ABSTRACT INTERPRETATION USING DOMAIN THEORY

Flemming Nielson

Doctor of Philosophy
University of Edinburgh
1984
ABSTRACT

A framework is developed for describing and proving the correctness of certain data flow analyses. This is done by ascribing several semantics to the programming language studied. The standard semantics is the usual semantics and an approximating semantics describes a data flow analysis. A value in the latter semantics describes a set of values in the former and this is expressed using the framework of abstract interpretation pioneered by P. and R. Cousot. Their view of a programming language is limited because a program is viewed as a (kind of) flowchart and the main aim of this work therefore is to extend the framework to all programming languages that have a denotational semantics. This is accomplished except for "storable procedures". A secondary aim is to extend abstract interpretation to include certain aspects of termination.

The first aim is addressed by studying a metalanguage for denotational semantics. A collecting semantics is defined and it may be viewed as the most precise of all data flow analyses. Its formulation requires a study of (relational) powerdomains. It is then proved correct and "as precise as possible" with respect to the standard semantics. The development of abstract interpretation generalises the previous approaches. In particular, the tensor product is found to generalise the relational method just as cartesian product corresponds to the independent attribute method. It is studied how to pass between such methods and how functionals like conditional are likely to look.

To achieve the second aim another powerdomain is required. The development of abstract interpretation distinguishes between two partial orders: $\mathcal{E}$ is the usual partial order of denotational semantics and $\preceq$ expresses the data flow analysis idea of "safe approximation". (It was $\mathcal{E}$ above.) An example is given that requires this framework. Another example studies a nondeterministic language and its usual (nondeterministic) denotational semantics. It is shown that the semantics may be viewed as a data flow analysis upon a deterministic language where oracles resolve the choice of paths. This view motivates the definition of a nondeterministic denotational semantics that distinguishes between "may diverge" and "may have to diverge".
PREFACE

My work upon combining abstract interpretation with denotational semantics started when I was a M.Sc. student and I learned much about data flow analysis from my supervisor N. Jones. As a Ph.D. student I learned much domain theory from my supervisor G. Plotkin and this cleared the way for a much more general combination. I would also like to thank A. Mycroft for sharing my interest in abstract interpretation.

This research was funded for two years and six months by The Danish Natural Science Research Council.

My wife already knows my gratitude to her.
DECLARATION

I hereby declare that this thesis has been composed by myself, that the work presented has not been presented for any university degree before and that the work is my own except as follows. Major techniques suggested by my supervisor are indicated by a reference to /PloPS/. Sections 5.3 to 5.5 contain material that builds on chapter 2 of /Myc81/; this material was previously published as /My81/ and in its present form is mostly my work. A preliminary version of sections 5.6 and 5.7 has been published as /Nie83/.

Flemming Nielsen
CONTENTS

Abstract 1
Preface 2
Declaration 3
Contents 4
Chapter 1: Introduction 7
  1.1 The general setting 8
  1.2 Abstract interpretation 11
  1.3 Denotational formulations of program properties 21
  1.4 Aims of the thesis and overview 23
Chapter 2: Standard Semantics 33
  2.1 Syntax of the metalanguage 33
    The types 33
    The expressions 38
  2.2 Preliminary domain theory 43
    Basic order theoretic definitions 44
    Categorical formulation 48
    Recursive domain equations 54
  2.3 Semantics of the metalanguage 69
    Type part 70
    Expression part 76
  2.4 Applications of the metalanguage 80
Chapter 3: Collecting Semantics 91
  3.1 Relational powerdomains 92
  3.2 Tensor products 101
    Definition and construction 101
    Other characterisations 113
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3 Collecting interpretation</td>
<td>127</td>
</tr>
<tr>
<td>Type part</td>
<td>128</td>
</tr>
<tr>
<td>Expression part</td>
<td>136</td>
</tr>
<tr>
<td>Chapter 4: Abstract Interpretation</td>
<td>163</td>
</tr>
<tr>
<td>4.1 Pairs of adjoined functions</td>
<td>164</td>
</tr>
<tr>
<td>4.2 Approximating interpretation</td>
<td>174</td>
</tr>
<tr>
<td>Relating the interpretations</td>
<td>176</td>
</tr>
<tr>
<td>Inducing an interpretation</td>
<td>180</td>
</tr>
<tr>
<td>4.3 Compositional definition of abs and con</td>
<td>185</td>
</tr>
<tr>
<td>The framework for change of method</td>
<td>186</td>
</tr>
<tr>
<td>Example changes of method</td>
<td>192</td>
</tr>
<tr>
<td>4.4 Expected definitions</td>
<td>198</td>
</tr>
<tr>
<td>4.5 Pragmatics of the tensor product</td>
<td>215</td>
</tr>
<tr>
<td>Chapter 5: Strong Abstract Interpretation</td>
<td>223</td>
</tr>
<tr>
<td>5.1 Non-continuous domain theory</td>
<td>226</td>
</tr>
<tr>
<td>5.2 Powerdomains</td>
<td>233</td>
</tr>
<tr>
<td>As a dcpo</td>
<td>234</td>
</tr>
<tr>
<td>As an augmented dcpo</td>
<td>240</td>
</tr>
<tr>
<td>5.3 Standard and collecting semantics</td>
<td>247</td>
</tr>
<tr>
<td>5.4 Abstract interpretation</td>
<td>253</td>
</tr>
<tr>
<td>5.5 Applications</td>
<td>262</td>
</tr>
<tr>
<td>5.6 Nondeterminism as abstract interpretation</td>
<td>270</td>
</tr>
<tr>
<td>The example language and its semantics</td>
<td>270</td>
</tr>
<tr>
<td>The partly collecting semantics</td>
<td>274</td>
</tr>
<tr>
<td>The partly induced semantics</td>
<td>278</td>
</tr>
<tr>
<td>5.7 Other notions of nondeterminism</td>
<td>287</td>
</tr>
<tr>
<td>On replacing ${0,1}^w$ by an &quot;equivalent&quot; subset</td>
<td>288</td>
</tr>
<tr>
<td>Angelic semantics</td>
<td>294</td>
</tr>
</tbody>
</table>
The idea behind abstract interpretation, the subject matter of this thesis, is best explained with an example taken from /CoCo77b/.

The text \(-1515\times17\) may be interpreted to denote computations on the abstract universe \(\{(+), (-), (+)\}\) where the semantics of arithmetic operators is defined by the rule of signs. The abstract computation

\[-1515\times17 \rightarrow -(+)\times (+) \rightarrow (-)\times (+) \rightarrow (-)\]

shows that \(-1515\times17\) is a negative number. Abstract interpretation is concerned with a particular underlying structure of the universe, in this example the sign. It gives a summary of some facets of the actual computations of a program. In general this summary is simple to obtain but inaccurate, e.g. \(-1515+17 \rightarrow -(+)\times (+) \rightarrow (-)+(+) \rightarrow (+)\).

The results of abstract computations should be correct with respect to the semantics of the program. Usually this has been considered with respect to an operational semantics for flowcharts or occasionally a predicate transformer semantics. The purpose of this thesis is to develop abstract interpretation in the framework of denotational semantics /Sto77/ with its underlying domain theory. Denotational semantics is used partly because it allows considering programs in their source representation and partly because denotational semantics has been used to give semantics for most language features.

In the remainder of this chapter we shall

- overview a general setting where abstract interpretation may be encountered,
informally review abstract interpretation,
overview some earlier denotational formulations of properties of computations,
describe the aims of the work reported and
give an overview of the contents of the thesis.

The main body of previous work that is built upon is abstract interpretation (e.g. /CoCo77b/ and /CoCo79/) and domain theory (e.g. /SmPl82/ and /PloLN/). It is helpful if the reader is acquainted with abstract interpretation but the overview in section 1.2 may be sufficient. The domain theory needed is reviewed in section 2.2 but this overview may be hard to follow if the reader is not already acquainted with "Scott's ideas". Some set theory (e.g. /Hal60/) is needed in chapter 5.

1.1 THE GENERAL SETTING

The motivation behind the subject matter of this thesis concerns the implementation of programming languages whether by compilation (e.g. ALGOL and PASCAL) or by interpretation (e.g. APL and SETL). Compilation may be viewed as consisting of lexical analysis, syntax analysis, generation of intermediate code, code optimization, generation of machine specific code and peephole optimization /AhUl78/.

The phase of interest for abstract interpretation is code optimization. Similar considerations apply for the interpretation of programs.

Code optimization may be viewed as consisting of many program transformations. The purpose of each is to improve the quality of the program with respect to some measure (e.g. running-time, program length, storage requirements) while preserving the meaning of the program. For an example consider the program
... \( y := 2 \) ... (no \( y \)'s) ... \( x := y + (1 + 1) \) ... (no \( x \)'s) ... \( x := 0 \) ...

and the replacement of \( x := y + (1 + 1) \) by \( x := y + 2 \). It is correct to do so because the two pieces of program have the same meaning. Many interesting transformations are not of this kind and rely on data flow analysis to provide information about the context in which the program transformation is to be performed so that the two pieces of program need not have identical semantics. This is illustrated by the following two examples taken from /Nie81b/.

The first transformation (constant folding /AhU78/) is to replace \( x := y + (1 + 1) \) by \( x := 4 \). These two pieces of program have different semantics but the transformation is semantically correct because \( y \) will always have the value 2 before the assignment statement in question. The data flow analysis constant propagation /AhU78/ computes before each statement a (safe approximation to a) set of variable and value pairs, so that a given pair is included iff the variable has always that value at that point. This analysis is classified as a forward analysis /Hec77/ because it propagates information from left to right.

Another transformation (useless code elimination /AhU78/) is to remove \( x := y + (1 + 1) \). The semantics of this piece of program does not equal that of the empty statement but the transformation is correct because the value of \( x \) is not subsequently used before \( x \) has been assigned a new value. The data flow analysis live variable analysis /AhU78/ computes after each statement (a safe approximation to) the set of variables that in some computation will be used before having been assigned a new value. The analysis is a backward analysis /Hec77/ because it propagates information from right to left.
The data flow information computed is a safe approximation to the desired set. The need for approximation arises because the problem of determining for a program whether an element is in the desired set at a given point, is in general not decidable. For live variable analysis the problem is recursively enumerable and for constant propagation is the complement of recursively enumerable. By safe is meant that the computed set should be smaller/larger than desired if this means fewer transformations seem allowed. For live variable analysis safe means larger and for constant propagation it means smaller.

In code optimization a program is usually represented as a flowchart whose nodes contain sequences of very simple assignment statements. Most data flow analyses have been developed using this scheme (see /Hec77/ for an overview). However, it is often possible to perform much optimization at the source level (perhaps extending it with some primitives). An advantage of this is that the transformed program may be displayed in a way understandable to the programmer. For an indication of the potential efficiency gains one may note /BaSa74/, where it is estimated that 80% of the run-time dimension, type and value checks performed in a naive APL interpreter may be eliminated using data flow information. It therefore seems worthwhile to develop data flow analysis at this level. This has been done (/Hec77/, /Ros77/, /Wil81/, ...) but like most data flow analyses the information obtained is not proved correct with respect to any formal semantics. Such a programme is possible using the ideas of abstract interpretation. Some data flow analyses that can be handled using abstract interpretation are constant propagation, type determination and information for array bound checking.
1.2 Abstract Interpretation

Data flow analysis specifies at each program point information about the context in which it appears. Often this information describes (a safe approximation to) the possible states at that point. Constant propagation is an example of this but live variable analysis is not. The important step of formally relating data flow information to sets of states was taken by Patrick and Radhia Cousot in their work on abstract interpretation /CoCo76, CoCo77a, CoCo77b, CoCo78, CoCo79, CouBl/. In data flow analysis much work has been directed towards efficient algorithms for performing the data flow analyses but this has played a small role in abstract interpretation and we shall ignore the issue in this thesis.

To explain abstract interpretation consider the following flowchart. Nodes are labelled by $n_1$ and $n_2$ is a test node whereas the others are assignment nodes. We shall assume that the variables $x$ and $y$ take their values in the set $Z = \{ \ldots, -1, 0, 1, \ldots \}$ of integers. The set of states may be taken to be $Z^2$. The semantics of an assignment node is given by a (total) function $f: Z^2 \to Z^2$. For the test node $n_2$ there is a predicate (i.e. a total function) $f_2: Z^2 \to \{ \text{tt}, \text{ff} \}$. Here tt means true and ff means false.

A first step is to consider the collecting semantics (called the static semantics in /CoCo77b/). The aim is to specify for each node $n_i$ the set $G_i \subseteq Z^2$ of states possible at entry to that node. It is assumed that $G_i$ is some given set $G_{\text{start}}$ (often $Z^2$). The semantics of an assignment node is given by a function $g: \mathcal{P}(Z^2) \to \mathcal{P}(Z^2)$ where $\mathcal{P}(Z^2)$ is the powerset of $Z^2$. It is defined
by

\[ g_1(Y) = \{f_1(u,v) \mid (u,v) \in Y \} \]

The test node \( n_2 \) gives rise to two such functions:

\[ g_{2,\text{yes}}(Y) = \{(u,v) \mid (u,v) \in Y \land f_2(u,v) = \text{tt}\} \]
\[ g_{2,\text{no}}(Y) = \{(u,v) \mid (u,v) \in Y \land f_2(u,v) = \text{ff}\} \]

Intuitively the \( G_i \) must satisfy the following equations:

\[ G_1 = \text{start} \]
\[ G_2 = g_1(G_1) \cup g_5(G_5) \]  \hspace{1cm} \text{(G-equations)}
\[ G_3 = g_{2,\text{no}}(G_2) \]
\[ G_4 = g_{2,\text{yes}}(G_2) \]
\[ G_5 = g_4(G_4) \]

Another way to view these equations is that they define a function \( g \) from \( (\wp(\mathbb{Z}^2))^5 \) to itself, i.e.
We shall see that this system of equations always has a least fixed point solution, i.e. a solution that is included in every other solution. This fixed point solution (written \(G_i\)) is of interest because \(G_i\) contains precisely those states that may arise at entry to \(n_i\) for some computation starting at \(n_i\) with a state in \(G_{\text{start}}\). For the example flowchart \(G_{\text{start}} = \{(9, 9)\}\) gives \(G_4 = \{(9, 0), (10, 9)\}\) and hence \(x\) may be 9 or 10 at entry to \(n_4\).

A possible data flow analysis is the detection of signs. Let \(I = \{-, 0, +\}\) be a "degraded" version of the integers. The idea is to replace \(\mathcal{P}(\mathbb{Z}^2)\) by \(\mathcal{P}(I^2)\) and so represent a set \(Y = \{(-1, 1), (2, -2), (3, -3)\}\) by \(X = \{(-, +), (+, -)\}\). The meaning of \(X \subseteq I^2\) is formally defined as \(\text{con}(I) \subseteq \mathbb{Z}^2\) where \(\text{con}: \mathcal{P}(I^2) \rightarrow \mathcal{P}(\mathbb{Z}^2)\) is a concretization function. If the function \(r: \mathbb{Z} \rightarrow I\) is defined by \(r(-1) = -, r(0) = 0, r(1) = +\) etc., we may define

\[
\text{con}(X) = \{(u, v) \mid (r(u), r(v)) \in X\}
\]

Similarly there is an abstraction function \(\text{abs}: \mathcal{P}(\mathbb{Z}^2) \rightarrow \mathcal{P}(I^2)\) defined by

\[
\text{abs}(Y) = \{(r(u), r(v)) \mid (u, v) \in Y\}
\]

It satisfies that \(\text{abs}(Y)\) is the smallest set \(X\) such that \(\text{con}(X)\) includes \(Y\). This relationship between \(\text{abs}\) and \(\text{con}\) will be formalised by the definition of pairs of adjoined functions below and is central to the theory of abstract interpretation.

One may then formulate a new set of equations:

\[
\begin{align*}
G'_1 &= G'_{\text{start}} \\
G'_2 &= g'_1(G'_1) \cup g'_5(G'_1)
\end{align*}
\]
\[ G'_3 = g'_{2, \text{no}}(G'_2) \]
\[ G'_4 = g'_{2, \text{yes}}(G'_2) \]
\[ G'_5 = g'_{4}(G'_4) \]  

(G'-equations)

It is natural to define \( g'_i = \text{abs} \cdot g_i \cdot \text{con} \) as this intuitively is the most precise account of \( g_i \) (and \( f_i \)) that can be given when considering subsets of \( I^2 \). We shall define \( g'_{2, \text{yes}} \) and \( g'_{2, \text{no}} \) similarly although these are frequently assumed to be identity functions, i.e. \( g'_{2, \text{yes}}(X) = X = g'_{2, \text{no}}(X) \). For \( G'_1 \) it is natural to use \( \text{abs}(G'_{\text{start}}) \). We shall see that a least fixed point solution \( G'_1 \) exists and that \( \text{con}(G'_1) \supseteq G'_1 \). Because of the relation between \( G'_1 \) and the set of states possible before \( n_i \) we see that \( G'_1 \) describes a superset of this set. Returning to the example we will get

\[ G'_{\text{start}} = \{(+,+)\} \] and \( G'_4 = \{(+,0), (+,+)\} \).

Not all data flow analyses use powersets. In /Kil73/ and /KaUl77/ a theory of monotone frameworks was developed where powersets are replaced by complete lattices of finite height (see below). Most data flow analyses fall within this framework and efficient solution methods have been developed (e.g. /GrWe76/). Abstract interpretation may be viewed as extending the theory with means for showing the specified solutions to be semantically correct (using concretization functions). To explain this and the theory for when least fixed points exist we need some lattice theoretic notions. More details can be found in /Cou81/.

For the overview of lattice theoretic concepts let \( L = (L, \leq) \) be a partially ordered set. An upper bound of a subset \( H \) of \( L \) is an element \( l \) of \( L \) such that \( h \leq l \) whenever \( h \) is an element of \( H \). The
subset $H$ has a (necessarily unique) least upper bound denoted $\bigcup H$ iff $\bigcup H$ is an upper bound such that $\bigcup H \leq l$ for every upper bound $l$. Dually (i.e. using $\cap$) there may be a greatest lower bound $\bigcap H$. If all least upper bounds exist we say $L$ is a complete lattice and then all greatest lower bounds exist too because

$$\bigcap H = \bigcup \{ l \mid \forall h \in H : l \leq h \}$$

It is convenient to write $L = \{ \emptyset, \tau = \bigcap \emptyset, x \wedge y = \bigcup \{ x, y \} \}$ and $x \vee y = \bigcap \{ x, y \}$. An (increasing) chain is a sequence $(l_n)_{n}$ of elements of $L$ that are indexed by the natural numbers $N = \{ 1, 2, \ldots \}$ in such a way that $l_1 \leq l_2 \leq \ldots$. If no such chain has infinitely many distinct elements then $L$ is said to be of finite height. The least upper bound $\bigcup \{ l_n \mid n \in N \}$ of the chain $(l_n)_{n}$ is written $\bigcup_{n} l_n$. By $L^n = L \times \cdots \times L$ is meant the n-fold cartesian product of $L$ with itself. The partial order is defined componentwise, i.e. $(l_1, \ldots, l_n) \leq (l'_1, \ldots, l'_n)$ iff $l_1 \leq l'_1$ and $\ldots$ and $l_n \leq l'_n$. It follows that $L^n$ (for $n \geq 1$) is a complete lattice if $L$ is.

A function $g : L \to M$ between partially ordered sets is monotonic iff $l \leq l'$ implies $g(l) \leq g(l')$. It is strict iff $g(\bot)$ is the least element of $M$ whenever $\bot$ is the least element of $L$. It is completely additive iff

$$g(\bigcup H) = \bigcup \{ g(h) \mid h \in H \}$$

whenever $H$ is a subset of $L$ that has a least upper bound. It is additive iff $g(l \cup l') = g(l) \cup g(l')$ whenever $l \cup l'$ exists. The partial order $\leq$ on $M$ may be extended pointwise to functions by defining $g \leq g'$ iff $g(l) \leq g'(l)$ holds for all elements $l$ of $L$. A fixed point of $g : L \to L$ is an element $l$ of $L$ such that $g(l) = l$. It is the least
fixed point if \( l' \in L \) holds for all fixed points \( l' \). If \( L \) is a complete lattice and \( g \) is monotonic there always is a least fixed point and it is given by

\[
\text{LFP}(g) = \bigcap \{ l \mid g(l) \leq l \}
\]

This result is due to /Tar55/ and may be proved as follows. If \( g(l) \leq l \) the monotonicity of \( g \) gives \( g(\text{LFP}(g)) \leq g(l) \leq l \) and therefore \( g(\text{LFP}(g)) \leq \text{LFP}(g) \). Further, monotonicity gives \( g(g(\text{LFP}(g))) \leq g(\text{LFP}(g)) \) and hence \( \text{LFP}(g) \leq g(\text{LFP}(g)) \) follows from the definition of \( \text{LFP}(g) \).

Turning to abstract interpretation we now address the details that were omitted before. The formulation of the collecting semantics used the complete lattice \( (\wp(\mathbb{Z}^2), \subseteq) \) but this will not be essential below so we shall just assume a complete lattice \( L = (L, \subseteq) \). It was already said that the equations define a function \( g \) from \( L^n \) to \( L^n \) (with \( n=5 \)). It is not hard to see that \( g \) is monotononic and hence there is a least fixed point as stated. We shall write \((G_1, \ldots, G_n)\) for this least fixed point. In the detection of signs analysis there is a monotonic function \( g' \) from \( L'^n \) to \( L'^n \). In the example \( L' \) is \( (\wp(\mathbb{Z}^2), \subseteq) \) but we shall only need that \( L' \) is a complete lattice. Again \( g' \) has a least fixed point which is written \((G'_1, \ldots, G'_n)\).

