= at \sqcup af \sqcup (R \textstyle\bigsqcup)(cl1\langle pla,\underline{s}\rangle)
\end{aligned}
$$

where

$$
\begin{aligned}
at &= \textstyle\bigsqcup\{\, (g2;c')(Dt\text{-}cond(s)) \mid s \in cl1\langle pla,\underline{s}\rangle plaT\}\\
&\ \mathrm{EQ}\ \textstyle\bigsqcup\{\, c'(s) \mid s \in cl2\langle pla,\underline{s}\rangle plaB\} \sqcup (R \textstyle\bigsqcup)(cl2\langle pla,\underline{s}\rangle)\\
&\qquad \text{which was by P-G}(g2,Ag2)\text{ where}\\
&\qquad cl2\langle pla,\underline{s}\rangle = \mathrm{Close}(\ Ag2\langle plaT, \lambda s.s\rangle\psi 2, plaT,\\
&\qquad\qquad\qquad \{Dt\text{-}cond(s) \mid s \in cl1\langle pla,\underline{s}\rangle plaT\})\\
&\qquad\qquad = \mathrm{Close}(\ Ag2\langle plaT, Dt\text{-}cond\rangle\psi 2,\ plaT,\ cl1\langle pla,\underline{s}\rangle plaT)\\
&\qquad \text{reasoning as in Lemma 5.2-9}
\end{aligned}
$$

and

$$
\begin{aligned}
af &= \textstyle\bigsqcup\{\, (\mathrm{attach}(pla\text{-}out);c)(Df\text{-}cond(s)) \mid s \in cl1\langle pla,\underline{s}\rangle plaT\}\\
&\ \mathrm{EQ}\ \textstyle\bigsqcup\{\, c(s) \mid s \in cl3\langle pla,\underline{s}\rangle\ pla\text{-}out\} \sqcup (R \textstyle\bigsqcup)(cl3\langle pla,\underline{s}\rangle)\\
&\qquad \text{which is by 5.2-5 where}\\
&\qquad cl3\langle pla,\underline{s}\rangle = \mathrm{Close}(\ A\mathrm{attach}(pla\text{-}out)\langle plaT, \lambda s.s\rangle\psi 2,\ plaT,\\
&\qquad\qquad\qquad \{Df\text{-}cond(s) \mid s \in cl1\langle pla,\underline{s}\rangle plaT\})\\
&\qquad\qquad = \mathrm{Close}(\ A\mathrm{attach}(pla\text{-}out)\langle plaT, Df\text{-}cond\rangle\psi 2,\ plaT,\\
&\qquad\qquad\qquad cl1\langle pla,\underline{s}\rangle plaT)\\
&\qquad \text{reasoning as above.}
\end{aligned}
$$

Then

$$
\begin{aligned}
a\ \mathrm{EQ}\ & \textstyle\bigsqcup\{\, c'(s) \mid s \in cl2\langle pla,\underline{s}\rangle plaB\} \sqcup\\
& \textstyle\bigsqcup\{\, c(s) \mid s \in cl3\langle pla,\underline{s}\rangle pla\text{-}out\} \sqcup\\
& (R \textstyle\bigsqcup)(cl1\langle pla,\underline{s}\rangle \sqcup cl2\langle pla,\underline{s}\rangle \sqcup cl3\langle pla,\underline{s}\rangle)\\
=\ & \textstyle\bigsqcup\{\, c'(s) \mid s \in [cl1\langle pla,\underline{s}\rangle \sqcup cl2\langle pla,\underline{s}\rangle \sqcup cl3\langle pla,\underline{s}\rangle]plaB\} \sqcup\\
& \textstyle\bigsqcup\{\, c(s) \mid s \in [cl1\langle pla,\underline{s}\rangle \sqcup cl2\langle pla,\underline{s}\rangle \sqcup cl3\langle pla,\underline{s}\rangle]pla\text{-}out\} \sqcup\\
& (R \textstyle\bigsqcup)[\ cl1\langle pla,\underline{s}\rangle \sqcup cl2\langle pla,\underline{s}\rangle \sqcup cl3\langle pla,\underline{s}\rangle]
\end{aligned}
$$

where the last step is because $cl1\langle pla,\underline{s}\rangle plaB = cl3\langle pla,\underline{s}\rangle plaB = \emptyset$ and $cl1\langle pla,\underline{s}\rangle pla\text{-}out = cl2\langle pla,\underline{s}\rangle pla\text{-}out = \emptyset$.

2) We now derive a formulation of $\bigsqcup\{(g[c])^{k} c'\ s \mid s \in \underline{s}\}$. This formulation is not the desired one but serves as a useful auxiliary stage. By calculations for k=1,2 one may guess (for $k \geq 1$):

$$
\begin{aligned}
\textstyle\bigsqcup\{(g[c])^{k} c' s \mid s \in \underline{s}\}\ \mathrm{EQ}\ & \textstyle\bigsqcup\{\, c'(s) \mid s \in bk\ plaB\} \sqcup\\
& \textstyle\bigsqcup\{\, c(s) \mid s \in [bk \sqcup \ldots \sqcup b1]\ pla\text{-}out\} \sqcup\\
& (R \textstyle\bigsqcup)\ [bk \sqcup \ldots \sqcup b1]
\end{aligned}
$$

where

$$
\begin{aligned}
b1 &= cl1\langle pla\text{-}begin,\underline{s}\rangle \sqcup cl2\langle pla\text{-}begin,\underline{s}\rangle \sqcup cl3\langle pla\text{-}begin,\underline{s}\rangle\\
bk &= cl1\langle plaB,\ b[k\text{-}1]\ plaB\rangle \sqcup cl2\langle plaB,\ b[k\text{-}1]\ plaB\rangle \sqcup cl3\langle plaB,\ b[k\text{-}1]\ plaB\rangle
\end{aligned}
$$

It may be helpful to note that $bk \in Pla\ \text{-t>}\ \mathcal{P}(S)$ so $(R \bigsqcup)(bk \sqcup \ldots \sqcup b1) \in Pla\ \text{-t>}\ S$. We verify the guess by induction on k.

k=1: The result is immediate from "1)".

k+1>2: We calculate

$$\textstyle\bigsqcup\{\ (g[c])^{k+1}\ c'\ s \mid s \in \underline{s}\}$$

= ⊔{ (g[c])^k(g[c];c')s | s∈s}
EQ ⊔{ (g[c];c')s | s∈(bk plaB)} ⊔ {c(s) | s∈([bk⊔...⊔b1] pla-out)} ⊔
(R ⊔)[bk⊔...⊔b1]
which was by induction hypothesis; by "1)" we get:
EQ ⊔{ c'(s) | s∈(b[k+1] plaB)} ⊔
⊔{ c(s) | s∈(b[k+1]⊔...⊔b1) pla-out}⊔
(R ⊔)(b[k+1]⊔...⊔b1)

3) We now prove ∀k≥1: bk = ak because then the desired result easily follows from "2)". The proof is by induction on k.

k=1: The proof can be obtained by adapting (simplifying) the proof below.

k+1>2: We have by the inductive hypothesis:
b[k+1] = cl1<plaB,(bk plaB)> ⊔ cl2<plaB,(bk plaB)> ⊔ cl3<plaB,(bk plaB)>
= cl1<plaB,(ak plaB)> ⊔ cl2<plaB,(ak plaB)> ⊔ cl3<plaB,(ak plaB)>
Abbreviate
trans1 = Ag<pla-begin,λs.s>ψ2
trans2 = {< <plaB,plaR>, λs.s>}
and let pla' be arbitrary. We now show b[k+1] pla' = a[k+1] pla'. We use trans(i) as a shorthand for < <pla[i-1],pla[i]>, d[i]>.

Inclusion ⊆:

i) Assume s' ∈ cl1<plaB,(ak plaB)> pla'. Then
∃ s∈s
∃ trans(1),...,trans(m) ∈ trans1∪trans2
so pla[0] = pla-begin ∧ pla[m] = plaB ∧
|{i | i≤m ∧ trans(i) ∈ trans2}| = k-1 ∧
dm(...(d1(s))) ∈ (ak plaB)
Also ∃ trans(m+1),...,trans(n) ∈ Ag1<plaB,λs.s>ψ2
so pla[m] = plaB ∧ pla[n] = pla' ∧
s' = dn(...(dm(...(d1(s)))))
By 5.1-4(b,c) we have
Ag1<plaB,λs.s>ψ2 ⊆ Ag1<pla-begin,λs.s>ψ2 ∪ trans2 ⊆ trans1∪trans2.
From "ii" of P-F(Ag1) and by 5.1-4(b): j∈{m+2,...,n} =>
pla[j-1] ∈ xpld(occ§<1>) => pla[j-1] ≠ plaB => trans(j) ∉ trans2. Also
trans(m+1) = <<plaB,plaR>,λs.s> follows from pla[m] = plaB and 5.1-4(c)
so that trans(m+1) ∈ trans2. This shows s' ∈ a[k+1] pla'.

ii) Let s' ∈ cl2<plaB, ak plaB>. Then (as above)
∃ s∈s
∃ trans(1),...,trans(n) ∈ trans1∪trans2
so pla[0] = pla-begin ∧ pla[n] = plaT ∧
dn(...(d1(s))) ∈ cl1<plaB, ak plaB> plaT ⊆ a[k+1] plaT
Also ∃ trans(n+1),...,trans(q) ∈ Ag2<plaT,Dt-cond>ψ2 ⊆ trans1
so pla[n] = plaT ∧ pla[q] = pla' ∧
s' = dq(...(dn(...(d1(s)))
Then s' ∈ a[k+1] pla' because Ag2<plaT, Dt-cond>ψ2 ∩ trans2 = ∅
which is by "ii" of P-F(Ag2).

iii) Let s' ∈ cl3<plaB, ak plaB>. As before s' ∈ a[k+1] pla'.