It remains to relate the \( G'_i \) to the \( G_i \). We already saw that this is to be done using a concretization function \( \text{con} : L' \rightarrow L \) and an abstraction function \( \text{abs} : L \rightarrow L' \). It is apparent from the two examples given that the partial order represents the data flow analysis idea of "can safely be approximated by". It is therefore natural to assume that \( \text{abs} \) and \( \text{con} \) are monotonic for this just means that they preserve "safe approximation". We shall also assume that
con(abs(l)) ≥ l is always the case. This condition may be illustrated by

\[ L \xrightarrow{\text{abs}} l \xleftarrow{\text{con}} L' \]

and it merely says that abs(l) is a safe description of l. It is common also to assume that abs(con(l')) ≥ l' holds for all l'. This implies that abs(l) is the least element that safely describes l, i.e. that

\[ \text{abs}(l) = \cap\{l' \mid \text{con}(l') \geq l\} \]

because whenever con(l') ≥ l we get l' ≥ abs(con(l')) ≥ abs(l). These conditions upon abs and con may be reformulated as

\[ \forall l: \forall l': (\text{abs}(l) \geq l' \iff l \subseteq \text{con}(l')) \]

and (abs, con) is said to be a pair of adjoined functions whenever this is the case /CoCo79/. It is easy to see that this condition holds for the particular abs and con displayed earlier. The adjoinedness condition is further motivated and studied in /CoCo79/ and /Nie81a/. One property worth noting is that abs is completely additive (as is straightforward to show).

The desired relation between \( G'_1 \) and \( G'_1 \) is that \( G'_1 \subseteq \text{con}(G'_1) \), i.e. that \( G'_1 \) is a safe description of \( G'_1 \). Writing \( \text{con}^n(l'_1, \ldots, l'_n) = (\text{con}(l'_1), \ldots, \text{con}(l'_n)) \) this may be written \( \text{LFP}(g) \subseteq \text{con}^n(\text{LFP}(g')) \).

To achieve this it suffices to show that \( g' \subseteq \text{con}^n \circ \text{con}^n \circ g' \) for then one has
\[ g(\text{con}^n(LFP(g'))) \in \text{con}^n(g'(LFP(g'))) = \text{con}^n(LFP(g')) \]

and by /Tar55/ the desired result follows. When (abs, con) is a pair of adjoined functions also (abs^n, con^n) is and \( g' \text{con}^n \text{con}^n g' \) is then equivalent to \( \text{abs}^n g' \text{con}^n \text{con}^n g' \). Returning to the examples considered earlier it is straightforward to use the complete additivity of abs to show that equality holds and hence \( G'_i \) is a safe description of \( G_i \). The function \( \text{abs}^n g' \text{con}^n \) is said to be induced from \( g \) /CoCo79/ and is of interest because it intuitively is the best representation of \( g \) that can be obtained using the approximate space \( L' \).

Abstract interpretation is a quite general framework and we shall consider a few additional examples. The detection of signs analysis (as well as the collecting semantics) is a relational method /Jon81a/ because the use of \( L' = \emptyset(I^2) \) means that all combinations of values are considered. An alternative might be to use \( L'' = (\emptyset(I))^2 \) which gives an independent attribute method where combinations of values cannot be explicitly considered. The relation between \( L' \) and \( L'' \) may be defined by the following concretization and abstraction functions

\[
\text{con}'(Y_1, Y_2) = Y_1 \times Y_2
\]

\[
\text{abs}'(Y) = (\{u \exists v: (u,v) \in Y\}, \{v \exists u: (u,v) \notin Y\})
\]

As before a system of equations may be defined where \( G'' \text{start} = \text{abs}'(G'_\text{start}) \) and \( g'' = \text{abs}'(g'_i) \text{con}' \). This gives a monotonic function \( g'' \) from \( (L'')^n \) to \( (L'')^n \) and it is induced from \( g' \). It follows that the least fixed point satisfies \( G'_i \in \text{con}'(G''_i) \) and hence

\[
G'_i \in (\text{con}' \text{con}') (G''_i)
\]

so that \( G''_i \) is a safe description of \( G_i \).
One can move closer to the example given in the introduction by replacing $\mathcal{P}(I)$ with

Here the elements are $\emptyset, -, 0, +, \text{all}$ and the partial order has $x \leq y$ iff there is a non-descending path from $x$ to $y$. One way to say that the above space is more approximate than $\mathcal{P}(I)$ is to note that more subsets have the same least upper bounds. So the least upper bound operation may be viewed as expressing the "degree of approximation" just as the partial order expresses "can safely be approximated by".

Another example where many sets have the same least upper bounds are in constant propagation where $\mathcal{P}(Z)$ is replaced by

with the concretization function being the obvious one.

So far it has been assumed that the least fixed point is the desired solution. It is usually called the MFP solution (for maximal fixed point) because originally $\Xi$ was used where we have used $\mathcal{E}$. Another solution is the MOP solution (for meet over paths). If $p$ is a path from the entrance of node $n_1$ to the entrance of $n_j$ the composite effect of the $\bar{g}_k$'s (expressing the effect of the nodes) is written $\bar{g}_p$, e.g.

$$\bar{g}_{n_1n_2n_4n_5} = \bar{g}_4 \cdot \bar{g}_2 \cdot \text{yes} \cdot \bar{g}_1$$
The MOP solution specifies that

\[ \bar{M}_1 = \bigcup \left\{ \bar{g}_P(M_{\text{start}}) \mid p \text{ is a possibly cyclic path from } n_1 \text{ to } n_i \right\} \]

is the desired information at entrance to \( n_i \). In the following we shall make use of the fact that there is no nonempty path that ends in \( n_1 \). If \( \bar{M}_{\text{start}} = \bar{G}_{\text{start}} \) one can prove that \( \bar{M}_1 = \bar{G}_1 \) by induction on the maximal length of paths considered /KaUl77, CoCo77b/. If each \( \bar{g}_k \) is completely additive one can show that \( (\bar{M}_1, \ldots, \bar{M}_n) \) is a fixed point of the corresponding system of equations and hence \( \bar{M}_1 = \bar{G}_1 \). This is the case for the collecting semantics (where \( \bar{g}_k = g_k \)). If not all \( \bar{g}_k \) are completely additive the solutions may be different and this is the case for constant propagation (for a suitable flowchart /KaUl77/).

It is usually agreed that the MOP solution is the ideally desired solution rather than the MFP solution. Also it is quite easy to show that if \( \bar{M}_{\text{start}} \in \text{con}(\bar{M}_{\text{start}}) \) and \( \bar{g}_k \in \text{con} \bar{g}_k \) then the respective MOP solutions satisfy \( \bar{M}_1 \in \text{con}(\bar{M}_1) \). Despite this the MOP solution is not usually computed. This is because there are complete lattices of finite height (e.g. as in constant propagation) such that computation of the MOP solution corresponds to solving the halting problem /KaUl77/. By contrast the MFP solution is decidable if the complete lattice has finite height. The idea is that finite height guarantees that \( g^n(\omega) = g(g^n(\omega)) \) for some \( n \) and \( g^n(\omega) \) then equals LFP(\( g \)). We omit the details required for a formal statement of this.
1.3 Denotational Formulations of Program Properties

Several papers give denotational formulations of properties of program computations. Some papers that have influenced the present work are surveyed below. A further reference is /Ple81/ that considers storage sharing in SCHEME (a dialect of LISP).

In /Don78/ an imperative language with while-loops is given a continuation-style standard semantics using locations. Then a non-standard semantics defining properties of computations is introduced. Where the standard semantics uses a flat domain of integers ($\mathbb{Z}_1$ with elements $\mathbb{Z}_1 \cup \mathbb{Z}$) and $x \leq y$ if $x =_1 o r x = y$) the non-standard semantics uses a complete lattice $L$ having as elements certain nonempty subsets of $\mathbb{Z}_1$. It is assumed that all elements of $L$ are left-closed (i.e. if $x \leq y$ and $y$ is a member so is $x$) and a further condition that we shall ignore. All sets $\{\bot, x\}$ for $x \in \mathbb{Z}_1$ are elements of $L$ and the partial order is set-containment $\subseteq$.

Where the standard semantics uses a domain expression $\mathbb{Z}_1^x \ldots$ the non-standard one uses $L^x \ldots$ and a function $f : \mathbb{Z}_1^x \ldots \rightarrow \mathbb{Z}_1^x \ldots$ becomes $g : L^x \ldots \rightarrow L^x \ldots$. In other words an "independent attribute method" is used. This is complemented by a formal proof that $g(S) \subseteq \{f(x) | x \in S\}$ but equality is not considered; we shall see in section 3.3 that it will not hold in general. The idea in how to define $L$ builds on /WaSh77/ (see section 4.1) which shares some ideas with abstract interpretation but is not as "developed".

A possible critique of such an approach is that one may only deduce the properties holding at the end of the program and not at other program points. The latter is achieved in /Don81/ for a similar language. A standard semantics is modified to produce a
list of the intermediate values arising at various program points. Then non-standard semantics are defined that map program points to properties holding there. For the four data flow analyses considered in the paper it is proved that the non-standard semantics includes descriptions of all the intermediate values produced. The notation used is that of the Semantics Implementation System (SIS /Mos79/). This may be the reason for why the formulation is further removed than /Don78/ from the monotone frameworks or abstract interpretation approaches. The SIS system will, however, loop upon iterative programs (as is explained in chapter 6) so the analyses cannot be automatically executed /Ple81/. The solution specified is not related to the MOP and MFP solutions of the traditional approach.

In /Nie82/ the framework of abstract interpretation is given a denotational formulation for a language similar to the previous. First the semantics is given using a store semantics (see /MiSt76/). It is then modified so as to map program points to the set of states possible there. Thirdly, a "static semantics" is obtained by modifying the previous semantics to accept sets of states as input. It is proved that with input $S$ the states specified at some point are the union over the elements of $S$ of those the previous semantics specified there. The fourth semantics replaces sets of states by complete lattices of description elements. These complete lattices are related to sets of states using pairs of adjoined functions (in fact a weaker condition) and the latter semantics is proved correct with respect to the "static semantics" (i.e. it specifies a safe approximation). All these semantics are in continuation style and it is shown that the abstract information specified is the MOP solution. It is also possible to obtain the MFP solution.
A companion paper /Nie81b/ shows that the data flow information can be used to assert the correctness of the program transformation method of constant folding (and to some extent also useless code elimination). In these papers the denotational formulation of abstract interpretation is mainly viewed as a convenient step in proving the correctness of program transformations depending on data flow information. It is not discussed whether it would be feasible to execute the denotational formulations directly (e.g. on a modified SIS system).

An application of abstract interpretation to the optimization of applicative programs is contained in /Myc81/. Further an attempt is made at extending abstract interpretation to include certain aspects of termination. The attempt is of rather limited applicability, however, and the main application is incorrect. The motivation for sections 5.3 to 5.5 of this thesis therefore was to present a correct and slightly more general development.

1.4 AIMS OF THE THESIS AND OVERVIEW

The denotational approaches to formulating abstract interpretation (or just data flow analyses) have only considered toy languages. This gives rise to two shortcomings:

a) it is not completely specified how an ordinary denotational semantics is changed to one that performs abstract interpretation,

b) it is unclear what language features in some realistic language might give rise to difficulties that have not been solved for some of the previous toy languages.
Further,

c) formulations generally use an "independent attribute method" even in cases where a "relational method" poses no technical problems.

The operational formulations of abstract interpretation also has shortcoming b). Shortcoming a) occurs in a slightly weaker form because some generality is obtained by considering arbitrary flow-charts but it is not clear that the flowchart view encompasses all language features. Shortcoming c) does not surface in the flowchart view but is still implicit.

A further shortcoming of these approaches (except in a crude way parts of /CoCo79/ and /CoCo78/) is that

d) no notion of termination is considered

This is a problem mostly ignored in data flow analysis but when considering ambitious program transformations (like in-line expansion of procedures) this seems to be needed.

The development in chapters 2, 3 and 4 is concerned with a), b) and c) above. First, the use of denotational semantics gives the potential for remedying a) because denotational semantics has been used to give the semantics of most realistic languages. To fully achieve a) a metalanguage is defined (in section 2.1) and the development works for any language whose semantics can be defined in that metalanguage. The types are classified into two levels and this gives information about which types in the collecting semantics should be powersets (in general not all types should). There are some syntactical limitations in the metalanguage that requires further
research to overcome. This means that b) is achieved because the language features that cannot be treated (e.g. procedures as parameters) are those whose semantics uses features not included in the metalanguage. Whether c) is achieved or not depends on generalizing the notion of "relational method". If this is taken to be "as in the collecting semantics" then some progress is made. In particular, the study of the tensor product (in section 3.2) seems to generalize the relational method of /Jon81a/ from powersets to a large class of complete lattices (as is discussed in section 4.5).

Some simplifying decisions have been made in order to allow the development to concentrate upon a), b) and c). An important one is not to consider program points. This means that the development in a sense is closer to /Don78/ than to /Nie82/. It is hoped that program points can be introduced much as in /Nie82/. Once this is done program transformations and implementation of the analyses may be considered. Another important decision is only to treat "domain-independent" issues of abstract interpretation. An example "domain-independent" issues is expected definitions of functionals like conditional. But examples of how to choose an approximation lattice for some particular purpose (e.g. type checking) will not be given as these can be found elsewhere.

Chapter 5 is concerned with d), the problem of expressing termination. The mathematics is more complicated so it is only done at the level of toy languages. The motivation for this work is to bring more data flow analyses within the world of abstract interpretation. One application is to validate a data flow analysis that has been found useful for detecting when call-by-name can be replaced
by call-by-value /Myc81/. Another, more philosophical, application is to show that nondeterminism is nothing but a particular data flow analysis. A more detailed overview of each chapter is given below.

**Overview of chapter 2**

The semantic metalanguage is defined in section 2.1. It consists of syntax for types (domains) and expressions (elements). The types are divided into two nearly identical levels. The level of a type is used to determine (in the collecting semantics) whether the domain or its powerdomain (roughly powerset) should be used. The levels might be viewed as separating what is sometimes called static and dynamic semantics. The metalanguage has one major limitation namely that functions are not "first class citizens".

The basic domain theory is reviewed in section 2.2. In particular domain constructors (like cartesian product) are defined and it is shown how to solve (possibly nested) recursive domain equations. Categorical concepts (mainly category, functor and limiting cone) are found useful for stating definitions and results. (These concepts are all explained in section 2.2.) Much of the material is quite standard so many facts are stated with no or little proof. The proofs given generally use order theoretic methods (as in /Ploln/ and /SmP182/) rather than categorical methods (as in /LeSm81/). In particular, nested recursive domain equations is handled by considering certain functors over a category whose morphisms are pairs of functions.

The metalanguage will be interpreted in many different ways (one for each kind of semantics considered). It is therefore helpful to
define the notion of an interpretation to specify those aspects that are not fixed throughout the thesis. This is done in section 2.3 and the semantics of the metalanguage is then given relative to an interpretation. The standard semantics /MiSt76/ (i.e. the ordinary denotational semantics) is then defined by specifying the standard interpretation $S$.

The metalanguage has a "flavour" that will be useful later but differs from e.g. /Sto77/ (e.g. the functionality of conditional). So to show the "expressiveness" of the metalanguage section 2.4 gives examples of its use. The major example is to define the semantics of an imperative language with procedures (like SMALL in /Gor79/). The other example is a simple applicative language (recursion equation schemes).

**Overview of chapter 3**

The first step towards formulating abstract interpretation is to define the analogue of the collecting semantics of /CoCo77b/. (It was called the "static semantics" in /CoCo77b/ but this conflicts with the terminology of static versus dynamic semantics. Unfortunately the terminology "collecting semantics" was used in /Nie82/ to mean something slightly different.) Previously the powerset $\mathcal{P}(\ldots)$ has been used but for the denotational formulation some power-domain /Plo76, Smy78/ seems to be needed. Three candidates are considered in section 3.1 and the relational (or lower) powerdomain $\mathcal{P}_R(\ldots)$ is chosen because $\mathcal{P}_R(\mathbb{Z}_L)$ essentially is $\mathcal{P}(\mathbb{Z})$. The section then covers the necessary theory about this powerdomain.
Section 3.2 defines and studies the tensor product $L \otimes M$ of certain complete lattices $L$ and $M$. The tensor product is of interest because $\mathcal{P}_R(D \times E)$ essentially is $\mathcal{P}_R(D) \otimes \mathcal{P}_R(E)$. The practical use of it is further discussed in section 4.5 and it is claimed that the tensor product allows for a generalization (from powersets to complete lattices) of the relational method of /Jon81a/; the independent attribute method then amounts to use of cartesian product. The treatment of the tensor product in section 3.2 includes giving different characterisations of it (one as a function space) and a picture of a tensor product of non-powerdomains.

The collecting interpretation is defined in section 3.3. The idea is that a function $f: Z \times Z \to Z$ becomes $g: \mathcal{P}_R(Z) \otimes \mathcal{P}_R(Z) \to \mathcal{P}_R(Z)$. The major theorem is to show that this is done in such a way that $g$ is the "pointwise" application of $f$. Intuitively this means that $g(Y) = \{f(y) \mid y \in Y\}$. This result requires the use of several techniques, most notably the use of predicates defined on domains given as solutions to recursive domain equations. Luckily some of the results are proved in sufficient generality that they can be reused in chapter 4.

Overview of chapter 4

The development of abstract interpretation builds on the notion of a pair of adjoined (abstraction and concretization) functions. To stay within the usual (continuous) domain theory some additional assumptions are needed and these are formulated in section 4.1. Further, two additional properties upon the abstraction functions are studied. One amounts to when it could have been defined by a
"representation function". The other is that so-called "essential" elements are preserved. This condition is found of use in section 4.4. It is argued that this property formalises a vague intuition about when one approximation space is more natural than another.

The main development of abstract interpretation is performed in section 4.2. The idea is that a data flow analysis is specified as an interpretation with certain properties; such interpretations are called approximating interpretations. The collecting interpretation should be thought of as an approximating interpretation that is approximated by all others. First, the notion of when one approximating interpretation is safely approximated by another is defined using a family of adjoined abstraction and concretization functions (indexed by the types in one of the levels). It is shown that this condition is sufficient and necessary for a similar condition \( \leq_{\text{con},t} \) to hold between the semantics of expressions given by the two approximating interpretations. When \( g \) and \( h \) correspond to nodes in the flowchart the condition \( g \leq_{\text{con},t} h \) is nothing but the condition

\[
g \cdot \text{con} \leq_{\text{con}} h \leq_{\text{con}} han

met earlier. Secondly, it is possible to specify an induced interpretation (generalizing the induced predicate transformers of [CoCo79]). This requires a given approximating interpretation, a family of more approximate spaces and a family of abstraction and concretization functions. It gives an approximating interpretation that uses the approximate spaces and is a safe approximation to the given interpretation. In a certain sense it is as precise as possible.
Section 4.3 considers how to define the abstraction and concretization functions by structural induction upon the indexing types. This was not of importance above but is in keeping with the principle of compositionality of denotational semantics. This is complicated by the possibility that the two data flow analyses use different "methods", e.g. one uses a relational method where the other uses an independent attribute method. The solution adopted is to specify the "change of method" using the categorical notion of a natural transformation. Several examples of such transformations are studied.

For some purposes an induced interpretation may be viewed as being too precise. Composition is an example of this. One may expect the definition of "composition" to be functional composition. Even if this is the case in some given approximating interpretation it needs not hold in the induced interpretation. It is, however, safe to use functional composition instead. Many similar examples can be given and these are studied in section 4.4. For some of these it will be required that the abstraction functions preserve "essential" elements. Finally, section 4.5 goes further into the discussion of why use of the tensor product gives a relational method.

Overview of chapter 5

In the beginning of chapter 5 it is argued that for some purposes a development based on the relational powerdomain is unsuitable. The problem is that it is impossible to express (in the collecting semantics) that some expression will terminate. To be able to do so
it is necessary to use another powerdomain and its theory is covered in section 5.2. It is sufficient for the purposes of this chapter but is not as general as might be desired (often the powerdomain is not a "domain"). In this development it is no longer valid to assume continuity and a "domain theory" based on monotonicity is covered in section 5.1 (but not for recursive domain equations).

Sections 5.3 to 5.5 consider a simple applicative language. First its standard and collecting semantics are defined and related. Because of the richness of the powerdomain the collecting semantics is a more faithful representation of the standard semantics. Secondly, it is argued that the powerdomain and all approximation spaces should be equipped with two partial orders. One represents "safe approximation" and the other corresponds to the usual partial order that is used in domain theory to define least fixed points. Chapters 2 to 4 may be viewed as a study of the special case where they are equal! The intuitive difference between these partial orders was already stated in /Nie81a/ and /Nie82/ and an attempt at formalising it was made in /Myc81/ (but using the wrong powerdomain). For such spaces a framework of abstract interpretation is developed. Thirdly, the applications show the correctness of a data flow analysis that was used in /Myc81/ to detect when call-by-name can be replaced by call-by-value.