Inclusion ⊇:
Assume s' ∈ a[k+1] pla'. Then
∃ s∈s

∃ trans(1),...,trans(m) ∈ trans1 ∪ trans2
so pla[0] = pla-begin ∧ pla[m] = pla'∧
s' = dm(...(d1(s))) ∧
|{i | trans(i) ∈ trans2}| = k
Define q = max{j | trans(j) ∈ trans2}. Obviously q is well-defined because k>1. Then trans(q) = <<plaB,plaR>,λs.s> and d[q-1](...(d1(s))) ∈ (ak plaB). By 5.1-4(c) we have trans(q) ∈ Ag1<plaB,λs.s>ψ2 so that
q1 = max{j | trans(q),...,trans(j) ∈ Ag1<plaB,λs.s>ψ2} is well-defined. Clearly dq1(...(d1(s))) ∈ cl1<plaB,(a[k] plaB)> pla[q1] so that we are finished if q1 = m. Otherwise q1<m and trans(q+1) is in Ag2<plaT,Dt-cond>ψ2 or Aattach(pla-out)<plaT,Df-cond>ψ2.

i) Assume trans(q1+1) ∈ Ag2<plaT,Dt-cond>ψ2. Then pla[q1] ∈ xpld(occ§<1>) ∧ pla[q1] ∈ xpld(occ§<2>)∪{plaT} so pla[q1] = plaT. Furthermore
q2 = max{j | trans(q1+1),...,trans(j) ∈ Ag2<plaT,Dt-cond>ψ2} is well-defined and dq2(...(dq1(...(d1(s))) ∈ cl2<plaB,(ak plaB)> pla[q2].

We are finished if q2 = m. To see that q2 = m assume otherwise: Then trans[q2+1] ∈ Ag1<pla-begin,λs.s>ψ2 ∪ Aattach(pla-out)<plaT,Df-cond>ψ2 implying [by P-F(Ag1) and 5.2-5] that
pla[q2] ∈ xpld(occ§<1>)∪{pla-begin} which is impossible when P-F(Ag2) implies pla[q2] ∈ xpld(occ§<2>).

ii) Assume trans(q1+1) ∈ Aattach(pla-out)<plaT,Df-cond>ψ2. Then pla[q1] ∈ xpld(occ§<1>) ∧ pla[q1] ∈ {plaT} so that pla[q1] = plaT and d[q1+1](...(d1(s))) ∈ cl3<plaB,(ak plaB)> pla[q1+1]. We are done if q1+1 = m. This is the case because
pla[q1+1] = pla-out ∉ xpld(occ§<1>) ∪ xpld(occ§<2>) ∪ {pla-begin}. []

PROOF OF 5.2-8:
The proof is by structural induction. We only show the proof for WHILE...DO...OD. Part of the result follows by 5.1-4(a). For the remaining part, assume pure[Occ](occ) and make the abbreviations of Lemma 5.2-11. To save space they are not repeated here.

1) Our first goal is to show (for arbitrary c and s)
⊔{FIX(g[c])s | s∈s}
EQ ⊔{c(s) | s∈cl(out(Ag))} ⊔ (R ⊔ cl)
where cl = Close([Ag ALSO {<<plaB,plaR>,λs.s>}]
<pla-begin,λs.s>ψ2, pla-begin, s)
By 5.1-4 and 5.1-5 this amounts to
P-G(λc.FIX(g[c]), Ag ALSO {<<plaB,plaR>,λs.s>}) except that pla-in of P-G can only be pla-begin.

By the induction hypothesis and Lemma 5.2-4 the result of Lemma 5.2-11 holds in this setting. So
⊔{ FIX(g[c]) s | s∈s}
= ⊔{ ⊔{ $(g[c])^n$ ⊥ s | n≥1} | s∈s}
= ⊔{ ⊔{ $(g[c])^n$ ⊥ s | s∈s} | n≥1}
which was by Lemma 2.1.1-7 twice, and by Lemma 5.2-11 we get:
EQ ⊔{ ⊔{ c(s) | s∈([an⊔...⊔a1]pla-out)} ⊔ (R ⊔)(an⊔...⊔a1) | n≥1}
= x ⊔ y
where
x = ⊔{ ⊔{ c(s) | s∈([an⊔...⊔a1] pla-out)} | n≥1}

= ⊔{ c(s) | ∃n≥1: s∈([an⊔... ⊔a1] pla-out)}
which was by Lemma 2.1.1-7
= ⊔{ c(s) | s ∈ ⊔((⊔{an|n≥1}) pla-out)}
and
y = ⊔{ (R ⊔)(an⊔...⊔a1) | n≥1}
= ⊔{λpla. ⊔[(an pla)∪... ∪(a1 pla)] | n≥1}
= λpla. ⊔{ ⊔[(an pla)∪...∪(a1 pla)] | n≥1}
= λpla. ⊔(∪{(an pla)∪...∪(a1 pla) | n≥1})
which was by Lemma 2.1.1-7
= λpla. ⊔((⊔{an|n≥1})pla)
= (R ⊔) (⊔{an|n≥1})
so ⊔{ FIX(g[c])s | s∈s̲}
EQ ⊔{ c(s) | s∈((⊔{an|n≥1})pla-out)} ⊔ (R ⊔) (⊔{an|n≥1})
This shows the result because ⊔{an|n≥1}
= ⊔{ Luk(Ag<pla-begin,λs.s>↓2, pla-begin, s̲, n,
{<<plaB,plaR>,λs.s>}) | n≥0}
= Close(Ag<pla-begin,λs.s>↓2 ∪ {<<plaB,plaR>,λs.s>}, pla-begin, s̲)
= Close([Ag ALSO {<<plaB,plaR>,λs.s>}]<pla-begin,λs.s>↓2, pla-begin,s̲)

2) By Lemma 5.2-5 P-G(attach(pla-begin), Aattach(pla-begin)) so that "1)" and reasoning as in the proof of Lemma 5.2-7 yields
P-G(𝒞⟦WHILE exp DO cmd OD⟧ occ, A𝒞⟦WHILE exp DO cmd OD⟧ occ) where we have used A𝒞⟦WHILE exp DO cmd OD⟧ occ =
(Ag ALSO {<<plaB,plaR>,λs.s>})*Aattach(pla-begin). This ends the proof of the WHILE exp DO cmd OD case. []

PROOF OF 5.2-12:
Lemmas 5.2-4, 5.2-8 and 5.2-7 imply
P-G(λc.𝒟⟦dcl⟧<1>;𝒞⟦cmd⟧<2>;c, A𝒞⟦cmd⟧<2>*A𝒟⟦dcl⟧<1>). Since
< <>, "(pro"> ∉ Local(A𝒞⟦cmd⟧<2>*A𝒟⟦dcl⟧<1>) we get ∀i̲n̲p̲∈𝒫(Inp):
𝒫ind⟦BEG dcl IN cmd END⟧i̲n̲p̲
= ⊔{ (𝒟⟦dcl⟧<1>;𝒞⟦cmd⟧<2>;finish)s |
s ∈ { uabs{ <λide."nil" inVal,inp,<>,<> > | inp∈i̲n̲p̲}}}
EQ (R ⊔) (Close(A𝒫⟦BEG dcl IN cmd END⟧, <<>,"(pro">,
{ uabs{ <λide."nil" inVal, inp, <>, <>> | inp∈i̲n̲p̲}}))
= λpla. ⊔{dn(...(d1(uabs{<λide."nil" inVal, inp, <>, <>> | inp∈i̲n̲p̲}))) |
n≥1 ∧ ∃<<pla[0],pla[1]>,d1>,...,<<pla[n-1],pla[n]>,dn>
∈ A𝒫⟦BEG dcl IN cmd END⟧
so that pla[0] = < <>,"(pro"> ∧ pla[n] = pla } []

PROOF OF 6.1.2-3:

The proof is an extension of the proof of Lemma 4.2.1-2 (for the std case):

1) For any $c \in C$, $ss1 \in S \text{ -c> } R$ and $ss2 \in S \text{ -c> } R$ we can prove $(c \oplus ss1) \oplus ss2 = c \oplus (ss1 * ss2)$ as in 4.2.1-2.

2) For $g \in Par \text{ -t> } C \text{ -c> } C$ any primitive function and $c \in C$ we can prove $g(par)(c) = c \oplus [Pg(par)]$ as in 4.2.1-2.

3) For $pla \in Pla$ and $c \in C$ we can prove $attach(pla)(c) = c \oplus [Pattach(pla)]$ as in 4.2.1-2.

4) For $c \in C$, $ss1 \in S \text{ -c> } R$ and $ss2 \in S \text{ -c> } R$ we have $c \oplus Pcond(ss1,ss2) = cond(c \oplus ss1, c \oplus ss2)$. To see this: $cond(c \oplus ss1, c \oplus ss2)$

$= \lambda s.\ Vcond(s) \rightarrow (Scond(s) \rightarrow c \oplus ss1,\ c \oplus ss2)(Bcond(s)),\ \text{"wrong"}inA$

$= \lambda s.\ Vcond(s) \rightarrow [\ c \oplus (Scond(s) \rightarrow ss1,\ ss2)](Bcond(s)),\ \text{"wrong"}inA$

(which was by cases of Scond(s))

$= c \oplus [\ \lambda s.\ Vcond(s) \rightarrow [\ Scond(s) \rightarrow ss1,\ ss2\ ](Bcond(s)),\ \text{"wrong"}inR\ ]$

(which was by cases of Vcond(s))

$= c \oplus Pcond(ss1,ss2)$

5) Define $strict\ c = \lambda r.\ r \in Sta \rightarrow c(r \mid Sta),\ \text{"wrong"}\ inA$. It is not difficult to show that $strict \in [S \text{ -c> } A] \text{ -c> } [R \text{ -c> } A]$.

6) The result is shown by a structural induction. We only consider the "difficult" case WHILE exp DO cmd OD. So assume root(tree).str = "WHILE exp DO cmd OD" and $c \in C$ and abbreviate:

$(P)g0 = (P)attach\langle root(tree).lab, \text{"("}\rangle$

$(P)g1 = (P)\mathcal{T}(son1(tree))$

$(P)g2 = (P)\mathcal{T}(son2(tree))$

$(P)g3 = (P)attach\langle root(tree).lab, \text{")"}\rangle$

$cn\ = [\ \lambda c'.\ g1;cond(g2;c',\ g3;c)]^n \perp$

$ssn = [\ \lambda ss.\ Pcond(ss*Pg2,\ Pg3)*Pg1]^n \perp$

It is easy to show
$c \oplus ss = c' \Rightarrow c \oplus [\ Pcond(ss*Pg2,Pg3)*Pg1\ ] = g1;cond(g2;c',\ g3;c)$ using induction hypotheses and "4", "3" and "1". Since $c \oplus \perp = \perp$ a proof by induction yields $\forall n \geq 0$: $cn = c \oplus ssn$. We now want to deduce $\sqcup\{cn \mid n \geq 0\} = c \oplus (\sqcup\{ssn \mid n \geq 0\})$. We have $\sqcup\{cn \mid n \geq 0\}$