Sections 5.6 and 5.7 study a simple nondeterministic language. The major aim is to show that the nondeterministic semantics of a program $c$ is nothing but a certain abstract interpretation of a deterministic program $c'$. The idea is to replace every occurrence

$c_1$ or $c_2$
of a nondeterministic choice in c by

\[
\textbf{if} \; \text{hd}(o) \; \textbf{then} \; o := \text{tl}(o) ; c_1 \; \textbf{else} \; o := \text{tl}(o) ; c_2
\]

in order to get c'. Here o is an oracle /Mil75/. It is used to resolve which choice should be made and is modified so it can be used to resolve the next choice. The idea is that the semantics of c equals an abstract interpretation of c' where all oracles are assumed possible. To perform this development the powerdomain of section 5.2 seems needed and abstract interpretation is extended somewhat. It is then a natural data flow analysis question to investigate what happens if not all oracles are possible. Under some reasonable assumptions the amount of potential difference is shown to be bounded by the nondeterministic semantics and a modified semantics that distinguishes between "may diverge" and "may have to diverge".
This chapter develops the framework of denotational semantics in which the formulation of abstract interpretation will be performed in chapter 4. The syntactic aspects of the semantic metalanguage are given in section 2.1 and the two-level type structure is explained. Applications of the metalanguage for defining semantics for example languages are given in section 2.4. The semantic aspects of the metalanguage are treated in section 2.3 and central to this is the definition of an interpretation. This is exemplified by defining the standard interpretation. The domain theory needed for this is developed/reviewed in section 2.2.

2.1 SYNTAX OF THE METALANGUAGE

The metalanguage has notation for defining types, which are to denote domains, and for defining expressions, which are to denote elements of domains. We begin with a study of the types and their two-level nature. Later the expressions are considered.

THE TYPES

The two levels of types are called the top-level and the bottom-level. We use the metavariable t for the former and gt for the latter. The syntax for the two levels is rather similar so underlining is used to disambiguate:
The top-level types should be rather familiar. The $A_i$ are unspecified basic types but we shall assume that the truth values $T$ are included. Types are combined using $k$-ary type constructors: $\times$ is cartesian product, $\times$ is smash product and $+$ is coalesced sum. These are defined in section 2.2 and we shall assume that $k > 2$. To be precise we should be explicit about the value of $k$, e.g. have $x^2$, $x^3$ etc., but this would make the notation heavier and detract from readability. A new least element may be added by lifting, as in $t_\bot$, and function spaces may be defined, as in $t_1 \to t_2$. Recursive types are available via $recX.t$ and the possibility of using $X_i$ in $t$. An example is $recX.N+(N\times X)$ that is the type of nonempty lists of integers (assuming $N$ is the type of integers). This notation allows for nested use of recursive types. We shall postpone the discussion of $ft$ (and $gt$).

For a type expression $t$ and a finite set $V$ of variables $X_i$, the static well-formedness condition $V \vdash t$ is defined by structural induction on $t$ as indicated below. Among other things it says that the free variables of $t$ are included in $V$.

<table>
<thead>
<tr>
<th>$t$</th>
<th>$A_i$</th>
<th>$t_1 \ldots t_k$</th>
<th>$\ldots recX.t$</th>
<th>$X$</th>
<th>$gt_1 \rightarrow gt_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\forall t \vdash t$</td>
<td>$\bigwedge_{i=1}^k \forall t_i$</td>
<td>$\ldots \forall [X] \vdash t$</td>
<td>$\forall \in V$</td>
<td>$\bigwedge_{i=1}^k \varnothing \vdash gt_i$</td>
<td></td>
</tr>
</tbody>
</table>

The use of the empty set $\varnothing$ in the rule for $V \vdash gt_1 \rightarrow gt_2$ will be explained later. Type expressions $t$ such that $\varnothing \vdash t$ will be of
special interest and are said to be closed.

It seems clear that the top-level types suffice for most imperative languages so it needs to be explained why a bottom-level is introduced. Consider command continuations /Sto77/ (of type C) for a language with states (of type $S$). In a continuation style semantics the meaning of a command is a continuation transformer (of type $G$). The definitions that are to be used are $G = C \rightarrow C$ in the top-level and $C = S \rightarrow S$ in the bottom-level. For a standard semantics /MiSt76/ this distinction is superfluous so we shall consider a non-standard semantics. In the collecting semantics it should be clear that elements of $C$ are to be functions from $\mathcal{P}(S)$ to $\mathcal{P}(S)$. Further, elements of $G$ are still to be functions from $C$ to $C$, not $\mathcal{P}(C)$ to $\mathcal{P}(C)$, as abstract interpretation is only concerned with the "state transformation" aspects and does not change a continuation style semantics to something else /Nie82/. So the use of the two levels is a welcome guide in the development of chapters 3 and 4.

Remark. One may ponder whether the use of two levels might be useful for other purposes than abstract interpretation. An example that seems to show this is the case is code generation /MiSt76/. Here $C$ is to be thought of as the domain of code (for state transformations) while $G$ is still a domain of functions from code to code. In effect $\rightarrow$ is to be interpreted as a function space construction whereas $\rightarrow$ is to be interpreted as textual representation for functions. Perhaps one may view the top-level as the place to express what is sometimes called static semantics and the bottom-level as the place to express dynamic semantics. In
particular it will emerge (later in this section and in section 2.4) that mutually recursive function definitions will be "solved" in the top-level.

The bottom-level types are much like the top-level types. In particular the unspecified base types $\mathbb A_i$ are assumed to include the truth-values $T$. Domain constructors will be interpreted differently in various semantics and are underlined so they can be distinguished from those in the top-level. As an example $+$ will be $+$ in the standard semantics but in the collecting semantics it will be $\times$ because $\mathcal P(\mathbb A + \mathbb B)$ "is" $\mathcal P(\mathbb A) \times \mathcal P(\mathbb B)$ and not $\mathcal P(\mathbb A) + \mathcal P(\mathbb B)$. It might be of practical use in some semantics to have several versions of the domain constructors (e.g. $+$, $\cdot$, ...) so that they could be interpreted differently later and this should not cause any problems. Finally, the static well-formedness condition $V \vdash gt$ is defined much as before.

The main difference between the two levels is that the bottom-level has no function-spaces, i.e. we do not have $gt ::= ft$. Amending this would cause problems in chapter 4 that are not immediate to solve (see chapter 6). It is a consequence that e.g. procedures as parameters and "storable procedures" cannot be treated. This limits the generality of working at the metalanguage level but it is desirable that limitations in the theory are brought out at the syntactic level!

The interplay between the two levels is somewhat restricted. Using $t ::= ft$ the bottom-level types may be included among the top-level types. This is useful e.g. for the function get defined in section 2.4. The functionality of get is $L \rightarrow S \rightarrow V \times S$ which means that
get accepts a location and gives a "state transformation" from states to values and states. In the collecting semantics this becomes \( L \rightarrow P(S) \rightarrow I(V \times S) \) so the idea is that identifiers always denote single locations. If this is not the case the metalanguage may be unsuitable because \( t ::= gt \) is not allowed. For the purposes of this thesis this is not felt too restrictive. It is in accord with the spirit of abstract interpretation where "state transformations" (i.e. ft-types) are of prime interest. Further, if \( t ::= gt \) is allowed then the close connection between standard semantics and the collecting semantics that is proved in the next chapter is likely to fail (see the discussion after theorem 3.3:14).

There is no mechanism for binding a top-level type into a bottom-level type. The need for doing so does not arise in this thesis. Also the possibility of viewing ft-types as "code" may suggest that this would not always be meaningful. Consequently the bottom-level is in a sense "below" the top-level. Finally, it should be explained why \( V \vdash gt_1 \rightarrow gt_2 \) iff \( \emptyset \vdash gt_1 \) and \( \emptyset \vdash gt_2 \). This enforces that no domain variable ranging over a bottom-level type (i.e. gt-type) can be bound (by rec) in the top-level and this simplifies part of the development. Another way to enforce this might be to distinguish between bottom-level and top-level domain variables and assume they can only be bound by a rec operator of the appropriate level.

Remark. Since a development of abstract interpretation involves making operational ideas denotational it may be well to ask whether the type constructors are natural from an operational point of view. One idea might be that functions in the top-level should be total functions between domains and that cartesian product corresponds to
tupling. Further, one might believe that functions in the bottom-level should be partial functions between predomains (domains possibly without \(\mathbb{1}\)) and that cartesian product corresponds to tupling. We shall use total functions between domains but by using the smash product much the same effect is obtained. This suggests that smash product in the top-level and cartesian product in the bottom-level are not "natural". In fact these two type constructors give technical problems later (in particular in sections 4.2 and 4.4).

///

THE EXPRESSIONS

Next the syntax of expressions is defined. Contrary to the types it is possible to use "top-level notation" inside "bottom-level notation" and vice versa. The definition below annotates the various pieces of syntax with the corresponding domain constructors and suggests the intended meaning. Further hints are presented afterwards and the formal semantics is given in section 2.3.

\[
e := (e_1, \ldots, e_k) \mid \text{def } e' \mid \text{down } e' \\
\quad \mid \text{in}_1 e' \mid \text{is}_i e' \mid \text{out}_i e' \\
\quad \mid \text{up } e' \mid \text{def } e' \mid \text{down } e' \\
\quad \mid \lambda x : t. e' \mid e_1(e_2) \mid x \\
\quad \mid \text{mkrec } e' \mid \text{unrec } e' \\
\quad \mid e_1 \rightarrow e_2, e_3 \mid \text{case } e_1, \ldots, e_k \\
\quad \mid \text{tuple } e_1, \ldots, e_k \mid \text{take}_i \\
\quad \mid \text{smashtuple } e_1, \ldots, e_k \mid \text{smashtake}_i \\
\quad \mid \text{in}_1 \mid \text{case } e_1, \ldots, e_k
\]

(for \(X\): tupling, indexing)

(for \(\Psi\): tupling, indexing)

(for \(+\): inject, test, extract)

(for \(\downarrow\): inject, test, extract)

(for \(\rightarrow\))

(for rec: inject, extract)

(conditional, fixed points, constants)

(for \(\Psi\))

(for \(+\))
We shall use parentheses freely. The constants $f_i$ are left unspecified but at some points we shall need to assume that certain functions are included. The most common example of this is the identity function $id$ of type $gt \rightarrow gt$.

The static well-formedness relation defined below gives the respective types and this may clarify the intention with the constructs.

The syntax for top-level constructors is much as in /Sto77/ and /PloLN/. The syntax for bottom-level constructors are in a different style as will be motivated later. It may therefore help to sketch the intended meaning in the standard semantics:

\[
\begin{align*}
\text{(tuple } f_1, f_2)(v_1, v_2) &= (f_1(v_1), f_2(v_2)) \\
\text{(case } f_1, f_2)(\text{in } v) &= f_1(v) \\
\text{(lift } f)(\text{up } v) &= f(v) \\
\text{(cond } f_1, f_2, f_3)(v) &= f_1(v) \rightarrow f_2(v), f_3(v) \\
&\text{i.e. } \text{ LT, } f_2(v) \text{ or } f_3(v) \text{ depending on } f_1(v) = \text{ LT, } \text{TT or FF} \\
(f_1 \mu f_2)(v) &= f_1(f_2(v))
\end{align*}
\]

The static well-formedness relation $\text{tenv } \vdash e : t$ means that $e$ has type $t$ in type environment $\text{tenv}$. A type environment is a function from a finite subset of identifiers ($x$, $y$, $x'$ etc.) to the set of closed top-level types. It will be assumed throughout that $t$ is closed. The relation is defined structurally on $e$ by the following inference rules and axioms (some of which have side conditions).
<table>
<thead>
<tr>
<th>Rule</th>
<th>Type Environment</th>
<th>Type Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>product</strong></td>
<td>( \text{tenv} \vdash e_i : t_i ) (all ( i ))</td>
<td>( \text{tenv} \vdash e : t_1 \ldots t_k )</td>
</tr>
<tr>
<td></td>
<td>( \text{tenv} \vdash (e_1, \ldots, e_k) : t_1 \ldots t_k )</td>
<td>( \text{tenv} \vdash e_i : t_i )</td>
</tr>
<tr>
<td><strong>smash</strong></td>
<td>( \text{tenv} \vdash e_i : t_i ) (all ( i ))</td>
<td>( \text{tenv} \vdash e : t_1 \ldots t_k )</td>
</tr>
<tr>
<td></td>
<td>( \text{tenv} \vdash (e_1, \ldots, e_k) : t_1 \ldots t_k )</td>
<td>( \text{tenv} \vdash e_i : t_i )</td>
</tr>
<tr>
<td><strong>product</strong></td>
<td>( \text{tenv} \vdash (\forall e_1, \ldots, e_k) : t_1 \ldots t_k )</td>
<td>( \text{tenv} \vdash e_i : t_i )</td>
</tr>
<tr>
<td></td>
<td>( \text{tenv} \vdash e : t_1 \ldots t_k )</td>
<td>( \text{tenv} \vdash t_1 \ldots t_k )</td>
</tr>
<tr>
<td></td>
<td>( \text{tenv} \vdash e_i : t_i )</td>
<td>( \text{tenv} \vdash e_i : e : T )</td>
</tr>
<tr>
<td><strong>sum</strong></td>
<td>( \text{tenv} \vdash e_i : t_i )</td>
<td>( \text{tenv} \vdash e : t_1 \ldots t_k )</td>
</tr>
<tr>
<td></td>
<td>( \text{tenv} \vdash e : t_1 \ldots t_k )</td>
<td>( \text{tenv} \vdash e : t_i )</td>
</tr>
<tr>
<td></td>
<td>( \text{tenv} \vdash e : t_1 \ldots t_k )</td>
<td>( \text{tenv} \vdash e : t_i )</td>
</tr>
<tr>
<td></td>
<td>( \text{tenv} \vdash e : t_1 \ldots t_k )</td>
<td>( \text{tenv} \vdash e : e : T )</td>
</tr>
<tr>
<td></td>
<td>( \text{tenv} \vdash e : e : T )</td>
<td>( \text{tenv} \vdash e : t_i )</td>
</tr>
<tr>
<td><strong>lifting</strong></td>
<td>( \text{tenv} \vdash e : t )</td>
<td>( \text{tenv} \vdash e : t )</td>
</tr>
<tr>
<td></td>
<td>( \text{tenv} \vdash e : t )</td>
<td>( \text{tenv} \vdash e : t )</td>
</tr>
<tr>
<td></td>
<td>( \text{tenv} \vdash e : t )</td>
<td>( \text{tenv} \vdash e : t )</td>
</tr>
<tr>
<td><strong>function</strong></td>
<td>( \text{tenv}[t_1/x] \vdash e : t_2 )</td>
<td>( \text{tenv} \vdash e : t )</td>
</tr>
<tr>
<td></td>
<td>( \text{tenv} \vdash \lambda x : t_1 . e : t_1 \rightarrow t_2 )</td>
<td>( \text{tenv} \vdash e : t )</td>
</tr>
<tr>
<td></td>
<td>( \text{tenv} \vdash e_1 : t_1 \rightarrow t_2 )</td>
<td>( \text{tenv} \vdash e_2 : t_1 )</td>
</tr>
<tr>
<td></td>
<td>( \text{tenv} \vdash e_2 : t_1 )</td>
<td>( \text{tenv} \vdash e_1(e_2) : t_2 )</td>
</tr>
<tr>
<td></td>
<td>( \text{tenv} \vdash e : t ) if ( \text{tenv}(x) = t )</td>
<td>( \text{tenv} \vdash e : t )</td>
</tr>
<tr>
<td><strong>recursive</strong></td>
<td>( \text{tenv} \vdash t )</td>
<td>( \text{tenv} \vdash t )</td>
</tr>
<tr>
<td><strong>types</strong></td>
<td>( \text{tenv} \vdash \text{mkrec } e : \text{recX} . t )</td>
<td>( \text{tenv} \vdash t )</td>
</tr>
<tr>
<td></td>
<td>( \text{tenv} \vdash \text{unrec } e : t[\text{recX} . t/X] )</td>
<td>( \text{tenv} \vdash e : t[\text{recX} . t/X] )</td>
</tr>
<tr>
<td><strong>cond.</strong></td>
<td>( \text{tenv} \vdash e_1 : T )</td>
<td>( \text{tenv} \vdash e : t )</td>
</tr>
<tr>
<td></td>
<td>( \text{tenv} \vdash e_2 : t )</td>
<td>( \text{tenv} \vdash e : t )</td>
</tr>
<tr>
<td></td>
<td>( \text{tenv} \vdash e_3 : t )</td>
<td>( \text{tenv} \vdash e : t )</td>
</tr>
<tr>
<td></td>
<td>( \text{tenv} \vdash e_1 \rightarrow e_2 , e_3 : t )</td>
<td>( \text{tenv} \vdash e : t )</td>
</tr>
<tr>
<td><strong>fixed</strong></td>
<td>( \text{tenv} \vdash e : t )</td>
<td>( \text{tenv} \vdash e : t )</td>
</tr>
<tr>
<td><strong>points</strong></td>
<td>( \text{tenv} \vdash X : e : t )</td>
<td>( \text{tenv} \vdash e : t )</td>
</tr>
<tr>
<td><strong>const.</strong></td>
<td>( \text{tenv} \vdash f_i : t_i )</td>
<td>( \text{tenv} \vdash f_i : t_i )</td>
</tr>
<tr>
<td></td>
<td>( \text{tenv} \vdash e_0 , \ldots , e_i : t )</td>
<td>( \text{tenv} \vdash e_0 , \ldots , e_i : t )</td>
</tr>
</tbody>
</table>

Let us pause at this point in listing the defining rules for the relation and clarify the notation. The type of truthvalues is \( T \), and \( \text{tenv}[t'/x] \) denotes the type environment that is \( t' \) upon \( x \) and
otherwise as tenv. Syntactic substitution \( t[t'/X] \) is defined as in the \( \lambda \)-calculus (see e.g. /Sto77/). To be precise in \( i \) e should be written \( i(t_1, \ldots, e, \ldots, t_k) \) but the simpler notation should not cause confusion. A similar ambiguity occurs with \( \text{mkrec} \) because a term \( t[\text{recx}.t/X] \) needs not determine \( t \) uniquely.

The types \( t_i \) of constants \( f_i \) will not be specified but it is important for the development to restrict the types that are allowed. (An example is the definition of \( \text{view}_t \) in section 3.3.) This is achieved using the predicate \( \text{CP}(V, t) \). A type \( t \) is said to be contravariantly pure iff \( \text{CP}(\emptyset, t) \) holds and the idea is that this is the case iff no \( f_t \) occurring in \( t \) is in the domain of a function space construction. The intention with \( V \) is that it contains those domain variables that denote a type that contains some \( f_t \). The definition of \( \text{CP}(V, t) \) is by structural induction on \( t \) and uses the auxiliary predicate \( P(V, t) \) (for no \( f_t \) whatsoever).

<table>
<thead>
<tr>
<th>( t )</th>
<th>( \text{CP}(V, t) )</th>
<th>( P(V, t) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A_1 )</td>
<td>( \top )</td>
<td>( \top )</td>
</tr>
<tr>
<td>( t_1 \cdot \ldots \cdot t_k )</td>
<td>( \bigwedge_{i=1}^k \text{CP}(V, t_i) )</td>
<td>( \bigwedge_{i=1}^k P(V, t_i) )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( t_1 \to t_2 )</td>
<td>( P(V, t_1) \wedge \text{CP}(V, t_2) )</td>
<td>( \bigwedge_{i=1}^2 P(V, t_i) )</td>
</tr>
<tr>
<td>( \text{recx}.t' )</td>
<td>( \text{CP}(V_u[X], t') \vee P(V, t') )</td>
<td>( P(V, t') )</td>
</tr>
<tr>
<td>( X )</td>
<td>( \top )</td>
<td>( \text{X}	ext{e}V )</td>
</tr>
<tr>
<td>( f_t )</td>
<td>( \top )</td>
<td>( \text{ff} )</td>
</tr>
</tbody>
</table>

Some examples may help: \( \text{recx}.N \to X \to X(N \to N) \) is contravariantly pure and so is \( \text{recx}.N \to (X \to X) \) but \( \text{recx}.X \to N \to N \) is not (unfold the definition).
We now continue listing the rules for the well-formedness relation. Recall that $T$ is the type of the truth-values in the bottom-level.

\[
\text{product} \quad \text{tenv}$- $e_i : gt \rightarrow gt_1$ (all i) \hfill \text{tenv}$- $\text{tuple } e_i, ..., e_k : gt \rightarrow gt_1 \rightarrow \ldots \rightarrow gt_k$ \\
\text{smash} \quad \text{tenv}$- $e_i : gt \rightarrow gt_1$ (all i) \hfill \text{tenv}$- $\text{smashtuple } e_i, ..., e_k : gt \rightarrow gt_1 \rightarrow \ldots \rightarrow gt_k$ \\
\text{sum} \quad \text{tenv}$- $\text{in}_i : gt \rightarrow gt_1 + \ldots + gt_k$ \hfill \text{tenv}$- e_i : gt_1$ (all i) \hfill \text{tenv}$- \text{case } e_i, ..., e_k : gt_1 + \ldots + gt_k \rightarrow gt$

\[
\text{lifting} \quad \text{tenv}$- e : gt \rightarrow gt'$ \hfill \text{tenv}$- \text{up} : gt \rightarrow gt'$ \\
\text{rec.} \quad \text{tenv}$- \text{fold} : gt[\text{rec}X. gt/X] \rightarrow \text{rec}X. gt$ \hfill \text{tenv}$- \text{unfold} : \text{rec}X. gt \rightarrow gt[\text{rec}X. gt/X]$

\[
\text{cond.} \quad \text{tenv}$- e_1 : gt \rightarrow T \hfill \text{tenv}$- e_2 : gt \rightarrow gt'$ \hfill \text{tenv}$- e_3 : gt \rightarrow gt'$ \\
\text{comp.} \quad \text{tenv}$- e_1 : gt_2 \rightarrow gt_1 \hfill \text{tenv}$- e_2 : gt_3 \rightarrow gt_2$ \hfill \text{tenv}$- e_1, e_2 : gt_3 \rightarrow gt_1$

Again there are minor ambiguities. For example $\text{in}_i$ should be indexed by $gt_1, \ldots, gt_k$ and perhaps underlined to distinguish it from the similar syntax in the top-level metalanguage.

We now comment upon the syntax of expressions. The syntax relating to the top-level type constructors focuses on elements, as
in \((v_1, \ldots, v_k)\) that is a tuple, whereas the syntax relating to the bottom-level type constructors focuses on functions, as in 
\((\text{tuple } f_1, \ldots, f_k)\) that is a function. It is possible to use a syntax focusing on functions throughout (see /PloLN/) but it seems to be more common to focus on elements (e.g. /Sto77/). For the syntax relating to bottom-level types it seems necessary to focus on functions, e.g. in the definition of cond in section 3.3. This is in accord with abstract interpretation where it is certain "state transformation" functions that are of interest. Also note that the constructs tuple, smashtuple, case, lift, cond and \(\emptyset\) may be viewed as functionals of type \(ft^n \rightarrow ft\). They are the prime examples of "constants" whose types are not contravariantly pure. In particular there are no constructs of type \(ft \rightarrow T\) (or \(gt \rightarrow T\)) so in a sense the dynamic semantics cannot influence the static semantics.

2.2 PRELIMINARY DOMAIN THEORY

This section reviews/develops the domain theory needed to define the standard semantics of the metalanguage. Most of the material is quite standard and can be found elsewhere, e.g. in /SmP182/ in a terse presentation and in /PloLN/. The material originates with Dana Scott who gives a "modern" presentation of the theory in /Sco82/ but in a different flavour than used here. Less standard results will be proved but much will be stated without proof. A few concepts from category theory /ArMa75/ are used but no previous knowledge is assumed.
Partially ordered sets, complete lattices and chains were defined in chapter 1. A cpo (countably complete partial order) is a partially ordered set $D = (D, \leq)$ with a least element and least upper bounds of chains. A subset $\mathcal{H}$ of $D$ is directed if each finite subset of $\mathcal{H}$ has an upper bound in $\mathcal{H}$. Then one can prove that a partially ordered set is a cpo iff it has a least element and every countable directed set has a least upper bound. A cpo $D$ is flat iff the formula

$$d \leq d' \Leftrightarrow (d = \bot \lor d = d')$$

is true. Example flat cpo's are:

<table>
<thead>
<tr>
<th>N_\bot: 0 1 2 ...</th>
<th>T_\bot: tt ff</th>
<th>D: T</th>
<th>U: \bot</th>
</tr>
</thead>
<tbody>
<tr>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
</tr>
</tbody>
</table>

In general, for any set $S$ there is the flat cpo $S_\bot$ with elements $S \cup \{ \bot \}$ (assuming a disjoint union). The set $\emptyset_\bot$ will be of special interest and is denoted $U$. We shall sometimes write $\emptyset$ for $\{ \bot \}_\bot$.

An element $d$ of a cpo $D$ is finite iff for every chain $(d_n)_n$ we have that $d \leq \bigcup\nolimits_{n} d_n$ implies that $d \leq d_n$ for some $n$. The idea is that finite elements correspond to approximative information about fully defined elements /Sco82/. This is exemplified by the cpo of partial functions from $\mathbb{N}$ to $\mathbb{N}$ ordered by subset on the graphs of the functions. Here the finite elements are the partial functions that have a finite domain. For this cpo every element is the least upper bound of a chain of finite elements. This is often the case and motivates:

**Definition** A cpo $D$ is countably algebraic (or just algebraic) if the set $B_D = \{ b \in D \mid b \text{ is finite} \}$ is countable and each $d \in D$ has
d = \bigcup_n b_n \text{ for some chain } \{b_n\} \text{ with each } b \in B_D. \quad ///

We shall mostly assume cpo's to be algebraic as this simplifies some constructions; this assumption is made in many developments of domain theory. One can show that in an algebraic cpo every directed set has a least upper bound. Clearly \(S_4\) is algebraic iff \(S\) is countable. It is convenient to know:

Fact 2.2:1. A cpo \(D\) is algebraic iff there is a countable subset \(H\) whose elements are finite and where each element \(d\) of \(D\) can be written \(\bigcup H'\) for a directed subset \(H'\) of \(H\). The set \(H\) then equals \(B_D\). \quad ///

Not all constructions upon cpo's that we will consider preserve algebraicity. We therefore define an additional property and we shall see that the conjunction of the two properties is preserved.

Definition A cpo \(D\) is \textbf{consistently complete} iff \(\bigcup H\) exists for every subset \(H\) of \(D\) that has an upper bound in \(D\). \quad ///

A cpo \(D\) is consistently complete iff the partially ordered set \(D'\) obtained by adding a new greatest element to \(D\) is a complete lattice. It is generally believed that in practice cpo's are algebraic and consistently complete; such cpo's are often called \textit{domains} (essentially as in /Sco82/). On the other hand it is often unnatural to assume cpo's to be complete lattices because the "artificial" greatest element may invalidate equalities that hold operationally (see e.g. /PloLN/).

We now define several ways of defining a cpo in terms of others. Let \(D, D_1, ..., D_k\) be cpo's and assume that \(k \geq 2\). The \textbf{cartesian}
product $D_1 \times \ldots \times D_k$ or $\prod_i D_i$ has as elements tuples $(d_1, \ldots, d_k)$ with $d_i \in D_i$ and is partially ordered componentwise, i.e.

$$(d_1, \ldots, d_k) \leq (d_1', \ldots, d_k') \text{ iff for all } i \text{ that } d_i \leq d_i'. $$

This gives a cpo with $\bigcup$ and $\prod$ defined componentwise and it is algebraic/consistently-complete/a domain if all $D_i$ are. We write $(d_1, \ldots, d_k)^{|i}$ for $d_i$. Closely associated with the cartesian product is the smash product $D_1 \wedge \ldots \wedge D_k$. The elements are those tuples $(d_1, \ldots, d_k)$ where some $d_i$ is $\bot$ iff all are. The partial order is as before and defines a cpo that is algebraic / consistently complete/ a domain if all $D_i$ are. The function $\text{smash}: D_1 \times \ldots \times D_k \rightarrow D_1 \wedge \ldots \wedge D_k$

sends $(d_1, \ldots, d_k)$ to $(\bot, \ldots, \bot)$ and otherwise acts as the identity.

The coalesced sum $D_1 + \ldots + D_k$ or $\sum_i D_i$ has as elements pairs $(i, d_i)$ with $d_i \in D_i$ and $d_i \neq \bot$ and additionally the element $\bot$. The partial order has $\bot$ to be least and $(i, d_i) \leq (j, d_j)$ iff $i = j$ and $d_i \leq d_j$. This gives a cpo that is algebraic/ consistently complete/ a domain if all $D_i$ are. The function $\text{in}_i: D_i \rightarrow D_1 + \ldots + D_k$ sends $\bot$ to $\bot$ and otherwise $d_i$ to $(i, d_i)$. Similarly $\text{out}_i: D_1 + \ldots + D_k \rightarrow D_i$ sends $(i, d_i)$ to $d_i$ and otherwise gives $\bot$. Finally, $\text{is}_i: D_1 + \ldots + D_k \rightarrow T$ sends $(i, d_i)$ to $tt$ and $\bot$ to $\bot$ and otherwise gives $ff$.

The lifting $D_\bot$ has as elements a new least element $\bot$ and for each element $d \in D$ the element $(0, d)$. The partial order has

$$(0, d) \leq (0, d') \text{ iff } d \leq d'. $$

This gives a cpo that is algebraic/ consistently complete/ a domain if $D$ is. The function $\text{up}: D \rightarrow D_\bot$ sends $d$ to $(0, d)$ and $\text{down}: D_\bot \rightarrow D$ uses $\bot$ to $\bot$ and $(0, d)$ to $d$.

Finally, $\text{def}: D_\bot \rightarrow T$ sends $\bot$ to $\bot$ and $(0, d)$ to $tt$. 

46
Before defining function spaces we explain the notion of continuity. A function \( f: D_1 \rightarrow D_2 \) is \textit{continuous} iff it is monotonic and for each chain \((d_n)_n\) of \(D_1\) it is the case that \(f(\bigcup_n d_n) = \bigcup_n f(d_n)\).

Note that the monotonicity of \( f \) ensures that \((f(d_n))_n\) is a chain in \(D_2\) and that \(f(\bigcup_n d_n) \supseteq \bigcup_n f(d_n)\). It is helpful to know that \(f(\bigcup \mathcal{H}) = \bigcup\{f(h) \mid h \in \mathcal{H}\}\) holds when \( f \) is continuous and \( \mathcal{H} \) is countable and directed. If \( D_1 \) is algebraic this holds even when \( \mathcal{H} \) is not countable. It is not hard to show that \( f: D_1 \times \ldots \times D_k \rightarrow D \) is continuous iff for all \( d_1, \ldots, d_k \) and \( i \) the equation \( g(d) = f(d_1, \ldots, d_i, \ldots, d_k) \) defines a continuous function \( g: D_i \rightarrow D \). All the functions defined so far in this section are continuous and it is generally believed that all functions should be continuous /Sco82/.

The function space \( D_1 \rightarrow D_2 \) has as elements all continuous functions from \( D_1 \) to \( D_2 \). The partial order is defined pointwise, i.e. \( f \leq f' \) iff \( f(d) \leq f'(d) \) holds for all elements \( d \) of \( D_1 \). This gives a cpo with least upper bounds determined pointwise. It is a domain if \( D_1 \) and \( D_2 \) are but needs not be algebraic if \( D_1 \) and \( D_2 \) are only algebraic /PloLN/.

If \( f \) is a continuous function upon a cpo it has a least fixed point \( \text{LFP}(f) = \bigcup_n f^n(\bot) \). Here \( f^n \) is the \( n \)-fold composition of \( f \) and the monotonicity of \( f \) gives that \((f^n(\bot))_n\) is a chain. Clearly \( \bigcup_n f^n(\bot) \) is a fixed point and to see it is less than any other fixed point \( d \) one shows \( f^n(\bot) \leq d \) by induction on \( n \) and then deduces that \( \bigcup_n f^n(\bot) \leq d \). The formula \( \text{LFP}(f) = \bigcap \{d \mid f(d) \leq d\} \) suggested by /Tar55/ still holds: For \( \bigcup_n f^n(\bot) \) is an element of the set on the righthand side and any other element \( d \) has \( \bigcup_n f^n(\bot) \leq d \) as can be shown by a proof similar to the proof that \( \bigcup_n f^n(\bot) \) is the least fixed point.
Using \( \text{LFP}(f) = \bigcup_n f^n(\downarrow) \) it is not hard to show that LFP is a continuous function from \( D \rightarrow D \) to \( D \) (see e.g. /Sto77/).

The proof that \( \bigcup_n f^n(\downarrow) \) is the least fixed point is typical of many proofs. To facilitate such proofs one can use the following proof rule. A predicate \( P \) on the cpo \( D \) is admissible (inclusive) iff \( P(\downarrow) \) is true and if for every chain \( (d_n) \) the truth of all \( P(d_n) \) gives the truth of \( P(\bigcup_n d_n) \). Then for a continuous \( f:D \rightarrow D \) and admissible \( P \) we have the rule of fixed point induction (Scott-induction)

\[
P(x) \Rightarrow P(f(x))
\]

\[
P(\text{LFP}(f))
\]

For since \( P \) is admissible \( P(\downarrow) \) holds and induction on \( n \) shows that all the \( P(f^n(\downarrow)) \) hold and hence \( P(\bigcup_n f^n(\downarrow)) \) holds. (Often \( P(\downarrow) \) is made a premise of the rule and then not assumed in the definition of admissibility.) As an example the predicate \( P(x) \) defined by

\[
Vd: (f(d) \leq d \Rightarrow x \leq d)
\]

may be used to prove that \( \text{LFP}(f) \leq d \) whenever \( f(d) \leq d \).

**CATEGORICAL FORMULATION**

The remainder of this section is devoted to the solution of (parameterised) recursive domain equations. The development is closer to /SmPl82/ and /PloLN/ than to the universal domain approach /Sco76/. This calls for the use of some categorical concepts to expedite the development. The necessary concepts are explained below and the constructs defined earlier are formulated in this setting. The two central concepts are those of category and functor /ArMa75,Mac71/. 
A category allows for a concise way of naming the cpo's and functions of interest at some point in the development. A category consists of a set of objects and for each two objects A and B a set of morphisms \( g: A \to B \) from A to B. (Some authors require that a morphism like \( g \) must uniquely determine its domain A and range B.)

The most common category is \( \mathbf{Set} \) that has sets as objects and total functions as morphisms. (Strictly speaking not all sets can be allowed as the class of objects then is "too big" to be a set; ways around this are discussed in /Mac71 p. 21-24/.) We shall make much use of \( \mathbf{CPO} \) that has cpo's as objects and (total) continuous functions as morphisms. To be a category there must be a composition of morphisms, as in \( f'g: A \to C \) that is the composition of \( f: B \to C \) and \( g: A \to B \), and for each object A an identity morphism \( \text{id}_A: A \to A \) (usually written \( \text{id} \)). They must satisfy the associative law \( (f'g)'h = f'(g'h) \) and the identity laws \( f'\text{id} = f \) and \( \text{id}'f = f \). In \( \mathbf{Set} \) and \( \mathbf{CPO} \) composition is ordinary function composition and the identities are the identity functions. For \( \mathbf{CPO} \) one must check that the identity function is continuous and that composition of continuous functions gives a continuous function.

It is convenient to name some additional categories. If only strict continuous functions are allowed the category is \( \mathbf{CPO_S} \). If all continuous functions are allowed but object must be domains (i.e. algebraic and consistently complete cpo's) the category is \( \mathbf{ACC} \). Similarly \( \mathbf{ACC_S} \) requires functions to be strict.

The category \( \mathbf{ACC_S} \) is said to be a subcategory of \( \mathbf{CPO} \) because all objects and morphisms of the former are in the latter and because the composition and identities agree. It is also a subcategory of
A product category like \( \text{CPO} \times \text{CPO} \) or \( \text{CPO}^k \) has objects and morphisms to be tuples with one component for each category in the product. In particular \( \text{CPO}^1 \) may be identified with \( \text{CPO} \) and \( \text{CPO}^0 \) is a category with one object NIL and one morphism \( \text{id}_{\text{NIL}} \). If \( f_1 : A_1 \to B_1 \) and \( f_2 : A_2 \to B_2 \) are morphisms then \( (f_1, f_2) : (A_1, A_2) \to (B_1, B_2) \) is a morphism in the product category. Composition and identities are defined componentwise, i.e. \( (f_1, f_2)(g_1, g_2) = (f_1g_1, f_2g_2) \) and \( \text{id}_{(A_1, A_2)} = (\text{id}_{A_1}, \text{id}_{A_2}) \).

A covariant functor (sometimes abbreviated to functor) allows for a precise description of the effect of domain constructors (e.g. lifting). A covariant functor \( F \) from a category \( A \) to a category \( B \) consists of two maps: one that sends an object \( A \) of \( A \) to an object \( F(A) \) of \( B \) and one that sends a morphism \( f : A \to A' \) of \( A \) to a morphism \( F(f) : F(A) \to F(A') \) of \( B \). It must satisfy the composition law \( F(f'g) = F(f)F(g) \) and the identity law \( F(\text{id}_A) = \text{id}_{F(A)} \).

It is now straightforward to turn lifting, cartesian product, smash product and coalesced sum into covariant functors. For lifting we define a functor \( (\_)_L : \text{CPO} \to \text{CPO} \) as follows. The effect upon objects has already been defined and for a morphism \( f : D \to E \), the functor gives \( (f)_L : D_L \to E_L \) that sends \( \bot \) to \( \bot \) and \( (0, d) \) to \( (0, f(d)) \).

It is straightforward to check that the composition and identity laws hold. It will become clear when solving domain equations that it is helpful that the domain constructors are defined as functors, i.e. have an effect upon morphisms.

Cartesian product is turned into a covariant functor \( \times : \text{CPO}^k \to \text{CPO} \) by defining \( f_1 \times \ldots \times f_k \) as the function that maps \( (v_1, \ldots, v_k) \) to \( (f_1(v_1), \ldots, f_k(v_k)) \). These functors preserve strictness and therefore
specialise to $\text{CPOs}$ (i.e. we could have used $\text{CPOs}$ above). To turn smash product and coalesced sum into covariant functors we need to assume strictness, i.e. they become functors $\text{CPOs}^k \to \text{CPOs}$. Smash product has $f_1 \ldots \times f_k$ to be the restriction of $\text{smash}^* f_1 \times \ldots \times f_k$ to the smash product. Coalesced sum has $f_1+\ldots+f_k$ to be the function mapping $(i,d_i)$ to $(i,f_i(d_i))$ when $f_i(d_i)\neq 1$ and giving 1 in all other cases. All of these functors preserve algebraicity and consistent completeness and therefore specialise to $\text{ACCS}$.

We shall consider two ways of combining functors. The tupling of covariant functors $F_i : L \to K_i$ is the covariant functor $(F_1, \ldots, F_k) : L \to K_1 \times \ldots \times K_k$ defined by $(F_1, \ldots, F_k)(A) = (F_1(A), \ldots, F_k(A))$ and similarly on morphisms. (So $(F_1, \ldots, F_k)$ is a functor and not a tuple of functors!) The composition of covariant functors $F : L \to M$ and $F' : M \to N$ is the covariant functor $F \circ F' : L \to N$ defined on objects by $(F \circ F')(A) = F(F'(A))$ and similarly on morphisms. The covariant identity functor $\text{Id}$ maps an object to itself and similarly for morphisms. If $A$ is an object of $L$ then a covariant constant functor $K_A : M \to L$ is defined by $K_A(B) = A$ and $K_A(f) = \text{id}_A$. The $i$'th projection functor $P_i : K_1 \times \ldots \times K_k \to K_i$ is the covariant functor defined by $P_i(A_1, \ldots, A_k) = A_i$ and similarly on morphisms.

The function space construction is more troublesome. The effect upon objects has already been defined and upon morphisms one may define $f_1 \Rightarrow f_2$ as the function sending $h$ to $f_2 \circ h \circ f_1$. The identity law holds but for composition we get

$$(f_1 \Rightarrow f_2) \cdot (g_1 \Rightarrow g_2) = g_1 \cdot f_1 \Rightarrow f_2 \cdot g_2$$

corresponding to $(f_1 \Rightarrow f_2) : (B_1 \Rightarrow A_2) \Rightarrow (A_1 \Rightarrow B_2)$ when $f_1 : A_1 \to B_1$. This functor is said to be contravariant in its left argument and
covariant in its right. It gives a functor \( \rightarrow: \text{CPO}^2 \rightarrow \text{CPO} \) that specialises to \( \text{CPOs} \), \( \text{ACC} \) and \( \text{ACCS} \).

Contravariance is troublesome to work with, e.g. because \( \rightarrow'((\text{Id}, \text{Id}) \) is neither contravariant nor covariant. To enable all functors to be covariant we shall use the category \( \text{CPO}_{\mathbb{P}} \) instead of \( \text{CPOs} \). This category has the same objects as \( \text{CPOs} \) but a morphism \( f: A \rightarrow B \) is a pair \( (f_1, f_2) \) of \( \text{CPOs} \) morphisms \( f_1: A \rightarrow B \) and \( f_2: B \rightarrow A \).

In other words a morphism of \( \text{CPOs} \) is a pair of strict continuous functions. Composition is defined by

\[
(f_1, f_2)^* (g_1, g_2) = (f_1^* g_1, g_2^* f_2)
\]

and may be viewed as being contravariant in its right argument.

It is this property that will allow contravariance over \( \text{CPOs} \) to be disguised as covariance over \( \text{CPOs} \). The identity morphism on \( A \) is the pair \( (\text{id}, \text{id}) \) of identity functions \( \text{id}: A \rightarrow A \). When \( f = (f_1, f_2) \) it is convenient to write \( f_1 = f_1^* \) and \( f_R = (f_2, f_1) \).

A symmetric functor /PloP/) (or a domain functor /Rey74/) \( G: \text{CPOs}^k \rightarrow \text{CPOs} \) is a covariant functor that satisfies the law

\[
G(f_1, \ldots, f_k)^R = G(f_1^R, \ldots, f_k^R)
\]

When \( k=1 \) this means that \( G((g_1, g_2)) = (h_1, h_2) \) implies that \( G((g_2, g_1)) = (h_2, h_1) \). Let \( F: \text{CPOs}^k \rightarrow \text{CPOs} \) be a mixed covariant and contravariant functor. A symmetric functor \( F^S: \text{CPOs}^k \rightarrow \text{CPOs} \) may be defined by

\[
F^S(A_1, \ldots, A_k) = F(A_1, \ldots, A_k)
\]

\[
F^S(f_1, \ldots, f_k) = (F(f_1^j_1, \ldots, f_k^j_k), \ldots, F(f_1^j_k, (3-j_1), \ldots, f_k^j_k, (3-j_k)))
\]

where \( j_i = 1 \) if \( F \) is covariant in the \( i \)-th argument and otherwise \( j_i = 2 \).

This means that all of \( \times, \vdash, +, (\_)_\perp \rightarrow \) may be viewed as symmetric
functors over $\text{CPO}^2$s. For example

$\lhd^S( (f_1, f_2), (g_1, g_2) ) = ( f_1 + g_1, f_2 + g_2 )$

$\rhd^S( (f_1, f_2), (g_1, g_2) ) = ( f_2 \rightarrow g_1, f_1 \rightarrow g_2 )$

The use of $\text{CPO}^2$s rather than $\text{CPO}^2$ is because $+$ and $\times$ were only defined as functors over $\text{CPO}^2$s. Also the identity functor, constant functors and projection functors can be transformed to $\text{CPO}^2$s using $^S$ and they "give themselves".