$= \sqcup\{\ c \oplus ssn \mid n \geq 0\ \}$

$= \sqcup\{\ (strict\ c) \circ ssn \mid n \geq 0\ \}$

$= (strict\ c) \circ (\sqcup\{ssn \mid n \geq 0\})$

which was because (strict c) continuous and $\{ssn \mid n \geq 0\}$ a directed set

$= c \oplus (\sqcup\{ssn \mid n \geq 0\})$

Hence $g0;(\sqcup\{cn \mid n \geq 0\}) = c \oplus ((\sqcup\{ssn \mid n \geq 0\}) * Pg0)$ showing $\mathcal{T}(tree)c = c \oplus P\mathcal{T}(tree)$. []

PROOF OF 6.1.3-2:
We first prove 4 auxiliary results and then (case 5) perform a proof by structural induction.

1) We show P-G(std-ss1,sts-ss1) $\wedge$ P-G(std-ss2,sts-ss2) =>
P-G(std-ss1*std-ss2,sts-ss1*sts-ss2)
Let $\underline{\text{sta}}\in\mathcal{P}$(Sta) and calculate (sts-ss1*sts-ss2)$\underline{\text{sta}}$ $\downarrow$1
= { (std-ss1(std-ss2(sta) |Sta) |Sta) | sta$\in\underline{\text{sta}}$ $\wedge$(std-ss2(sta)ESta)=true $\wedge$
(std-ss1(std-ss2(sta) |Sta) ESta) = true }
= { ((std-ss1*std-ss2)(sta) |Sta) | sta$\in\underline{\text{sta}}$ $\wedge$
((std-ss1*std-ss2)(sta) ESta) = true }

2) For any primitive function g it is easy to show
P-G(std-Pg(par),sts-Pg(par)).

3) It is easy to show P-G(std-Pattach(pla), sts-Pattach(pla)).

4) We show P-G(std-ss1,sts-ss1) $\wedge$ P-G(std-ss2,sts-ss2) =>
P-G(std-Pcond(std-ss1,std-ss2), sts-Pcond(sts-ss1,sts-ss2)). Let
$\underline{\text{sta}}\in\mathcal{P}$(Sta) and calculate: sts-Pcond(sts-ss1,sts-ss2) $\underline{\text{sta}}$ $\downarrow$1
= { (std-ss1(std-Bcond(sta)) |Sta) | sta$\in\underline{\text{sta}}$ $\wedge$ std-Vcond(sta) = true $\wedge$
std-Scond(sta) = true $\wedge$ (std-ss1(std-Bcond(sta)) ESta) = true}
$\cup$ { ... }
= { (std-Pcond(std-ss1,std-ss2)(sta) |Sta) | sta$\in\underline{\text{sta}}$ $\wedge$
(std-Pcond(std-ss1,std-ss2)(sta) ESta) = true}

5) We now perform the structural induction. We only consider the "difficult" case where root(tree).str = "WHILE exp DO cmd OD". Abbreviate
Pg0 = Pattach<root(tree).lab,"(">
Pg1 = P$\mathcal{T}$(son1(tree))
Pg2 = P$\mathcal{T}$(son2(tree))
Pg3 = Pattach<root(tree).lab,")">

ss[n][ss'] = [λss". Pcond(ss"*Pg2,Pg3) * Pg1 $]^n$ ss'

From P-G($\perp$,$\perp$), the induction hypotheses and "1", "3" and "4" a proof by induction yields $\forall$n$\geq$0: P-G(std-ss[n][$\perp$], sts-ss[n][$\perp$]). Our goal is to show P-G($\sqcup${std-ss[n][$\perp$] | n$\geq$0}, $\sqcup${sts-ss[n][$\perp$] | n$\geq$0}). To do so we first show an auxiliary result about {std-ss[n][$\perp$] | n$\geq$0} and then prove the desired P-G(...,...).

i) Auxiliary Result:
The result
$\forall$sta$\in$Sta: {std-ss[n][$\perp$]sta | n$\geq$0} = {$\perp$, $\sqcup${std-ss[n][$\perp$]sta | n$\geq$0}}
easily follows from (omitting prefix std- in the rest of "i)")
$\forall$n$\geq$0: $\forall$sta$\in$Sta: [ss[n][$\perp$]sta $\neq\perp$ => ss[n][$\perp$]sta = ss[n+1][$\perp$]sta]
From ss[n+1][$\perp$] = ss[n][ss[1][$\perp$]] we only need to show by induction in n:
$\forall$sta$\in$Sta: [ss[n][$\perp$]sta $\neq\perp$ => ($\forall$ss': ss[n][$\perp$]sta = ss[n][ss']sta)]
For n=0 the result is obvious so assume it for n$\geq$0 and prove it for n+1. Let sta and ss' be arbitrary and assume ss[n+1][$\perp$]sta $\neq\perp$. Our strategy is to prove ss[n+1][ss'] equivalent to an expression which does not depend on ss' (because then the desired result holds).