Most of the categories considered so far have some additional structure. A $\text{cpo-category}$ is a category where the set of morphisms between any two objects forms a cpo and where composition is continuous with respect to the partial orders. (This corresponds to a $\text{CPO-category}$ in /ArMa75/.) For $\text{CPO}^2$s the partial order is defined pointwise (as in the function space construct), for $\text{CPO}^2$s it is defined componentwise (i.e. $(f_1, f_2) \leq (g_1, g_2)$ iff $f_1 \leq g_1$ and $f_2 \leq g_2$) and for products of categories is defined componentwise (as in the cartesian product of cpo's). A functor between two cpo-categories is $\text{locally continuous}$ / $\text{locally monotonic}$ if its effect upon morphisms is continuous / monotonic. For a (mixed covariant and contravariant) functor $F: \text{CPO}^2 \rightarrow \text{CPO}^2$s local continuity just means that

$F( \bigcup_n f_n, \ldots, \bigcup_n h_n ) = \bigcup_n F(f_n, \ldots, h_n)$

for all chains $(f_n)_{n}$, ..., $(h_n)_{n}$ of strict continuous functions.

Clearly local continuity implies local monotonicity and if $F: \text{CPO}^2 \rightarrow \text{CPO}^2$s is locally continuous / locally monotonic then so is $F^S: \text{CPO}^2 \rightarrow \text{CPO}^2$s. All the functors $(.)^L$, $\times$, $\times$, $\rightarrow$, $\text{Id}$, $K_A$, $P_i$ are locally continuous and composition and tupling preserve this property.
The notion of an isomorphism is essential to explain what is meant by a solution to a recursive domain equation. An isomorphism in a category is a morphism \( \theta: A \to B \) for which there exists a (necessarily unique) morphism \( \theta^{-1}: B \to A \) such that \( \theta \circ \theta^{-1} = \text{id}_A \) and \( \theta^{-1} \circ \theta = \text{id}_B \). The morphism \( \theta^{-1} \) is called the inverse of \( \theta \) and is itself an isomorphism with inverse \( (\theta^{-1})^{-1} = \theta \). We write \( A \cong B \) if there is an isomorphism from \( A \) to \( B \). The identity is clearly an isomorphism with itself as inverse and the composition of isomorphisms give an isomorphism and one has the formula \( (\theta_1 \circ \theta_2)^{-1} = \theta_2^{-1} \circ \theta_1^{-1} \). If \( F \) is a covariant functor and \( \theta_1, \ldots, \theta_k \) are all isomorphisms then so is \( F(\theta_1, \ldots, \theta_k) \) and its inverse is \( F(\theta_1^{-1}, \ldots, \theta_k^{-1}) \). The isomorphisms in \( \text{CPO}, \text{CPO}_2, \text{ACC}, \text{ACC}_2 \) are the bijections such that it and its inverse are monotonic. In \( \text{CPO}_2 \) and \( \text{CPO}_2 \) isomorphisms are pairs of such functions.

For recursive domain equations we consider only systems of one equation. Systems of several equations can be solved too but are not needed for the semantics of the metalanguage. (The effect of several equations can be achieved by solving several parameterised equations /LeSm81/ so little generality is lost.) A domain equation like

\[ X = T + (X \times N) \]

corresponding to

\[ \text{rec } X. T + (X \times N) \]

in the metalanguage gives rise to a covariant functor

\[ + \cdot (K_T, X \cdot (\text{Id}_N, K_N)) \]
over \( \text{CPO}_2 \). To cater for occurrences of function space we shall use

\[
+^S \cdot (K_n, x^{\cdot}(\text{Id}, K_n))
\]

over \( \text{CPO}_2 \) instead. The analogue of a **fixed point** of a function is captured by:

**Definition** /SmPl82, PlLoLN/. The pair \((X, \theta)\) is a fixed point of a covariant functor \(G\) over some category iff \(X\) is an object and \(\theta: G(X) \to X\) is an isomorphism. ///

If \(F\) is a covariant functor over \( \text{CPO}_2 \) the pair \((X, \theta)\) is a fixed point of \(F\) iff \((X, (\theta, \theta^{-1}))\) is a fixed point of \(F^S\). Equality, i.e. \(\theta = \text{id}_X\), is usually not considered because the axiom of regularity of \(2\text{F}\) set theory shows that \(X \neq X \circ X\) cannot hold when \(X \neq \emptyset\) /Ham82/.

The notion of an embedding is essential to explain the analogue of least fixed points. An **embedding** in a cpo-category \(B\) is a morphism \(e: A \to B\) for which there exists another morphism \(e^U: B \to A\) such that \(e^U e = \text{id}_A\) and \(e' e^U \leq \text{id}_B\). The morphism \(e^U\) is the upper adjoint of \(e\) and is unique if it exists. Every isomorphism \(\theta\) is an embedding and \(\theta^U = \theta^{-1}\). The composition of embeddings gives an embedding and \((e_1 \cdot e_2)^U = e_2^U \cdot e_1^U\). Therefore one obtains a subcategory \(B^e\) of \(B\) by restricting the morphisms to be the embeddings of \(B\). The categories \(\text{CPO}_e\) and \(\text{CPO}^e_s\) are the same because an embedding of \(\text{CPO}\) is completely additive (hence strict) with a strict upper adjoint. An embedding in \(\text{CPO}_2^e\) is a pair \((e_1, e_2)\) where \(e_1\) and \(e_2\) are embeddings of \(\text{CPO}_2\) and the upper adjoint is \((e_1^U, e_2^U)\). A locally monotonic covariant functor specialises to a functor upon the subcategories of embeddings because if \(e_1, \ldots, e_k\) are embeddings then so is \(F(e_1, \ldots, e_k)\) with upper adjoint \(F(e_1^U, \ldots, e_k^U)\). Finally, the formula \(e_1 \leq e_2\) iff \(e_1^U \leq e_2^U\).
holds for embeddings $e_1$ and $e_2$.

The existence of an embedding $e:A \to B$ may be viewed as an analogue of $a \preceq b$ for elements of some partially ordered set. Consult /PloLN/ for this kind of motivation. Using the categorical concept initiality the analogue of a least fixed point is captured by the following definition that is explained afterwards.

**Definition** The pair $(X, \theta)$ is an initial fixed point of a covariant functor $G$ over some cpo-category iff it is a fixed point and for every fixed point $(X', \theta')$ there exists precisely one embedding $e:X \to X'$ such that $e'\theta = \theta'\cdot G(e)$.

If $F$ is a covariant functor over $\text{CPoCat}$ the pair $(X, \theta)$ is an initial fixed point of $F$ iff $(X, (\theta, \theta^{-1}))$ is an initial fixed point of $F^\cdot$.

The equation $e'\theta = \theta'\cdot G(e)$ may be formulated as the requirement that the diagram

\[
\begin{array}{ccc}
G(X) & \xrightarrow{\theta} & X \\
\downarrow G(e) & & \downarrow e \\
G(X') & \xrightarrow{\theta'} & X'
\end{array}
\]

commutes. Intuitively, the diagram requires the embedding $e$ of interest to identify elements of $X$ with elements of $X'$ in a way consistent with how $G$ was built. (It is essential for the diagram to make sense that $G$ is covariant.) The existence of precisely one embedding means that $X$ has the "right" collection of elements. Even when initial fixed points exist they need not be unique. However, any two initial fixed points $(X, \theta)$ and $(X', \theta')$ are isomorphic in the sense that there is an isomorphism $\psi:X \to X'$ such that $\psi'\theta = \theta'\cdot G(\psi)$. For the proof construct $\psi$ as the unique embedding from $X$ to $X'$ as
detailed in the definition. Similarly \( \varphi^{-1} \) is constructed from \( X' \) to \( X \). That \( \varphi^{-1} \cdot \varphi = \text{id}_X \) follows because both \( \varphi^{-1} \cdot \varphi \) and \( \text{id}_X \) are embeddings that can be used from \( X \) to \( X \) as detailed in the definition. Similarly \( \varphi \cdot \varphi^{-1} = \text{id}_X \), and it follows that \( \varphi \) is an isomorphism.

Solving in \( \text{CPO}_e \)

Let \( G \) be a locally monotonic symmetric functor over \( \text{CPO}_e \) corresponding to some domain equation that is to be solved. We have already seen that \( G \) specialises to a covariant functor over \( \text{CPO}_e \) and a covariant functor \( G^E: \text{CPO}_e \to \text{CPO}_e \) is obtained by

\[
G^E(A) = G(A)
\]
\[
G^E(e) = G((e,e^U)^U)
\]

This setting also includes a locally monotonic functor \( F \) over \( \text{CPO}_e \) as then \( G = F^S \) and \( G^E \) is the specialisation of \( F \) to \( \text{CPO}_e \). It is convenient to begin with considering an initial fixed point for \( G^E \). Later it will be transformed to \( \text{CPO}_e \) and certain sub-categories of \( \text{CPO}_e \).

Corresponding to a chain of elements of a cpo one may define a notion of a chain in a category. It is a pair \( E = ((D_n), (e_n)) \) where each \( D_n \) is an object and each \( e_n : D_n \to D_{n+1} \) is a morphism. A chain in \( \text{CPO}_e \) therefore is a pair \( ((D_n), (e_n)) \) where each \( D_n \) is a cpo and each \( e_n \) is an embedding of \( \text{CPO}_e \). The analogue of an upper bound for the chain \( E \) is that of a cone \( (\text{PloLN}) \). (In strict categorical language the prefix co- should be used.) A cone is a pair \( R = (D, (r_n)) \) where \( D \) is an object and each \( r_n : D_n \to D \) is a morphism such that \( r_{n+1} \cdot e_n = r_n \). This may be illustrated by the
and the requirement that all small triangles (hence all triangles) commute.

The analogue of a least upper bound is called a limiting cone /PloLN/ and builds on the categorical idea of initiality. The cone \( R \) is limiting (or initial) iff for every cone \( R' = (D', (r'_n)_n) \) there is precisely one morphism \( r:D \to D' \) such that \( r'r_m = r'_m \). The morphism \( r \) is said to mediate from \( R \) to \( R' \) /SmPl82/. This may be illustrated by the diagram

![Diagram](image)

and the requirement that all such triangles must commute. If a chain has a limiting cone it may well have more than one. If \( R \) and \( R' \) are limiting cones they are isomorphic in the sense that there is an isomorphism \( \theta:D \to D' \) such that \( \theta'r_n = r'_n \). To see this construct \( \theta \) as the mediating morphism from \( R \) to \( R' \) and \( \theta^{-1} \) as the mediating morphism from \( R' \) to \( R \). Since both \( \theta^{-1}, \theta \) and \( \text{id}_D \) mediate from \( R \) to \( R \) it follows that \( \theta^{-1} \theta = \text{id}_D \) and \( \theta \theta^{-1} = \text{id}_D \), is shown similarly.

In \( \mathbb{CPOe} \) a chain always has a limiting cone and it is easy to give a non-categorical criterion for when a cone is limiting.
Theorem 2.2: (essentially /SmPl82/).

Let $E = ((D_n), (e_n),)$ be a chain in CPOe and $R = (D, (r_n))$ a cone for $E$. Then

1. $R$ is a limiting cone iff $id_D = \bigvee_n r_n^* r_n^U$

2. $E$ always has a limiting cone

3. If $R$ is limiting and $R'$ is a cone then the mediating morphism from $R$ to $R'$ is $r = \bigvee_n r_n^* r_n^U$ with upper adjoint $\bigvee_n r_n^* r_n^{U'}$. //

Sketch of proof. First note that for any two cones $R$ and $R'$ the sequence $(r_n^* r_n^U)$ is a chain and therefore $\bigvee_n r_n^* r_n^U$ is well-defined as a function. Next let $R$ be a cone such that $id_D = \bigvee_n r_n^* r_n^U$. Then $r = \bigvee_n r_n^* r_n^U$ is an embedding with $\bigvee_n r_n^* r_n^{U'}$ as its upper adjoint.

It is immediate that $r$ mediates from $R$ to $R'$. If also $r'$ mediates from $R$ to $R'$ we have

$$r' = r^* id_D = \bigvee_n r_n^* r_n^U = \bigvee_n r_n^* r_n^{U'} = r$$

This shows "if" in (1) and (3) in a special case.

To show (2) we construct a cone $R$ such that $id_D = \bigvee_n r_n^* r_n^U$.

Let

$$D = \{ (d_0, d_1, \ldots) \mid d_n \in D_n \land d = e_n^U(d_{n+1}) \}$$

When partially ordered componentwise this gives a cpo. Further let

$$r_n^U(d_n) = (\ldots, e_{n-1}^U(d_n), d_n, e_n(d_n), \ldots)$$

and note that $r_n^U ((d_0, d_1, \ldots)) = d_n$. Since $\bigvee_n r_n^* r_n^U = id$ it follows that $R$ is limiting. It remains to show "only if" in (1) and (3) in the general case. This may be shown using the "isomorphism" between a given limiting cone $R'$ and the $R$ constructed above. //
It is convenient to define the effect of $G^E$ upon chains and cones. For a chain $E = ((D_n)_n, (e_n)_n)$ a chain $G^E[E]$ is defined by

$$G^E[E] = ((G^E(D_n))_n, (G^E(e_n))_n)$$

and for a cone $R = (D, (r_n)_n)$ a cone $G^E[R]$ is defined by

$$G^E[R] = (G^E(D), (G^E(r_n))_n)$$

and it is a cone for $G^E[E]$. To preserve limits we shall assume $G$ to be locally continuous:

**Lemma 2.2:** Let $G$ be a locally continuous symmetric functor and $E$ a chain in $\text{CPO}_e$ with limiting cone $R = (D, (r_n)_n)$. Then $G^E[R]$ is a limiting cone for $G^E[E]$ and if $r$ mediates from $R$ to a cone $R' = (D', (r'_n)_n)$ then the mediating morphism from $G^E[R]$ to $G^E[R']$ is $G^E(r)$.

This result is a straightforward consequence of the previous theorem and may be formulated as

$$G^E(\bigcup_n r_n \cdot r'_n \cdot U) = \bigcup_n G^E(r'_n) \cdot G^E(r_n) \cdot U$$

assuming that $\bigcup_n r_n \cdot r'_n \cdot U = \text{id}$.

For the least fixed point of a continuous function $f$ the chain $(f^n(\bot))_n$ was of interest. For a locally continuous and symmetric functor $G$ the analogue is

$$\text{CHAIN}(G) = ((D_n)_n, (e_n)_n)$$

where $D_0 = U$ (the one-element cpo), $D_{n+1} = G^E(D_n)$, $e_0 = \bot$ and $e_{n+1} = G^E(e_n)$. This gives a chain in $\text{CPO}_e$.

**Theorem 2.2:** Let $G: \text{CPO}_e \to \text{CPO}_e$ be a locally continuous and symmetric functor.
Let \((D, (r_n)_n)\) be a limiting cone in \(\mathbb{CPO}_\mathbb{E}\) of \(\textit{CHAIN}(G)\). Then \((D, \theta)\) with \(\theta = \bigsqcup_{n+1} r_n^* G^E (r_n)^U\) is an initial fixed point of \(G^E : \mathbb{CPO}_\mathbb{E} \rightarrow \mathbb{CPO}_\mathbb{E}\).

**Sketch of proof.** We shall write \(\textit{CHAIN}(G) = \{(D_n)_n, (e_n)_n\}\). The chain \(((D_n+1)_n, (e_n+1)_n)\) has \((D, (r_{n+1})_n)\) as a limiting cone but since the chain equals \(G^E[\textit{CHAIN}(G)]\) it follows from the previous lemma that also \((G^E(D), (G^E(r_n))_n)\) is a limiting cone. Hence the mediating embedding \(\Theta : G^E(D) \rightarrow D\) is an isomorphism. It follows that \((D, \Theta)\) is a fixed point for \(G^E\) and by theorem 2.2:2 the formula for \(\Theta\) follows.

Next let \((D', \Theta')\) be a fixed point and suppose the embedding \(e : D \rightarrow D'\) satisfies \(e^* \Theta = \Theta' \cdot G^E(e)\). Then \(e^* r_0 = \bot\) and \(e^* r_{n+1} = e^* \Theta^* G^E(r_n) = \Theta' \cdot G^E(e^* r_n)\) shows that \(e = e^* \text{id}_D = \bigsqcup_n (e^* r_n)^* r_n^U\) is uniquely determined. Next a cone \(((D', (r'_n)_n)\) for \(\textit{CHAIN}(G)\) may be defined by \(r'_0 = \bot\) and \(r'_{n+1} = \Theta' \cdot G^E(r'_n)\). Let \(r : D \rightarrow D'\) be the mediating morphism from the limiting cone to this cone. Then

\[
\begin{align*}
r^* \Theta &= \bigsqcup_n r^* r_{n+1}^* G^E(r_n)^U \\
&= \bigsqcup_n r'_n^* G^E(r'_n)^U \\
&= \Theta' \cdot \bigsqcup_n G^E(r'_n)^* G^E(r_n)^U \\
&= \Theta' \cdot G^E(r)
\end{align*}
\]

follows by (the discussion after) the previous lemma.

Note that local continuity was only required in order to use the previous lemma.

In general there are many limiting cones and initial fixed points, although they are all "isomorphic". When talking about the limiting cone or the initial fixed point we refer to the constructions of 2.2:2 and 2.2:4.
Viewing the solution in other categories

We now move the solutions from $\text{CPO}_e$ to $\text{CPO}_2s$ and certain sub-categories of $\text{CPO}_e$. We begin with chains, cones and limiting cones.

So let $E = \langle (D_n), (e_n) \rangle$ be a chain in $\text{CPO}_e$. It is already a chain in $\text{CPO}_e$ and a chain $S(E)$ in $\text{CPO}_2s$ may be defined by

$$S(E) = \langle (D_n), ((e_n, e_n^U)) \rangle$$

Next let $R = \langle (D), (r_n) \rangle$ be a cone of $E$ in $\text{CPO}_e$. It is already a cone of $E$ in $\text{CPO}_e$ and a cone $S(R)$ in $\text{CPO}_2s$ may be defined by

$$S(R) = \langle (D), ((r_n, r_n^U)) \rangle$$

It is convenient to write $e_n = (e_n, e_n^U)$ and $r_n = (r_n, r_n^U)$. Note that $e_n^U = e_n$ and $r_n^U = r_n$.

As analogues of theorem 2.2:2 we have:

**Lemma 2.2:5.** Let $E = \langle (D_n), (e_n) \rangle$ be a chain in $\text{CPO}_2s$ with embeddings such that $e_n^U = e_n$. It always has a limiting cone in $\text{CPO}_2s$. A cone $R = \langle (D), (r_n) \rangle$ is limiting iff $r_n$ are embeddings and $\bigcup_n r_n^U = \text{id}$. Then the mediating morphism from $R$ to $\langle (D'), (r_n') \rangle$ is $\bigcup_n r_n^U$. It is possible to choose $R$ such that $r_n^U = r_n$. ///

**Lemma 2.2:6.** Let $E = \langle (D_n), (e_n) \rangle$ be a chain in $\text{CPO}_e$ with embeddings. It always has a limiting cone in $\text{CPO}_e$. A cone $R = \langle (D), (r_n) \rangle$ is limiting iff $r_n$ are embeddings and $\bigcup_n r_n^U = \text{id}$. Then the mediating morphism from $R$ to $\langle (D'), (r_n') \rangle$ is $\bigcup_n r_n^U$. ///

The proofs are similar to that of 2.2:2 and are omitted. It follows that if $R$ is a limiting cone for $E$ in $\text{CPO}_e$ then $R$ is a limiting cone for $E$ in $\text{CPO}_e$ and $S(R)$ is a limiting cone for $S(E)$ in $\text{CPO}_2s$. When talking about the limiting cone for $S(E)$ we mean $S(R)$ where $R$ is
the limiting cone for \( E \). This means that an embedding \( r_n \) of the limiting cone for \( S(E) \) has \( r_n^R \) as its upper adjoint.

When giving the semantics of the bottom-level metalanguage it is convenient to be able to use other categories than \( \mathbb{CPO} \). This motivates defining a notion of admissible subcategory. A sub cpo-category \( \mathbb{B} \) of a cpo-category \( \mathbb{C} \) is a subcategory such that

i) the morphisms are partially ordered as in \( \mathbb{C} \),

ii) the sets of morphisms of \( \mathbb{B} \) may be viewed as admissible predicates upon the corresponding sets of morphisms in \( \mathbb{C} \).

Clearly a sub cpo-category is itself a cpo-category. An example is \( \mathbb{ACC} \) that is a sub cpo-category of \( \mathbb{CPO} \). When \( \mathbb{B} \) is a sub cpo-category of \( \mathbb{C} \) every chain in \( \mathbb{B}_e \) (the subcategory of embeddings) is a chain in \( \mathbb{C}_e \) as well. An admissible subcategory \( \mathbb{B} \) of a cpo-category \( \mathbb{C} \) is a sub cpo-category that

i) contains \( U \),

ii) for every chain in \( \mathbb{B}_e \) and limiting cone \( R \) in \( \mathbb{C}_e \) that \( R \) is a cone in \( \mathbb{B}_e \).

(It will not do if \( R \) is only in \( \mathbb{B} \).) A further analogue of theorem 2.2:2 then is:

**Lemma 2.2:7.** Let \( \mathbb{B} \) be an admissible subcategory of \( \mathbb{CPO}_e \) and \( E \) a chain in \( \mathbb{B}_e \). The limiting cone calculated in \( \mathbb{CPO}_e \) (by theorem 2.2:2) is also limiting in \( \mathbb{B} \) and the mediating morphism is as in \( \mathbb{CPO}_e \).