We have (from ss[n+1][ss'] = ss[1][ss[n][ss']])
ss[n+1][ss']sta = (Pcond(ss[n][ss']*Pg2,Pg3) * Pg1)sta
There are four possibilities of (Pg1 sta). If

(Pg1 sta) ∈ {⊥,⊤,"wrong"inR} then ss[n+1][ss']sta = (Pg1 sta) is
independent of ss'. Otherwise (Pg1 sta) = sta' inR for sta' independent
of ss'. Proceeding like this we are left with:
  ss[n+1][ss']sta = ss[n][ss']sta"
for sta" independent of ss'.

From ss[n+1][⊥]sta≠⊥ we get ss[n][⊥]sta"≠⊥ so by induction hypothesis
ss[n+1][ss']sta = ss[n][ss']sta" = ss[n][⊥]sta" is independent of ss'.

ii) To show P-G(⊔{std-ss[n][⊥] | n≥0}, ⊔{sts-ss[n][⊥] | n≥0}) we calculate
for sta∈𝒫(Sta): ((⊔{sts-ss[n][⊥] | n≥0})sta)↓1
 = ∪{{ ((std-ss[n][⊥]sta) |Sta) | sta∈sta ∧
 (std-ss[n][⊥]sta ƐSta) = true } | n≥0 }
 which was by ∀n≥0: P-G(std-ss[n][⊥],sts-ss[n][⊥])
 = { ((std-ss[n][⊥]sta) |Sta) | sta∈sta ∧
 ∃n≥0: (std-ss[n][⊥]sta ƐSta) = true }
 = { ⊔{ std-ss[n][⊥]sta | n≥0} | sta∈sta ∧
 (⊔{std-ss[n][⊥]sta | n≥0} ƐSta) = true }
 which was because i) implies
 ∃n≥0: (std-ss[n][⊥]sta ƐSta) = true <=>
 ((⊔{std-ss[n][⊥]sta | n≥0}) ƐSta) = true
 and
 (std-ss[n][⊥]sta ƐSta) = true =>
 std-ss[n][⊥]sta = ⊔{ std-ss[m][⊥]sta | m≥0 }
Then P-G(P𝒯std(tree), P𝒯sts(tree)) easily follows. []

PROOF OF 6.1.3-5:
The proof is by structural induction. We only show the proof of the case
root(tree).str = "WHILE exp DO cmd OD". Abbreviate (omitting prefix sts-).
 lab = root(tree).lab
 Pg0 = Pattach<lab,"(">
 Pg1 = P𝒯(son1(tree))
 Pg2 = P𝒯(son2(tree))
 Pg3 = Pattach<lab,")">
 ss[n] = [λss. Pcond(ss*Pg2,Pg3) * Pg1]^n ⊥
 F(sta) = Pg2(Dt-cond(Pg1(sta)↓1))↓1

Calculating ss[n+1]sta
 = ss[n](F(sta)) ⊔ Pg3(Df-cond(Pg1(sta)↓1)) ⊔
 <⊥, Pg2(Dt-cond(Pg1(sta)↓1))↓2 ⊔ Pg1(sta)↓2 >
makes it easy to show by induction on n≥0 that:
ss[n]sta = ⊔{< Df-cond(Pg1(F^j(sta))↓1)
 , ⊥[Df-cond(Pg1(F^j(sta))↓1) / <lab,")">] ⊔
 Pg2(Dt-cond(Pg1(F^j(sta))↓1))↓2 ⊔ Pg1(F^j(sta))↓2>
 | 0≤j<n }
So that (by Lemma 2.1.1-7) ⊔{ ss[n]sta | n≥0 }
 = ⊔{ < Df-cond(Pg1(F^n(sta))↓1)
 , ⊥[Df-cond(Pg1(F^n(sta))↓1) / <lab,")">] ⊔
 Pg2(Dt-cond(Pg1(F^n(sta))↓1))↓2 ⊔
 Pg1(F^n(sta))↓2 > | n≥0 }
and thereby P𝒯(tree)sta
 = < ⊔{ Df-cond(Pg1(F^n(sta))↓1) | n≥0}
 , ⊥[⊔{ Df-cond(Pg1(F^n(sta))↓1) | n≥0} / <lab,")">] ⊔
 ⊔{ Pg2(Dt-cond(Pg1(F^n(sta))↓1))↓2 | n≥0} ⊔
 ⊔{ Pg1(F^n(sta))↓2 | n≥0} ⊔
 ⊥[sta / <lab,"(">] >

By this result, by dom(tree) = dom(son1(tree)) $\cup$ dom(son2(tree)) $\cup$ {lab}, by induction hypotheses and assumptions on == it is straight-forward to show Pa(tree) $\wedge$ Pb(tree) $\wedge$ Pc(tree). []

PROOF OF 6.1.3-7:

The result is shown by a structural induction. So let tree $\in$ Tree $\wedge$ lab $\in$ dom(tree) $\wedge$ $\underline{sta} \in \mathcal{P}$(Sta) and assume root(tree $\underline{at}$ lab).cat $\in$ {"pro","dcl","cmd","exp"}. We are to prove Pd(tree,lab) $\wedge$ Pe(tree,lab). This is done by cases of root(tree).str using that root(tree).cat $\notin$ {"ide","ope","bas"}. Most cases consider two possibilities: lab = root(tree).lab and lab $\neq$ root(tree).lab. We are rigorous about the first possibility and slightly less rigorous about the second. Below we only show the proof of the case root(tree).str = "WHILE exp DO cmd OD". We omit the prefix sts- and abbreviate:

Pg0 = Pattach<root(tree).str,"(">
Pg1 = Pπ(son1(tree))
Pg2 = Pπ(son2(tree))
Pg3 = Pattach<root(tree).str,")">

F = $\lambda\underline{sta}'$. Pg2(Dt-cond(Pg1($\underline{sta}'$)$\downarrow$1))$\downarrow$1
a = Pπ(tree)$\underline{sta}$ $\downarrow$2

From the calculations in the proof of Lemma 6.1.3-5

a pla = $\perp$[$\sqcup${Df-cond(Pg1(F^n($\underline{sta}$))$\downarrow$1) | n$\geq$0} / <lab,")">] pla $\sqcup$
$\sqcup${ (Pg2(Dt-cond(Pg1(F^n($\underline{sta}$))$\downarrow$1))$\downarrow$2) pla | n$\geq$0} $\sqcup$
$\sqcup${ (Pg1(F^n($\underline{sta}$))$\downarrow$2)pla | n$\geq$0} $\sqcup$
$\perp$[$\underline{sta}$ / <lab,"(">] pla

i) Assume lab = root(tree).lab

To show Pd(tree,lab) we calculate:

a <lab,")"> = Pπ(tree)$\underline{sta}$ $\downarrow$1 by Pb(tree)
= Pπ(tree) (a<lab,"(">) $\downarrow$1 by Pa(tree)

To show Pe(tree,lab) we calculate:

a <lab§<1>,"("> = $\sqcup${ (Pg1(F^n($\underline{sta}$))$\downarrow$2) <lab§<1>,"("> | n$\geq$0} by Pc(...)
= $\sqcup${ F^n($\underline{sta}$) | n$\geq$0 } by Pa(son1(tree))
= $\sqcup${ F^n(a<lab,"(">) | n$\geq$0} by Pa(tree)

a <lab§<2>,"("> = $\sqcup${ (Pg2(Dt-cond(Pg1(F^n($\underline{sta}$))$\downarrow$1))$\downarrow$2) <lab§<2>,"("> | n$\geq$0}
which was by Pc(...)
= Dt-cond($\sqcup${ Pg1(F^n($\underline{sta}$))$\downarrow$1 | n$\geq$0}) by Pa(son2(tree))
= Dt-cond($\sqcup${ (Pg1(F^n($\underline{sta}$))$\downarrow$2) <lab§<1>,")"> | n$\geq$0})
which was by Pb(son1(tree))
= Dt-cond(a<lab§<1>,")">) by Pc(...)

ii) Assume lab $\neq$ root(tree).lab

Then lab = [root(tree).lab]§<i,...> for i $\in$ {1,2}. Since the cases are similar we consider only i=1 below.

To be rigorous the proof of Pd(tree,lab) $\wedge$ Pe(tree,lab) is by cases of root(tree $\underline{at}$ lab).str. To condense the proof we choose to give a general discussion covering all cases.

By inspection of the definitions of Pd(tree,lab) and Pe(tree,lab) it is easy to see that verification of Pd(tree,lab) $\wedge$ Pe(tree,lab) amounts to showing one or more equalities

(a pla1) = G[tree $\underline{at}$ lab] (a pla2)

for pla1$\downarrow$1 = lab§<...> $\wedge$ pure[Pla](pla1) and pla2$\downarrow$1 = lab§<...> $\wedge$ pure[Pla](pla2) and G[tree $\underline{at}$ lab]: $\mathcal{P}$(Sta) -> $\mathcal{P}$(Sta) a complete-$\sqcup$-morphism (Corollary 6.1.3-3) that does not depend upon tree except tree at lab.

Obviously,
(a pla1) = $\sqcup${ (Pg1(F^n(sta))$\downarrow$2)pla1 | n$\geq$0 }   by Pc(...) and i=1
 = $\sqcup${ G[son1(tree) at lab] ((Pg1(F^n(sta))$\downarrow$2)pla2) | n$\geq$0 }
  which was by Pd'(son1(tree)) $\wedge$ Pe'(son1(tree))
 = G[son1(tree) at lab] ($\sqcup${ (Pg1(F^n(sta))$\downarrow$2)pla2 | n$\geq$0})
  which was by G[...] a complete-$\sqcup$-morphism
 = G[tree at lab] (a pla2)
  which was by Pc(...) $\wedge$ i = 1 []

PROOF OF 6.1.4-2:
The proof is by induction on #lab. For #lab=0 it is obvious so we only need to show P1(t1,t2,lab§<i>,sta) => P1(t1,t2,lab,sta). Therefore, assume P1(t1,t2,lab§<i>,sta).