**Proof** The mediating morphism from \( \mathbb{CPO}_e \) is in \( \mathbb{B} \) because \( \mathbb{B} \) is a sub cpo-category of \( \mathbb{CPO}_e \). This proves existence and uniqueness is because a cone in \( \mathbb{B} \) is also a cone in \( \mathbb{CPO}_e \).
Example The category ACC\text{\#} is an admissible subcategory of CPO\text{\#}.

To see this let \((D,(r_n))\) be a limiting cone in CPO\text{\#} for an ACC\text{\#} chain \(\left((D_n)_n,(e_n)_n\right)\). We first show that \(D\) is algebraic with

\[B_D = \{r_n(b_n) \mid b \in B_{D_n} \wedge n \geq 0\}\]

and fact 2.2:1 will be used for this. If \(r\) is an embedding and \(b\) is finite then also \(r(b)\) is finite: for \(r(b) \leq \bigsqcup_{n \geq 0} r_n(d_n)\) so there exists an \(n\) such that \(b \leq r_n(d_n)\) and \(r(b) \leq d_n\) follows. Next if \(d \in D\) we have \(d = \bigsqcup_n r_n(r_n^{-1}(d))\) and for each \(n\) there is a chain \((b_n^n)\) of elements of \(B_{D_n}\) such that \(r_n(d) = \bigsqcup_m b_{m,n}\). It follows that

\[d = \bigsqcup \{r_n(b_{m,n}) \mid n \geq 0, m \geq 0\}\]

To see \(D\) is consistently complete let \(H\) be a subset and \(d\) an upper bound. Then \(\{r_n^{-1}(h) \mid h \in H\}\) has \(r_n^{-1}(d)\) as an upper bound so

\[d' = \bigsqcup \{r_n^{-1}(h) \mid h \in H\}\]

exists. Also \((r_n(d'_n))_n\) is an increasing chain so \(\bigsqcup_n r_n(d'_n)\) exists. It is the least upper bound of \(H\) in \(D\). ///

We now move the fixed points across to CPO\text{\#} and admissible subcategories of CPO\text{\#}. Since CPO\text{\#} is an admissible subcategory of CPO\text{\#} this includes CPO\text{\#} as well. A generalisation of fixed points leads to:

Definition An algebra of a covariant functor \(G\) upon some category is a pair \((D,g)\) where \(D\) is an object and \(g : G(D) \rightarrow D\) is a morphism. The algebra is initial iff for any algebra \((D',g')\) there is precisely one morphism \(g'' : D \rightarrow D'\) such that \(g''g = g'g(g'')\). ///

Clearly every fixed point is an algebra.

Theorem 2.2:8 (essentially /PloLN, SmP182/).

(1) Let \(G\) be a locally continuous and symmetric functor over CPO\text{\#}
with \((D, \theta)\) the initial fixed point of \(G^E\). Then \((D, (\theta, \theta^{-1}))\) is an initial algebra of \(G\) and an initial fixed point of \(G\) (so that the mediating morphism to another fixed point is an embedding).

(2) Let \(B\) be an admissible subcategory of \(\text{CPOs}\) and \(F\) a locally continuous and covariant functor over \(B\). Then \(F\) has an initial algebra that is also an initial fixed point and it may be constructed as in theorem 2.2:4.

The proof is analogous to that of 2.2:4 and is omitted.

Extending the solution to a functor

The metalanguage allows to nest recursive domain equations. The theory developed so far transforms a domain equation like \(\text{rec}X_1, X_1 + \ldots + X_N + A\) to a functor \(G: \text{CPO}^N \rightarrow \text{CPO}^N\) in order to solve the equation. It is therefore necessary to formulate the solution as a functor.

We begin with considering a locally continuous and symmetric functor \(G: \text{CPO}^N \rightarrow \text{CPO}^N\). The aim is to define a similar functor \(\text{REC}(G): \text{CPO}^{N-1} \rightarrow \text{CPO}^N\). We shall subsequently assume \(i=1\) and omit the subscript. It is straightforward to define the effect of \(\text{REC}(G)\) upon objects \(B_2, \ldots, B_N\). Define \(H = G^*(\text{Id}, K_{B_2}, \ldots, K_{B_N})\) and note that it is a locally continuous and symmetric functor over \(\text{CPO}^N\). It therefore has the initial fixed point \((D, \theta)\) and we define \(\text{REC}(G)(B_2, \ldots, B_N) = D\).

To define the effect upon morphisms we consider the chains involved in constructing \((D, \theta)\). The chain \(E = S(\text{CHAIN}(H))\) will be written \(\left( (H^n(U))_n, (H^n(L))_n \right)\) and let the limiting cone \(R\) in \(\text{CPO}^N\)
be \((D, (r_n^n))\). We then have \(\Theta = \bigcup_n r_n^n \cdot \Pi^n(U)\). Let \(f_i : B_i \rightarrow B'_i\) be morphisms and define \(H' = G'(\text{Id}, K_{B'_2}, \ldots, K_{B'_N}), E', R'\) and \((D', \Theta')\) similarly to above. To obtain a morphism from \(D\) to \(D'\) we will modify \(R'\) to a cone \(Rg'\) of \(U'\).

For this purpose define a grid from \(E\) to \(E'\) as a sequence \((q_n^n)\)
where each \(q_n : \Pi^n(U) \rightarrow \Pi^n(U)\) is a \(\text{CPO2s}\) morphism such that
\[q_{n+1} \cdot \Pi^n(U) = \Pi^n(U) \cdot q_n.\]
This may be illustrated by

\[
\begin{array}{c}
\begin{array}{ccc}
U & \xrightarrow{f} & \Pi(U) \\
\downarrow{g_0} & & \downarrow{g_1} \\
U & \xrightarrow{f} & \Pi'(U)
\end{array} & \begin{array}{ccc}
\Pi(U) & \xrightarrow{f} & \Pi^2(U) \\
\downarrow{g_1} & & \downarrow{g_2} \\
\Pi'(U) & \xrightarrow{f} & \Pi^2(U)
\end{array}
\end{array}
\]

and the information that all small (hence all) rectangles commute.

Now define \(g_0 = \bot\) and \(g_{n+1} = G(q_n^n, f_2, \ldots, f_N^n)\). To see this defines a grid note that the leftmost square commutes (because we are in \(\text{CPO2s}\)) and since \(G\) is covariant this means that all others do as well. Now define \(Rg'\) as \((D', (r'_{n}g_{n})_{n})\) and note this is a cone on \(E\)
because \((q_n^n)\) is a grid from \(E\) to \(E'\) and \(R'\) is a cone on \(E'\). By 2.2:5 there is precisely one mediating morphism \(r\) from \(R\) to \(Rg'\).

It is \(r = \bigcup_n r_{n} \cdot g_{n} \cdot r_{n} U\) (and \((r_{n} \cdot g_{n} \cdot r_{n} U)_{n}\) is a chain) and we define
\(\text{REC}(G) = r\).

We then have:

Theorem 2.2:9. If \(G\) is a locally continuous and symmetric functor over \(\text{CPO2s}\) then so is \(\text{REC}(G)\).

Proof The identity functor law that \(\text{REC}(G)(\text{id}, \ldots, \text{id}) = \text{id}\) is straightforward because each \(q_n\) is \(\text{id}_{U^n(U)}\) and \(\text{id} = \bigcup_n r_{n} \cdot r_{n} U\)
follows by 2.2:5. For the composition functor law let \(f_i : B_i \rightarrow B'_i\) and define \(H'', E'', R''\) as before. Let \((q_n^n)\) be the grid from \(E\) to \(E'\),
The grid from $E'$ to $E''$ and $(g_n')_n$ the grid from $E$ to $E''$. It is a straightforward numerical induction to show that $g_n' = g_n''$. Then

$$REC(G)\left( f_1', f_2', \ldots, f_N' \right) =$$

$$\bigcup_n r_n^U g_n' r_n =$$

$$\bigcup_n (r_n^U g_n' r_n') (r_n' g_n' r_n') =$$

$$REC(G)\left( f_1', f_2', \ldots, f_N' \right) \cdot \theta^{-1} \cdot \theta^{-1}$$

shows the result. To see that $REC(G)$ is symmetric note that the grid from $E'$ to $E$ constructed for $f_2^R, \ldots, f_N^R$ is $(g_n^R)_n$. Then

$$REC(G)\left( f_2^R, \ldots, f_N^R \right) =$$

$$\bigcup_n (r_n^R g_n^R r_n^R) =$$

$$REC(G)\left( f_2^R, \ldots, f_N^R \right)$$

follows because the cones $R$ and $R'$ have $r_n^U = r_n^R$ and $r_n^U = r_n^R$. Local continuity is immediate because the grid (i.e. each $g_n$) depends continuously on $f_2', \ldots, f_N'$. //

For later reference we state some lemmas about $REC(G)$. The first amounts to another way of defining the effect upon morphisms.

**Lemma 2.2:** $REC(G)\left( f_1', f_2', \ldots, f_N' \right)$ is the least fixed point of the continuous function mapping a CPO2s morphism $f$ to the morphism $\theta^{-1} \cdot G(f, f_1', f_2', \ldots, f_N') \cdot \theta^{-1}$. //

**Proof** Let $G'$ denote the continuous function defined. It suffices to prove $G'^n(\lambda) = r_n^U g_n r_n$ by induction on $n$. The base case is immediate and

$$G'^{n+1}(\lambda) = \theta^{-1} \cdot G(r_n^U g_n r_n, f_2', \ldots, f_N') \cdot \theta^{-1}$$

$$= \theta^{-1} \cdot H'(r_n^U) \cdot g_{n+1} \cdot H(r_n^U) \cdot \theta^{-1}$$

$$= r_{n+1}^U g_{n+1} r_{n+1}$$

shows the inductive step. //
Lemma 2.2: 11. \( r_{n}^{*}g_{n} = REC(G)(f_{2}, \ldots, f_{N})r_{n} \) and \( g_{n}^{*}r_{n} = r_{n}^{*}REC(G)(f_{2}, \ldots, f_{N}) \)

Proof The first result follows from the definition of \( REC(G) \) and lemma 2.2:5. The second result follows from \( g_{n}^{*}H^{n}(\omega)U = H^{n}(\omega)U^{*}g_{n+1} \) using that \( r_{n}^{U}r_{n}^{*} = H^{n}(\omega)U \).

It may help to summarise the last results by the following diagram

\[
\begin{align*}
H^{n}(U) & \xrightarrow{r_{n}} H^{n}(\omega) & \xrightarrow{g_{n}} H^{n}(U) \\
H^{n+1}(U) & \xrightarrow{r_{n+1}} H^{n+1}(\omega) & \xrightarrow{g_{n+1}} H^{n+1}(U) \\
H(D) & \xrightarrow{H(r_{n})} H(D) & \xrightarrow{H(r_{n+1})} H(D') \\
\end{align*}
\]

where \( r = REC(G)(f_{2}, \ldots, f_{N}) \) and all polygons commute.

The development so far may be specialised to apply for locally continuous and covariant functors upon admissible subcategories of CPOs. So let \( B \) be an admissible subcategory of CPOs and \( F \) a locally continuous and covariant functor over \( B \). Define \( H, H', E, E', R, R', (D, \theta), (D', \theta') \) and \( (g_{n})_{n} \) much as before and define

\[
\begin{align*}
REC(F)(B_{2}, \ldots, B_{N}) = D \\
REC(F)(f_{2}, \ldots, f_{N}) = \bigcup_{n} r_{n}^{*}g_{n}^{*}r_{n}^{U} 
\end{align*}
\]

Lemma 2.2:12. If \( F \) is locally continuous and covariant over an admissible subcategory of CPOs then also \( REC(F) \) is.

Analogues of lemmas 2.2:10 and 2.2:11 also hold in this setting.

Remark Suppose \( F: CPOs^{2} \rightarrow CPOs \) is a functor that is covariant in its left argument and contravariant in its right argument. Instead
of working with $F^S$ over $\text{CPO}_2$ one could directly define $F^{SE}$ over $\text{CPO_e}$ by $F^{SE}(f,g) = F(f,g^U)$. This is what is done in /SmPl82/. Instead of working with local continuity one may use a categorical notion of continuity for covariant functors over $\text{CPO_e}$. (Essentially 2.2:3 shows that if $F$ is locally continuous then $F^{SE}$ is continuous in the categorical sense.) This is done in /LeSm91/ where proofs have a distinct "categorical flavour".

2.3 SEMANTICS OF THE METALANGUAGE

Using the domain theory of section 2.2 we can now define the semantics of the metalanguage from section 2.1. As was sketched in the introduction (for basic blocks of flowcharts) it is of interest to assign several different meanings to the formulae of the metalanguage. To facilitate this we parameterise the semantics upon a specification of the details that may vary. Such a parameter will be called an interpretation. The idea is borrowed from algebra and logic and in data flow analysis similar ideas have motivated the monotone frameworks of /KaUl77/ and the interpretations of /CoCo77b/.

An interpretation consists of two parts: the type part and the expression part. We shall consider the type part before the expression part. For each of the two parts we begin with a formal definition. The definition is illustrated by defining that part of the standard interpretation, i.e. the interpretation that gives rise to the standard semantics /MiSt76/. Finally, the semantic equations for the corresponding part of the metalanguage is given.
**TYPE PART**

The type part of an interpretation $I$ specifies information needed to define the functors $I[t]$ and $I[gt]$ that express the semantics of the types. A top-level type $t$ will always be interpreted over $\text{CPO}_2$, i.e. $I[t]$ will be a locally continuous and symmetric functor over $\text{CPO}_2$. The need for morphisms to be pairs of functions has already been motivated (with contravariance of function space) as has the need for strictness (for $+$ and $\times$). A subcategory of $\text{CPO}$ (such as $\text{ACC}$) could be used but no complications arise from the use of $\text{CPO}$ and the larger category is therefore preferred. Domain constructors in the top-level will be as in section 2.2.

The bottom-level metalanguage will vary considerably in the way it is interpreted, so most constructs will be defined in the interpretation. There is no function space construction in the bottom-level metalanguage so $I[gt]$ may be interpreted as a locally continuous and covariant functor upon $\text{CPO}_2$ (i.e. no need for $\text{CPO}_2$).

The bottom-level type structure will be interpreted in different settings so it is helpful to allow admissible subcategories of $\text{CPO}_2$. Domain constructors are then defined as functors over that subcategory.

Formally the type part of an interpretation $I$ is a tuple with the following three components:

- a sub cpo-category $\mathbb{B}$ of $\text{CPO}$ and a (necessarily admissible) predicate $p$ upon each set of morphisms of $\mathbb{B}$ such that $\mathbb{B}_p$ (having as morphisms those of $\mathbb{B}$ that satisfy $p$) is an admissible subcategory of $\text{CPO}_2$.
for each of the domain constructors $X, Y, \oplus, \bot$ a locally continuous and covariant functor $\mathbb{B}^k \to \mathbb{B}^k$,

for each bottom-level domain $A_1$ a constant functor $\mathbb{B}^0 \to \mathbb{B}^0$.

We write $I(B), I(\bot), I(A_1)$ etc. for these entities. We will assume throughout that the meaning of domains in the top-level metalanguage are fixed and therefore it has not been made part of "an interpretation".

The standard interpretation $S$ has $S(B) = \{\text{ACC}\}$ and $S(B^\bot) = \{\text{ACC}\}$. It is helpful later (when constructing the powerdomain and the tensor product) that objects are algebraic. The assumption about consistently completeness is not necessary but would be needed if algebraicity had been assumed in the top-level type structure. The functors $S(X), S(Y), S(\oplus)$, and $S(\bot)$ are as in section 2.2 but specialised to $\text{ACC}$. Finally, $S(A_1) : \mathbb{B}^0 \to \mathbb{B}^0$ will not be specified but one may assume they give the same objects as is given by $A_1$ in the top-level.

For semantics more precision is needed about the connection between a domain variable $X$ of $t$ and the corresponding argument position for $I[t]$. We assume there is a bijective correspondence between the domain variables and the natural numbers. A domain variable $X$ is said to have index $i$ in the finite set $V$ of domain variables iff exactly $i-1$ domain variables from $V$ have an associated number less than that of $X$. Let $\text{card}(V)$ denote the cardinality of $V$. If $V \vdash t$ we define a functor $I[t]_V : \mathbb{CPO2}^{\text{card}(V)} \to \mathbb{CPO2}$ with $I[t]_V = P_i$ (the $i$'th projection functor). Similarly $V \vdash gt$ gives $I[gt]_V : \mathbb{B}^{\text{card}(V)} \to \mathbb{B}$. The use of index $V$ could be alleviated if it was always assumed that it was the least set $V'$ such that $V' \vdash t$. 71
However, we shall mostly omit the subscript $V$ and assume that the index is $i=1$. 

For the details of the semantics of the top-level types:

\[ \text{IFT}_1 \cdots \text{IFT}_k = \text{T} s \cdot (\text{IFT}_1, \ldots, \text{IFT}_k) \]

for $x: \text{CPO}_s \to \text{CPO}_s$ defined in section 2.2; it is made symmetric and composed with the arguments

\[ \text{IFT}_1 \cdots \text{IFT}_k = \text{T} s \cdot (\text{IFT}_1, \ldots, \text{IFT}_k) \]

where $\to$ is the function space construction of section 2.2

\[ \text{IFT}_1 \to \text{IFT}_2 = \text{T} s \cdot (\text{IFT}_1, \text{IFT}_2) \]

The last equation requires some explanation. Here $\to$ is a mixed covariant and contravariant functor $\text{B}_0 \to \text{CPO}_s$ giving on objects the cpo of $B$ morphisms from one to another and is defined in section 2.2 on morphisms. So $(\text{IFT}_1, \text{IFT}_2)$ is a covariant functor $\text{B}_0 \to \text{CPO}_s$. The category $\text{B}_0$ has one object NIL and composition with the constant functor $K_{\text{NIL}}: \text{CPO}_s \to \text{B}_0$ gives a functor.
$\text{CPO}_\text{card}(V) \rightarrow \text{CPO}_\text{card}$ which is then made symmetric.

For the details of the semantics of the bottom-level types:

$$I_{\text{A}_1}^{\text{V}} = I(A_1) \cdot K_{\text{NIL}}$$
where $K_{\text{NIL}}: \text{B}_\text{D} \rightarrow \text{B}_\text{D}$

$$I^{\text{gt}_1 \ldots \text{gt}_k} = I(X^i) \cdot (I^{\text{gt}_1} V \ldots, I^{\text{gt}_k} V)$$

$$I^{\text{gt}_1^{+} \ldots + \text{gt}_k} = I(+) \cdot (I^{\text{gt}_1} V \ldots, I^{\text{gt}_k} V)$$

$$I^{\text{gt}_1 \ldots \text{gt}_k} V = I(A) \cdot I^{\text{gt}} V$$

$$I^{\text{recX}, \text{gt}} V = \text{REC}_i (I^{\text{gt} V\{X\}}) \cdot (P_1, \ldots, P_{i-1}, P_i+j, \ldots, P_N)$$

where $i$ is the index of $X$ in $V\{X\}$, $N$ is $\text{card}(V)$ and

$$j=1 \text{ if } X \notin V \text{ and otherwise } j=0$$

$$I^{\text{X} V} = P_i$$

where $i$ is the index of $X$ in $V$

These definitions satisfy:

**Lemma 2.3.1:** If $V \vdash \text{gt}$ then $I^{\text{gt} V}: \text{B}_\text{D} \rightarrow \text{B}_\text{D}$ is a locally continuous and covariant functor. If $V \vdash t$ then $I^{t V}: \text{CPO}_\text{card}(V) \rightarrow \text{CPO}_\text{card}$ is a locally continuous and symmetric functor.

The proof is straightforward by structural induction. It uses the properties of an interpretation $I$ and the results of section 2.2.

For later reference it is convenient to state a consequence of the structural definition of $I^{t V}$ and $I^{\text{gt} V}$. Recall the syntactic substitutions $\ldots [\ldots / \ldots ]$ mentioned in section 2.1.

**Lemma 2.3.2:** If $V \vdash t$, $V \vdash t'$, $\text{card}(V) = N$ and $X$ has index 1 in $V$ then for all interpretations $I$

$$I^{t[t'/X]} V = I^{t V} \cdot (I^{t' V}, P_2, \ldots, P_N)$$
and if additionally $\forall t \cdot gt$, $\forall t \cdot gt'$ then

$$\llbracket \text{gt}^{\prime}/x \rrbracket = \llbracket \text{gt} \rrbracket \cdot (\llbracket \text{gt}^{\prime} \rrbracket, p_2, \ldots, p_N)$$

**Proof** The proof is by structural induction on $gt$ and $t$ and let us restrict attention to the case $gt = \text{rec}_Y gt_0$. When $Y$ is $X$ the result is straightforward so assume otherwise. Assume that $Y$ is an element of $V$ and has index $N$ and that $Y$ is not free in $gt'$. It is straightforward to adapt the proof should any of the assumptions fail (if $Y$ is free in $gt'$ a renaming with some $Y'$ not free in $gt'$ may be performed). Abbreviate the two sides of the equation to LHS and RHS respectively.

To show that $\text{LHS}(L_1, \ldots, L_N) = \text{RHS}(L_1, \ldots, L_N)$ define functors

$$\text{LH} = \llbracket \text{gt}_0 \rrbracket \cdot (\llbracket \text{gt}' \rrbracket, p_2, \ldots, p_N) \cdot (k_{L_1}, \ldots, k_{L_N}, \text{Id})$$

$$\text{RH} = \llbracket \text{gt}_0 \rrbracket \cdot (\llbracket \text{gt}' \rrbracket) (L_1, \ldots, L_N, k_{L_1}, \ldots, k_{L_{N-1}}, \text{Id})$$

It follows from the inductive hypothesis that $\text{LHS}(L_1, \ldots, L_N)$ is the object of the limiting cone $(L, (r_n)_n)$ for the chain $((\text{LH}_n(U))_n, (\text{LH}_n(\lambda))_n)$ and similarly that $\text{RHS}(L_1, \ldots, L_N)$ is the object of the limiting cone $(\tilde{L}, (\tilde{r}_n)_n)$ for $((\text{RH}_n(U))_n, (\text{RH}_n(\lambda))_n)$. To show $L=\tilde{L}$ and $r_n=\tilde{r}_n$, it suffices to prove that $\text{LH}_n(U) = \text{RH}_n(U)$ and $\text{LH}_n(\lambda) = \text{RH}_n(\lambda)$ for all $n$. This may be proved inductively with $n=0$ obvious. For the inductive step one may calculate

$$\text{LH}^{n+1}(U) =$$

$$\llbracket \text{gt}_0 \rrbracket (\llbracket \text{gt}' \rrbracket (L_1, \ldots, L_{N-1}, \text{LH}_n(U)), L_2, \ldots, L_{N-1}, \text{LH}_n(U))$$

(because $Y$ is not free in $gt'$)

$$\text{RH}^{n+1}(U) =$$

$$\llbracket \text{gt}_0 \rrbracket (\llbracket \text{gt}' \rrbracket (L_1, \ldots, L_N), L_2, \ldots, L_{N-1}, \text{RH}_n(U))$$

and $\text{LH}_n(\lambda) = \text{RH}_n(\lambda)$ is shown similarly.
To show LHS(\(f_1, \ldots, f_N\)) = RHS(\(f_1, \ldots, f_N\)) note that we have the formula

\[
\text{LHS}(f_1, \ldots, f_N) = \bigcup_{n} r_n^* \text{L} s_n r_n^U \quad \text{when } LS_0 = \bot
\]

\[
\text{RS}_{n+1} = I gt_0 (I gt') (f_1, \ldots, f_{n-1}, LS_n, f_2, \ldots, f_{N-1}, LS_n)
\]

Similarly

\[
\text{RHS}(f_1, \ldots, f_N) = \bigcup_{n} r_n^* \text{R} s_n r_n^U \quad \text{when } RS_0 = \bot
\]

\[
\text{RS}_{n+1} = I gt_0 (I gt') (f_1, \ldots, f_{n}, f_2, \ldots, f_{N}, RS_n)
\]

It is a consequence of what was shown earlier that the embeddings \(r_n^*\) and \(r_n^*\) are the same in the two definitions. As before an inductive proof shows that \(LS_n = RS_n\) and thereby \(\text{LHS}(f_1, \ldots, f_N) = \text{RHS}(f_1, \ldots, f_N)\).

It is also convenient to state a lemma showing when two interpretations give the same semantics (use \(V' = V\) below). This furthermore shows the first use of the predicate \(P(V, t)\) defined in section 2.1.

**Lemma 2.3.3.** Let \(V \vdash t\) and \(P(V', t)\) and \(\text{card}(V) = N\) and \(\text{card}(V') = n\). Suppose \(V'\) is the subset of \(V\) of variables having index at most \(n\) in \(V\). For any two interpretations \(I\) and \(J\) and cpo's \(D_i\) and \(E_i\) the two functors (over \(\text{cpo}_2s\))

\[
I [t] = (K_{D_1}, \ldots, K_{D_n}, P_{n+1}, \ldots, P_N)
\]

and

\[
J [t] = (K_{E_1}, \ldots, K_{E_n}, P_{n+1}, \ldots, P_N)
\]

are equal.

**Proof** The proof is by structural induction and only the case recX.t is non-trivial. This case follows the same lines as the previous proof and is therefore omitted.
The type $t$ of an expression $e$ is a top-level type with no free
domain variables. More precisely there is a type environment $tenv$
with finite domain $\text{dom}(tenv)$ such that $tenv \vdash e : t$ holds and we
assumed in section 2.1 that $t$ and all $tenv(x)$ were closed. The
semantics of the type therefore is a constant functor and it is the
object that is of interest. This motivates writing $I[t]$ instead of
$I[t]_0(\text{NIL})$. With this notation the semantics of $e$ is a function

$$I[e]_{tenv} : \left( \prod_{x \in \text{dom}(tenv)} I[tenv(x)] \right) \to I[t]$$

The function is continuous but since all morphisms encountered in
chapters 2 to 4 will be continuous we shall often let the statement
of continuity be implicit. Also we shall shortly omit the subscript
$tenv$ when confusion is not likely to occur.

The top-level aspects of the metalanguage will always be the same.
The bottom-level aspects are defined in the expression part of an
interpretation. Formally it is a tuple with three components:

- for each of the symbols take$_i$, smashtake$_i$, in$_i$, up, fold and
  unfold some functions must be specified. Taking take$_i$ as an
  example it is assumed that $tenv \vdash \text{take}_i : ft$ and the function
  must be an element of $I[ft]$ (and is hence continuous). This
  is slightly imprecise because take$_i$ has not been indexed
  with its type $ft$.

- for each of the symbols tuple, smashtuple, case, lift, cond
  and $\&$ some functional must be specified, i.e. an element of

$$I[ft] \times \cdots \times I[ft]_n \to I[ft]$$

Taking tuple as an example it is required that
An \( tenv \cdot e_i : ft \) implies \( tenv \cdot \text{tuple } e_1, \ldots, e_n : ft \)
and again this is slightly imprecise because tuple has not
been indexed with \( ft_1, \ldots, ft_n, ft \).
- for each constant \( f_i \) of contravariantly pure type \( t_i \), there
must be an element of \( I(t_i) \).

We write \( I(\text{take}_i), I(\text{tuple}), I(f_i) \) etc. for these entities. The
conditions upon the functionals are essentially as in the notion of
a "continuous algebra".

To define the expression part of the standard \( S \)
we use the conditional \( v_1 \rightarrow v_2 \cdot v_3 \); it denotes \( v_2 \cdot v_3 \) or \( \bot \) according
to whether \( v_1 \) is \( \top, \bot \) or \( \bot \). The functions are given by:

\[
\begin{align*}
S(\text{take}_i)(v) & = v_i \\
S(\text{smashtake}_i)(v) & = v_i \\
S(\text{in}_i)(v) & = \text{in}_i(v) \\
S(\text{up})(v) & = \text{up}(v) \\
S(\text{fold})(v) & = \emptyset(v) \quad \text{-- see below} \\
S(\text{unfold})(v) & = \emptyset^{-1}(v) \quad \text{-- see below}
\end{align*}
\]

Here it is assumed that \( \emptyset \cdot \text{fold} : \text{gt} \xrightarrow{\text{recX} \cdot \text{gt}/X} \text{recX} \cdot \text{gt} \) and that \( \emptyset \) is
the second component of the initial fixed point \( (D, \emptyset) \) of the functor
\( \amalg \text{gt} \xrightarrow{\text{accs}} \text{accs} \). (That the functionalities of \( \emptyset \) and \( \emptyset^{-1} \) are
correct follows from lemma 2.3:2.) The functionals are given by:

\[
\begin{align*}
S(\text{tuple})(f_1, \ldots, f_k)(v) & = (f_1(v), \ldots, f_k(v)) \\
S(\text{smashtuple})(f_1, \ldots, f_k)(v) & = \text{smash}(f_1(v), \ldots, f_k(v)) \\
S(\text{case})(f_1, \ldots, f_k)(v) & = \text{is}_1(v) \rightarrow f_1(\text{out}_1(v)), \ldots \\
& \quad \text{is}_k(v) \rightarrow f_k(\text{out}_k(v)), \bot
\end{align*}
\]
\( S(lift)(f)(v) = \text{def}(v)\Rightarrow f(down(v)), \bot \)
\( S(\text{cond})(f_1,f_2,f_3)(v) = f_1(v)\Rightarrow f_2(v), f_3(v) \)
\( S(\text{do})(f_1,f_2)(v) = f_1(f_2(v)) \)

Finally, \( S(f_1) \) will not be specified. It is straightforward to verify that \( S \) is an interpretation.

The definition of the semantics is mostly straightforward but for completeness sake we give all the clauses. We begin with the top-level metalanguage and use the notation of the previous section:

\[
\begin{align*}
I[e_1,\ldots, e_k]_{tenv}(env) &= (I[e_1]_{tenv}(env),\ldots,I[e_k]_{tenv}(env)) \\
I[e'_i]_{tenv}(env) &= (I[e'_j]_{tenv}(env))_{i} \\
I[x{{}\equiv e}_1,\ldots, e_k]_{tenv}(env) &= \text{smash}(I[e_1]_{tenv}(env),\ldots) \\
I[e'_i]_{tenv}(env) &= (I[e'_j]_{tenv}(env))_{i} \\
I[is_i e'_i]_{tenv}(env) &= is_i(I[e'_j]_{tenv}(env)) \\
I[in_i e'_i]_{tenv}(env) &= in_i(I[e'_j]_{tenv}(env)) \\
I[ou_i e'_i]_{tenv}(env) &= ou_i(I[e'_j]_{tenv}(env)) \\
I[\text{def } e'_i]_{tenv}(env) &= \text{def}(I[e'_j]_{tenv}(env)) \\
I[\text{up } e'_i]_{tenv}(env) &= \text{up}(I[e'_j]_{tenv}(env)) \\
I[\text{down } e'_i]_{tenv}(env) &= \text{down}(I[e'_j]_{tenv}(env)) \\
I[\lambda x:t.e'_i]_{tenv}(env) &= f \\
\quad \text{where } f(v) = \frac{I[e'_j]}{tenv[t/x]}_{tenv}(env[v/x]) \\
I[e'_i e''_i]_{tenv}(env) &= (I[e'_j]_{tenv}(env))_{i}(I[e'']_{tenv}(env)) \\
I[x]_{tenv}(env) &= \text{env}(x) \\
I[mkrec e'_i]_{tenv}(env) &= \Theta(I[e'_j]_{tenv}(env))_{i} \\
\quad \text{where } \text{tenv} \vdash \text{mkrec } e'::\text{rec }X.t \text{ and } (D, (\Theta, \Theta^{-1})) \text{ is the initial fixed point of } I[t]\text{ over } \text{CP}02; \text{ by lemma 2.3:2 the functionality is correct} \\
I[unrec e'_i]_{tenv}(env) &= \Theta^{-1}(I[e'_j]_{tenv}(env)) \\
\quad \text{where } \text{tenv} \vdash e'::\text{rec }X.t \text{ and } \Theta^{-1} \text{ is as above}
\end{align*}
\]
\[
\mathcal{I}[e_1 \rightarrow e_2, e_3]_{\text{tenv}}(\text{env}) = \mathcal{I}[e_1]_{\text{tenv}}(\text{env}) \rightarrow \mathcal{I}[e_2]_{\text{tenv}}(\text{env}), \quad \mathcal{I}[e_3]_{\text{tenv}}(\text{env})
\]

\[
\mathcal{I}[\forall e']_{\text{tenv}}(\text{env}) = \text{LFP}(\mathcal{I}[e']_{\text{tenv}}(\text{env}))
\]

\[
\mathcal{I}[f]_{\text{tenv}}(\text{env}) = \mathcal{I}(f)
\]

and for the bottom-level:

\[
\mathcal{I}[\text{tuple} \ e_1, \ldots, e_k]_{\text{tenv}}(\text{env}) = \mathcal{I}(\text{tuple})(\mathcal{I}[e_1]_{\text{tenv}}(\text{env}), \ldots)
\]

\[
\mathcal{I}[\text{take} i]_{\text{tenv}}(\text{env}) = \mathcal{I}(\text{take}_i)
\]

\[
\mathcal{I}[\text{smashtuple} \ e_1, \ldots, e_k]_{\text{tenv}}(\text{env}) = \mathcal{I}(\text{smashtuple})(\mathcal{I}[e_1]_{\text{tenv}}(\text{env}), \ldots, \mathcal{I}[e_k]_{\text{tenv}}(\text{env}))
\]

\[
\mathcal{I}[\text{smashtake} i]_{\text{tenv}}(\text{env}) = \mathcal{I}(\text{smashtake}_i)
\]

\[
\mathcal{I}[\text{case} \ e_1, \ldots, e_k]_{\text{tenv}}(\text{env}) = \mathcal{I}(\text{case})(\mathcal{I}[e_1]_{\text{tenv}}(\text{env}), \ldots)
\]

\[
\mathcal{I}[\text{in} i]_{\text{tenv}}(\text{env}) = \mathcal{I}(\text{in}_i)
\]

\[
\mathcal{I}[\text{lift} e']_{\text{tenv}}(\text{env}) = \mathcal{I}(\text{lift})(\mathcal{I}[e']_{\text{tenv}}(\text{env}))
\]

\[
\mathcal{I}[\text{up}]_{\text{tenv}}(\text{env}) = \mathcal{I}(\text{up})
\]

\[
\mathcal{I}[\text{fold}]_{\text{tenv}}(\text{env}) = \mathcal{I}(\text{fold})
\]

\[
\mathcal{I}[\text{unfold}]_{\text{tenv}}(\text{env}) = \mathcal{I}(\text{unfold})
\]

\[
\mathcal{I}[\text{cond} \ e_1, e_2]_{\text{tenv}}(\text{env}) = \mathcal{I}(\text{cond})(\mathcal{I}[e_1]_{\text{tenv}}(\text{env}), \ldots)
\]

\[
\mathcal{I}[e_1, e_2]_{\text{tenv}}(\text{env}) = \mathcal{I}((\mathcal{I}[e_1]_{\text{tenv}}(\text{env}), \mathcal{I}[e_2]_{\text{tenv}}(\text{env}))
\]

**Lemma 2.3.4.** If \( \text{tenv} \cdot e : t \) then the above equations define a continuous function \( \mathcal{I}[e]_{\text{tenv}} : \mathcal{T} \times \text{dom(tenv)} \rightarrow \mathcal{T} \).

The proof is by structural induction and is omitted. It is a consequence that LFP is only applied to a continuous function.

The notion of interpretation defined here extends that of /CoCo77b/ in e.g. including specification of functionals rather than assuming a fixed interpretation of these. Still some things that are fixed might ideally be allowed to vary and this identifies limitations.
in the present degree of "ambition". One such point is the fixed interpretation of \( \rightarrow \). Another is that \( Y \) and \( \text{rec} \) always give the least fixed point and initial solution, respectively. This will be discussed further in chapter 6.

Remark The use of \( \text{ACC} \) rather than \( \text{ACC}\_c \) for \( \\text{S(B)} \) is not entirely satisfactory. An argument against \( \text{ACC}\_c \) is that \( \text{up} \) is not strict. A stronger argument in favour of \( \text{ACC} \) is that the morphisms in the bottom-level express "state transformations" and they should be strict because "once something has failed to terminate nothing more can be done" /Sto77 p.203/. This will of course entail removing \( \text{up} \) (and \( \text{lift} \) and \( \downarrow \)) and claim that \( \text{up} \) does not correspond to any operational intuition. This amounts to continuing the discussion started in the second remark in section 2.1. In view of this it is interesting to note that \( \downarrow \) gives "problems" in section 4.4 which are similar to those \( \uparrow \) gives.

2.4 APPLICATIONS OF THE METALANGUAGE

The notation for the bottom-level metalanguage may be somewhat unfamiliar and there are limitations in the types allowed. The purpose of this section is therefore to show that the metalanguage is still useable. This is done by giving two example semantics but they will not be needed later.

The first and major example is an imperative language with procedures. The abstract syntax of programs \( \text{pro} \), expressions \( \text{exp} \), commands \( \text{cmd} \) and declarations \( \text{dcl} \) is given by:
pro ::= program cmd

exp ::= num | true | false | read | ide | ide(exp₁,...,expₖ)
    | if exp₁ then exp₂ else exp₃ | exp₁ ope exp₂

cmd ::= ide := exp | write exp | ide(exp₁,...,expₖ)
    | if exp then cmd₁ else cmd₂ | while exp do cmd
    | begin dcl ; cmd ; cmd₁ ; cmd₂

dcl ::= var ide := exp | fun ide(id₁,...,ideₖ) ; exp
    | proc ide(id₁,...,ideₖ) ; dcl₁;dcl₂

The syntax of identifiers ide, numerals num and operators ope is left unspecified. The language is similar to SMALL /Gor79/ but there are some syntactic deviations. For simplicity only identifiers may be assigned to, but to be general functions and procedures may have more than one argument. Variables are declared by var ide := exp but there is no facility for constants. The potential problem with constants is that we do not allow t::=gt. So one would have to treat constants using locations (as will be done for variables) or syntactic constraints must ensure that the identifiers referenced when initialising a constant are themselves constants. A semantic deviation is that we shall use call-by-value.

We now define the functionalities of the semantic valuations \( \mathcal{P},\mathcal{B},\mathcal{E} \) and \( \mathcal{I} \). The semantics will use locations and, to illustrate some possibilities, use a mixture of continuation style and direct style. To state the functionalities we need the bottom-level types \( I \) (input), \( O \) (output), \( S \) (state), \( E \) (value) and the top-level types \( L \) (locations) and \( R \) (environments). The functionalities to be used are listed below and explained afterwards.
Programs should be straightforward. For expressions one might have expected \( \xi(\text{exp}) : R \rightarrow L \rightarrow S \rightarrow E \times S \), i.e. that given an environment a "state transformation" is produced. However, the semantics of functions will require to know the "next free location" so that the arguments can be stored from there onwards. This information is available for the choice taken. Similar considerations apply for commands and declarations.

Before proceeding any further it is appropriate to point out that the metalanguage has been designed to be "minimal" in not having explicit notation for constructs that can be built from more primitive constructs. This simplifies the formal development but makes examples more cumbersome. An example is \( M = \text{rec}M \cdot 0+E \times M \) that will be used to associate locations (in the form of natural numbers) with their values. An informal expansion of the definition gives \( 0+E \times (0+...) \) which may be rearranged to give \( 0+E+E^2+E^3+... \). So \( M \) is the type of finite lists of values (of type \( E \)) and location 1 may be viewed as corresponding to the 1'th element. Operations upon such lists can be defined using the constructs of the metalanguage but the definitions become rather detailed. An example of this is the function get below for accessing the contents of a location. One way to improve upon this would be to extend the metalanguage with a list forming constructor and the associated operations. A more ad hoc solution would be to assume that \( M \) is a base type and that the operations are constants (thereby side stepping the metalanguage). We shall not do
either as this section still serves its purpose without. But for this reason no "worked examples" about the behaviour of the semantics will be given.

Continuing with the development we define the domains as follows:

\[ I = \text{rec}I.0+E*I \quad \text{input} \]
\[ O = \text{rec}O.0+E*O \quad \text{output} \]
\[ M = \text{rec}M.0+E*M \quad \text{"memory"} \]
\[ S = I*M*O \quad \text{states} \]
\[ E = T+... \quad \text{values} \]
\[ T \quad \text{truth values} \]
\[ L = N \quad \text{locations} \]
\[ F = L\rightarrow E*...*EWS \rightarrow EWS \quad \text{functions} \]
\[ P = (L\rightarrow S\rightarrow S)\rightarrow (L\rightarrow E*...*EWS\rightarrow S) \quad \text{procedures} \]
\[ R = \text{Id}e\rightarrow L+F+P \quad \text{environments} \]

The intention with these definitions is that e.g. \( S \) is a shorthand for \((\text{rec}I.0+E*I)*...*(\text{rec}O.0+E*O)\).

Most of the details due to the "minimality" of the metalanguage are encapsulated by the following four auxiliary functions. An expression like \([..] \) will be used instead of formally defining a constant function with that effect (in the standard semantics). Furthermore, where-clauses are used to help impose some structure upon the definitions. Also all occurrences of smashtuple and smashtake, are abbreviated to tuple and take. Finally, we shall write e.g. \( l-1 \) as a shorthand for the more correct \([\lambda l.l-1](l) \).
read : \( S \rightarrow E \times S \) (reads the next input value)

\[
\text{read} = \text{tuple}\left( \text{cases}\left( [\lambda x.\text{error value}, \text{take}_1] \circ \text{unfold} \circ \text{take}_1\right), \text{tuple}\left( \text{abbr}_1, \text{take}_2, \text{take}_3\right) \right)
\]

where \( \text{abbr}_1 = \text{cases}\left( \text{fold}_{\text{in}_1}, \text{take}_2\right) \circ \text{unfold} \circ \text{take}_1 \)

write : \( E \times S \rightarrow S \) (writes the value on the output)

\[
\text{write} = \text{tuple}\left( \text{take}_1, \text{take}_2, \text{take}_2, \ldots \right)
\]

\[
\text{... fold}_{\text{in}_2} \circ \text{tuple}\left( \text{take}_1, \text{take}_3, \text{take}_2\right) \right)
\]

get : \( L \rightarrow S \rightarrow E \times S \) (gets content of location)

\[
\text{get} = \lambda 1: L. \text{tuple}\left( \text{abbr}_1, \left( L \rightarrow L \right)^*\text{take}_2, [\lambda s.s]\right)
\]

where \( \text{abbr}_1 = \gamma\left( \lambda g: L \rightarrow M \rightarrow E \cdot \lambda 1: L. \text{abbr}_2 \circ \text{unfold} \right) \)

where \( \text{abbr}_2 = \text{cases}\left( \left[ \lambda 1.\text{error value} \right], \text{take}_1 = 1 \rightarrow \text{take}_1, g(1-1)\circ \text{take}_2 \right) \)

set : \( L \rightarrow E \times S \rightarrow S \) (updates contents of location)

\[
\text{set} = \lambda 1: L. \text{tuple}\left( \text{take}_1, \text{take}_2, \text{abbr}_1, \text{take}_3, \text{take}_2\right)
\]

where \( \text{abbr}_1 = (\text{abbr}_2)\left( L \rightarrow L \right)^*\text{take}_2, \text{take}_2, \text{take}_2\right) \)

where \( \text{abbr}_2 = \gamma\left( \lambda g: L \rightarrow E \times M \rightarrow M \cdot \lambda 1: L. \text{abbr}_3 \right) \)

where \( \text{abbr}_3 = \text{fold} \circ \text{cases}\left( \text{abbr}_7, \text{abbr}_6 \right) \circ \text{abbr}_5 \circ \text{abbr}_4 \)

where \( \text{abbr}_4 = \text{tuple}\left( \text{take}_1, \text{take}_1\right. \right)

\[
\left. \ldots, \text{cases}\left( \text{in}_1 \circ \text{fold}_{\text{in}_1}, \text{in}_2 \circ \text{unfold} \circ \text{take}_2 \right) \right)
\]

(of functionality \( E \times M \rightarrow E \times (M+E \times M) \))

where \( \text{abbr}_5 = \left[ [\lambda (x,y) \cdot \text{is}_1(y) \rightarrow \text{in}_1((x,\text{out}_1(y))) \right. \right)

\[
\left. \ldots \text{in}_2((x,\text{out}_2(y))) \right] \right)
\]

(of functionality \( E \times (M+E \times M) \rightarrow (E \times M) + E \times (E \times M) \))

where \( \text{abbr}_6 = \text{in}_2\left( l = 1 \rightarrow \text{tuple}\left( \text{take}_1, \text{take}_2, \text{take}_2 \right) \right) \)

\[
\ldots \text{tuple}\left( \text{take}_1, \text{take}_2, \text{take}_2 \right),
\left. g(l-1) \circ \text{tuple}\left( \text{take}_1, \text{take}_2, \text{take}_2 \right) \right) \right)
\]

(of functionality \( E \times (E \times M) \rightarrow 0 + E \times M \))
where \( \text{abbr}_7 = in_2^{(1=1 \to [\lambda x.x])} \), tuple([\lambda x. \text{some initial value}, q(1-1)])

(of functionality \( EWM \to \Theta + EWM \))

In the definition of get and set it should be noted that the fixed point operator \( Y \) has type \((t \to t) \to t\) for \( t = I \to ft \). This shows an "interaction" between the levels of types that seems to have no analogue for the recursion operators rec and rec allowed for types.