Let t$\in$Tree(<>,"pro") and t1',t2' $\in$ Tree(<>,"pro") $\cup$ Tree(<>,"dcl") $\cup$ Tree(<>,"cmd") $\cup$ Tree(<>,"exp") be such that t1 = t(lab§<i> <- t1') and similarly for t2. Define (for j=1,2)
 tj" = {<lab',prod'> | <lab§lab',prod'> $\in$ tj at lab}.
Clearly t1 = t(lab <- t1") and t2 = t(lab <- t2") as well as t1",t2" $\in$ Tree(<>,"pro") $\cup$ Tree(<>,"dcl") $\cup$ Tree(<>,"cmd") $\cup$ Tree(<>,"exp"). This leaves us with showing P2(t1,t2,lab,sta).

This proof is by cases of root(t1 at lab).str. We only show the proof for the case root(t1 at lab).str = "WHILE exp DO cmd OD". The definition of Tree ensures that i$\in${1,2}. Let i'=3-i so that {i,i'}={1,2}. Then P$\mathcal{J}$std(t1 at lab§<i'>) = P$\mathcal{J}$std(t2 at lab§<i'>) so that P2(t1,t2,lab§<i'>,sta) holds. We only use this weaker characterization because we then avoid having to consider the cases i=1 and i=2 separately. Abbreviate for j$\in${1,2}:
 Pgj0 = Pattach<lab,"(">
 Pgj1 = P$\mathcal{J}$(tj at lab§<1>)
 Pgj2 = P$\mathcal{J}$(tj at lab§<2>)
 Pgj3 = Pattach<lab,")">
 ssj[n] = [λss.Pcond(ss*Pgj2,Pgj3)*Pgj1]^n $\perp$
 sts-a = (P$\mathcal{J}$sts(t1)sta)$\downarrow$2

We must show $\forall$sta $\in$ sts-a<lab,"("> that
 P$\mathcal{J}$std(t1 at lab)sta = P$\mathcal{J}$std(t2 at lab)
Now P$\mathcal{J}$std(tj at lab) sta
 = (($\sqcup${std-ssj[n] | n$\geq$0}) * std-Pgj0) sta
 = $\sqcup${std-ssj[n](sta) | n$\geq$0}
so it suffices to show $\forall$n$\geq$0 $\forall$sta$\in$(sts-a<lab,"(">) that
 std-ss1[n](sta) = std-ss2[n](sta)
We do this by showing by induction in n
 $\forall$sta$\in$sts-a<lab§<1>,"(">: std-ss1[n](sta) = std-ss2[n](sta)
This is an appropriate result because Pe'(t1) implies sts-a<lab,"("> $\subseteq$ sts-a<lab§<1>,"(">. The case n=0 is obvious so we are left with showing it for n+1 assuming it holds for n$\geq$0.

Let sta $\in$ sts-a<lab§<1>,"(">. We show std-ss1[n+1](sta) = std-ss2[n+1](sta) by showing std-ssj[n+1]sta equivalent to an expression that does not depend on j. We have
 std-ssj[n+1]sta = [std-Pcond(std-ssj[n]*std-Pgj2,std-Pgj3)*std-Pgj1](sta)
By P2(t1,t2,lab§<1>,sta) we know std-Pg11(sta) = std-Pg21(sta) = std-a1 for some std-a1. If (std-a1ESta) $\neq$ true then std-ssj[n+1]sta = std-a1 is "independent of j". Otherwise std-a1 = sta1 inR for sta1 "independent of j" and

  std-ssj[n+1]sta = std-Pcond(std-ssj[n]*std-Pgj2, std-Pgj3) sta1
Also Pd'(t1) and 6.1.3-2 implies sta1 ∈ sts-a<lab§<1>,")">.

Proceeding like this we are left with the case
  std-ssj[n+1]sta = std-ssj[n]sta2
for sta2 ∈ sts-a<lab§<2>,")"> and sta2 "independent of j". The result follows by the induction hypothesis if sta2 ∈ sts-a<lab§<1>,"(">. It is not difficult to show (using Pd'(t1) and Pe'(t1) and 6.1.3-3) that
  sts-a<lab§<1>,"("> ⊇ sts-a<lab§<2>,")">

This finishes the proof for the case n+1. □

PROOF OF 6.1.4-3:
Let sta = {<λide."nil" inVal, inp, <>, <>> | inp∈inp}. Then
𝒫sts(t1)inp = sts-setup(𝒯sts(t1);sts-finish) inp
  = P𝒯sts(t1)sta ↓2
which was by 6.1.2-4 and definitions of sts-setup and sts-finish.

So the premise of the theorem merely amounts to P1(t1,t2,lab,sta).
By Theorem 6.1.4-2 then P1(t1,t2,<>,sta) which implies
  ∀sta ∈(P𝒯sts(t1)sta↓2)<<>,"(">:   P𝒯std(t1)sta = P𝒯std(t2)sta
which by Pa(t1) and 6.1.2-3 implies
  ∀sta∈sta: (𝒯std(t1);std-finish)sta = (𝒯std(t2);std-finish)sta
By definition of sta and std-setup this is equivalent to
  ∀inp∈inp: 𝒫std(t1)inp = 𝒫std(t2)inp □