The semantic functions are defined below, using the same notational conventions as above:

\[
P^{	ext{program}} : I \to O \]

\[
P^{	ext{program cmdl}} = \text{take}_3 a
\]

The definition of expressions uses the notation \( m_k \) that is defined below:

\[
P^{	ext{exp}} : R \to L \to S \to E^S
\]

\[
P^{	ext{num}} r 1 = [\lambda s.((\text{some value}), s)]
\]

\[
P^{	ext{true}} r 1 = [\lambda s. (\text{in}_1(tt), s)]
\]

\[
P^{	ext{false}} r 1 = [\lambda s. (\text{in}_1(ff), s)]
\]

\[
P^{	ext{read}} r 1 = \text{read}
\]

\[
P^{	ext{ide}} r 1 = \text{is}_1(r[\text{ide}]) \to \text{get}(\text{out}_1(r[\text{ide}])) \to [\lambda s. (\text{error value}, s)]
\]

\[
P^{	ext{ide}(exp_1, \ldots, exp_k)} r 1 = \text{is}_2(r[\text{ide}]) \to \text{abbr}_1, [\lambda s. (\text{error}, s)]
\]

where \( \text{abbr}_1 = (\text{out}_2(r[\text{ide}]) \to (\text{error value}, s))^{m_{k-1}}(\text{exp}_k r 1) \)

\[
P^\text{if exp}_1 \text{ then exp}_2 \text{ else exp}_3 r 1 = \text{cond}(\text{abbr}_1, \text{abbr}_2, \text{abbr}_3)
\]

where \( \text{abbr}_1 = (\text{out}_1) \text{take}_1 a(\text{exp}_1 r 1) \)

where \( \text{abbr}_2 = (\text{exp}_2 r 1) \text{take}_2 a(\text{exp}_1 r 1) \)

where \( \text{abbr}_3 = (\text{exp}_3 r 1) \text{take}_2 a(\text{exp}_1 r 1) \)
We now explain the equation for \( \text{ide}(\exp_1, \ldots, \exp_k) \) when \( k=2 \). Then \( \exp_1 \) and \( \exp_2 \) are of functionality \( S \rightarrow S \) and therefore cannot be composed. This is remedied by using

\[
m_1(\exp_2 \ r 1) : \text{EWS} \rightarrow \text{EWS}
\]

The idea is that the first argument remains unchanged. The formal definition is

\[
m_k(f) = \text{tuple}(\text{take }_1, \ldots, \text{take }_k, \text{take }_1 f \text{take }_{k+1}, \text{take }_2 f \text{take }_{k+1})
\]

The function \( m_k \) is not of contravariantly pure type so \( m_k(f) \) should be regarded as a shorthand for the defining expression. (The definition of \( m_k \) may be compared with fit of /MiSt76/.)

For commands we have

\[
\text{cmd} : R \rightarrow (L \rightarrow S \rightarrow S) \rightarrow (L \rightarrow S \rightarrow S)
\]

\[
\text{ide} := \exp \ r 1 = c(1) \circ \text{abbr}_1 \circ (\exp \ r 1)
\]

where \( \text{abbr}_1 = \text{is}_1(\text{ide}) \rightarrow \text{set}(\text{out}_1(\text{ide})) \), \( (\lambda (v,s). \text{error}) \)

\[
\text{write} \exp \ r 1 = c(1) \circ \text{write} \circ (\exp \ r 1)
\]

\[
\text{ide}(\exp_1, \ldots, \exp_k) \ r 1 = \text{is}_3(\text{ide}) \rightarrow \text{abbr}_3[\lambda s. \text{error state}]
\]

where \( \text{abbr}_3 = (\text{out}_3(\text{ide}) c 1) \circ m_{k-1} (\exp_k \ r 1) \circ \cdots \circ (\exp_1 \ r 1)
\]

\[
\text{if} \exp \ \text{then} \ \text{cmd}_1 \ \text{else} \ \text{cmd}_2 \ r 1 = \text{cond}(\text{out}_1 \circ \text{take}_1 \circ (\exp \ r 1)
\]

\[
, (\text{cmd}_1 \ r 1) \circ \text{take}_2 \circ (\exp \ r 1)
\]

\[
, (\text{cmd}_2 \ r 1) \circ \text{take}_2 \circ (\exp \ r 1)
\]

\[
\text{while} \ \exp \ \text{do} \ \text{cmd} \ r 1 = \text{Y}(\lambda c'.L \rightarrow S \rightarrow S, \lambda l:L. \ \text{abbr}_1)
\]

where \( \text{abbr}_1 = \text{cond}(\text{out}_1 \circ \text{take}_1 \circ (\exp \ r 1)
\]
For declarations we need a variant of $m_k$:

$$m'_k(f) = \text{tuple}(\text{take}_1, \ldots, \text{take}_k, \text{tuple}(\text{take}_{k+1}, \text{take}_{k+2}))$$

(It should be clear that $m_k$ and $m'_k$ are special cases of a more general construct.) We then have

$$\text{Z[dl]} : \text{RL} \rightarrow \text{RXLS} \rightarrow \text{S}$$

$$\text{Z[de]} \text{ ide} := \text{exp}(r, 1) = (r[\text{in}_1(l)/\text{ide}], l+1, \text{abbr}_1)$$

where $\text{abbr}_1 = \text{set}(l) \circ (\text{Z[exp]} r 1)$

$$\text{Z[f]} \text{ ide}(\text{ide}_1, \ldots, \text{ide}_k) ; \text{exp}(r, 1) = (\text{abbr}_1, 1, [\lambda_3, \ldots])$$

where $\text{abbr}_1 = r[\text{in}_2(Y(\lambda_3 : L \rightarrow S \rightarrow S) \rightarrow (L \rightarrow S \rightarrow S, \lambda_1 : L, \text{abbr}_2))/\text{ide}]$

where $\text{abbr}_2 = \text{Z[exp]} \text{ abbr}_3 (l+k) \circ \text{set}(l) \circ \ldots \circ m'_{k-1}(\text{set}(l+k-1))$

where $\text{abbr}_3 = r[\text{in}_3(g)/\text{ide}][\text{in}_1(l)/\text{ide}_1] \ldots [\text{in}_1(l+k-1)/\text{ide}_k]$}

$$\text{Z[pr]} \text{ ide}(\text{ide}_1, \ldots, \text{ide}_k) ; \text{cmd}(r, 1) = (\text{abbr}_1, 1, [\lambda_3, \ldots])$$

where $\text{abbr}_1 = r[\text{in}_3(Y(\lambda_3 : L \rightarrow S \rightarrow S) \rightarrow (L \rightarrow S \rightarrow S, \lambda_1 : L, \text{abbr}_2))/\text{ide}]$

where $\text{abbr}_2 = \text{Z[cmd]} \text{ abbr}_3 c (l+k) \circ \text{set}(l) \circ \ldots \circ m'_{k-1}(\text{set}(l+k-1))$

where $\text{abbr}_3 = r[\text{in}_3(g)/\text{ide}][\text{in}_1(l)/\text{ide}_1] \ldots [\text{in}_1(l+k-1)/\text{ide}_k]$}

$$\text{Z[dl]}_1, \text{Z[dl]}_2(r, 1) = (\text{abbr}_1, l_1, \text{abbr}_1, l_2, \text{abbr}_1, l_3 \circ \text{abbr}_2)$$

where $\text{abbr}_1 = \text{Z[dl]}_1(r, 1) \downarrow 1, \text{Z[dl]}_1(r, 1) \downarrow 2$

where $\text{abbr}_2 = \text{Z[dl]}_1(r, 1) \downarrow 3$

The clauses rely on set to create a new location if the requested location does not already exist.
The use of continuation style for commands and direct style for declarations is not essential. For expressions problems arise if a continuation style semantics is wanted. In the discussion below the presence of \( L \) will be ignored. The usual equation

\[
\begin{align*}
\mathcal{E}[\text{exp}_1 \text{ ope exp}_2] r k &= \mathcal{E}[\text{exp}_1] r (\lambda v_1. \\
&\quad \mathcal{E}[\text{exp}_2] r (\lambda v_2. \\
&\quad \quad k([\text{operator}](v_1, v_2))))
\end{align*}
\]

is not appropriate because the two levels are mixed in a prohibited way. The idea with \( m_1 \) might suggest

\[
\begin{align*}
\mathcal{E}[\text{exp}_1 \text{ ope exp}_2] r k &= \mathcal{E}[\text{exp}_1] r ( \\
&\quad M_1(\mathcal{E}[\text{exp}_2] r)( \\
&\quad \quad k([\text{operator}])))
\end{align*}
\]

but it is a problem to find \( M_1 \). If a store semantics /MiSt76/ had been used it would be feasible to give a continuation style semantics (as in /Nie82/) as well as a direct style semantics. In a sense the use of \( m_k \) may be viewed as "implementing" a store semantics locally.

**Remark** In section 2.1 it was suggested that the type \( ft \) possibly might be thought of as code. Even if one accepts this view the above semantics is not a compiler because of the top-level domains participating in the definition of functions and procedures. In /MiSt76/ a compiler is developed by transforming a standard semantics (through a store semantics) to a stack semantics. It would be interesting to investigate whether this process can be formalised as "moving top-level notions out of the definitions of functions and procedures".
The second example is a small applicative language of recursion equation schemes that will be studied in chapter 5. The syntax of expressions exp and programs pro is given by

\[
\begin{align*}
\text{exp} &::= x_i \quad (1 \leq i \leq k) \\
&| F_i(\text{exp}_1, \ldots, \text{exp}_k) \quad (1 \leq i \leq n) \\
&| \Lambda_i(\text{exp}_1, \ldots, \text{exp}_k) \quad (1 \leq i) \\
\text{pro} &::= \text{let } F_1(x_1, \ldots, x_k) = \text{exp}_1 \text{ & } \ldots \\
&| \ldots & F_n(x_1, \ldots, x_k) = \text{exp}_n \text{ in } \text{exp}_0
\end{align*}
\]

Here \( k \) and \( n \) are natural numbers that will remain constant throughout. We shall not specify their values except assume that they are greater than 1.

The language consists of auxiliary functions (the \( \Lambda_i \)), functions defined by mutual recursion (the \( F_i \)) and variables (the \( x_i \)). The parameter mechanism is call-by-name rather than call-by-value as before. The meaning of a program is a function whose arguments are values for the free variables (in \( \text{exp}_0 \) above). We shall assume that there is only one data type \( E \) as this will slightly simplify the notation in chapter 5. The meaning of a program therefore will be an element of

\[
F = E \times \ldots \times E \rightarrow E
\]

where there are \( k \) \( E \)'s to the left of \( \rightarrow \). For environments we may use

\[
R = F \times \ldots \times F
\]

where there are \( n \) \( F \)'s.
The semantics then is

\[ \mathcal{E}[\text{let } \ldots \text{ in } \exp_0] = \mathcal{E}[\exp_0](\lambda \text{env}: \mathcal{T}. \text{abbr}_1) \]

where \( \text{abbr}_1 = (\mathcal{E}[\exp_1] \text{ env}, \ldots, \mathcal{E}[\exp_k] \text{ env}) \)

and

\[ \mathcal{E}[x_i] \text{ env} = \text{take}_i \]

\[ \mathcal{E}[f_1(\exp_1, \ldots, \exp_k)] \text{ env} = \]

\[ (\text{env} \downarrow i) \circ \text{tuple}(\mathcal{E}[\exp_1] \text{ env}, \ldots, \mathcal{E}[\exp_k] \text{ env}) \]

\[ \mathcal{E}[a_i \circ \text{tuple}(\mathcal{E}[\exp_1] \text{ env}, \ldots, \mathcal{E}[\exp_k] \text{ env})] \text{ env} = \]

Here \( a_i \) are to be constants of contravariantly pure type \( (\mathcal{T} \text{ in fact}) \).

There is no explicit conditional above but it may be assumed to be one of the \( A_i \) because the parameter mechanism is call-by-name. It could be introduced explicitly by

\[ \mathcal{E}[\text{if } \exp_1 \text{ then } \exp_2 \text{ else } \exp_3] \text{ env} = \]

\[ \text{cond}(\lambda e. e \text{ represents } \text{tt}) \circ \mathcal{E}[\exp_1] \text{ env}, \]

\[ \mathcal{E}[\exp_2] \text{ env}, \mathcal{E}[\exp_3] \text{ env} \]

but we shall not do so in chapter 5.
In the previous chapter a metalanguage was specified and its standard interpretation was given. In the next chapter we consider abstract interpretation, i.e. specification of (forward) data flow analyses. As with previous developments of abstract interpretation it is helpful to consider the most precise of all data flow analyses: the "static semantics" of /CoCo77b/ which is called the collecting semantics in this thesis.

The idea is that a function \( f: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N} \) becomes \( g: \mathcal{P}(\mathbb{N} \times \mathbb{N}) \rightarrow \mathcal{P}(\mathbb{N} \times \mathbb{N}) \). Here \( \mathcal{P}(\mathbb{N} \times \mathbb{N}) \) is a powerdomain which is an analogue within domain theory of the powersets used in the introduction. In section 3.1 several notions of powerdomains are exemplified and the theory is developed for the notion chosen. Section 3.2 studies how to obtain e.g. \( \mathcal{P}(\mathbb{N} \times \mathbb{N}) \) from \( \mathcal{P}(\mathbb{N}) \) as is necessary in order to define the collecting semantics by merely giving a new interpretation (the collecting interpretation). This motivates a study of the tensor product \( \otimes \) because \( \mathcal{P}(\mathbb{N} \times \mathbb{N}) \) "is" \( \mathcal{P}(\mathbb{N}) \otimes \mathcal{P}(\mathbb{N}) \). Further use of the tensor product will be made in chapter 4. The collecting interpretation is then defined in section 3.3 and a suitable relationship between the standard and collecting semantics is proved. The relationship essentially says that \( g(Y) = \{ f(y) \mid y \in Y \} \) which means that \( g \) is the extension of \( f \) to powerdomains.
Powerdomains were developed in order to make it possible to give semantics to nondeterministic and parallel programs. The semantics of a nondeterministic program like

\[ x := x + 1; (x := x + 27 \text{ or } x := x + 3) \]

is a function \( f : \mathbb{N} \to \mathcal{P}(\mathbb{N}) \). We define \( \mathcal{P}(\mathbb{N}) \) below for three notions of powerdomain. It should be stressed that the general definitions are more complex as will be apparent when the theory of relational powerdomains is covered.

The Plotkin powerdomain \( \mathcal{P}_P(\mathbb{N}) /Plo76/ \) has as elements some of the \( \mathcal{P} \) subsets of \( \mathbb{N} \). A subset of \( \mathbb{N} \) is an element of the powerdomain iff it contains \( \bot \) or is finite but nonempty. The partial order is the Egli-Milner order defined by

\[ X \leq_E M Y \text{ iff for all } x \in X \text{ there is } y \in Y \text{ such that } x \leq y, \text{ and for all } y \in Y \text{ there is } x \in X \text{ such that } x \leq y \]

Then \( \mathcal{P}_P(\mathbb{N}) \) is a cpo and may be pictured as:

\[ \begin{array}{c}
\mathcal{P}_P(\mathbb{N}) : \\
\{ \bot, 2, 3, \ldots \} \\
\{ 2, 3 \} \\
\{ \bot, 2 \} \\
\{ \bot \} \\
\\end{array} \]

The Smyth powerdomain /Smy78/ has as elements \( \mathbb{N} \) and all finite and nonempty subsets of \( \mathbb{N} \). The partial order is superset inclusion, i.e., \( X \leq Y \iff X \supseteq Y \). This may be pictured as:
The relational powerdomain $\mathcal{P}_R(N_\bot)$ (taken from /PloLN/) has as elements all subsets of $N_\bot$ that contain $\bot$ and is partially ordered by subset inclusion, i.e. $X \subseteq Y$ iff $X \subseteq Y$. This may be pictured as:

\[
\mathcal{P}_R(N_\bot):
\begin{align*}
\{1\} & \rightarrow \{2\} & \rightarrow \{3\} \\
\{1, 2\} & \rightarrow \{2, 3\} \\
\{1, 2, 3\} & \rightarrow \ldots \\
N_\bot & \rightarrow \ldots \\
\{1\} & \rightarrow \ldots \\
\{1, 2\} & \rightarrow \ldots \\
\{1, 2, 3\} & \rightarrow \ldots \\
\end{align*}
\]

One may note that $\mathcal{P}_R(N_\bot)$ is isomorphic to the ordinary powerset of $N$.

It will now be argued that the relational powerdomain is the better choice of the three possibilities. One argument is the isomorphism (in CPO) that was mentioned above and therefore a development based on this powerdomain may be claimed to be "a generalisation" of the usual theory (e.g. /CoCo77b/). The Smyth powerdomain does not seem usable because it includes only one infinite subset of $N_\bot$ and in data flow analysis one often considers representations of infinite sets (as in the "detection of signs" example in the introduction).

The formulation of abstract interpretation given in the introduction focuses upon what set of states will be possible at some point (say at the end of the program) not whether it can be guaranteed that the point is always reached. This agrees with the relational powerdomain where all elements contain $\bot$. The Plotkin powerdomain (as was used in /Myc81/) enables one to express that a certain point is always
reached (i.e. that the set does not contain \( \bot \)). This is useful when extending abstract interpretation to include aspects of termination and is the subject matter of chapter 5. It will be argued, however, that the absence of infinite sets not containing \( \bot \) severely limits the additional generality that is obtained (and the major application in /Myc81/ is incorrect). For this reason, and less importantly because the theory of relational powerdomains is simpler, we shall adopt the relational powerdomain.

Definition of the relational powerdomain

To define the relational powerdomain we need the notion of an ideal. A tuple \( B = (B, \bot, \leq) \) is a **pointed quasi order** iff \( \leq \) is a quasi order on \( B \) (i.e. a partial order except that \( x \leq y \leq x \) does not imply \( x = y \)) and \( \bot \) is a least element. For \( D = (D, \leq) \) an algebraic cpo the tuple \( (B_D, \bot, \leq) \) is a **countable pointed quasi order** (because \( B_D \) is countable).

An **ideal** \( I \) of a pointed quasi order \( B \) is a subset of \( B \) that contains \( \bot \) and is left-closed: if \( b \leq b' \) and \( b' \notin I \) then \( b \notin I \). It is convenient to define \( LC_B(X) = \{ b \in B \mid \exists x \in X : b \leq x \} \) and the index \( B \) is usually omitted. The ideals of \( N_\bot \) are exactly the elements of \( \varrho_R(N_\bot) \). This motivates (omitting the subscript \( R \)):

**Definition** For a cpo \( D \) define \( \varrho(D) = (\{ \text{ideals of } B_D \}, \leq) \).

**Theorem 3.1:1** /PloLN/. If \( D \) is an algebraic cpo then \( \varrho(D) \) is an algebraic complete lattice with \( \bigcup X = \bigcup_{X} \cup X \) and finite elements those ideals \( I \) that equal \( LC(Y) \) for some finite subset \( Y \) of \( B_D \).

**Proof** It is straightforward that \( \varrho(D) \) is a complete lattice with least upper bounds as stated. If \( Y \subseteq B_D \) is finite and nonempty then \( I = LC(Y) \)
is a finite element: for if $I \subseteq \bigcup_{n} I_n$ for some chain $(I_n)_n$ then $\forall n \in \mathbb{N}$ so $\forall n \in \mathbb{N}$ for some $n$ and hence $\forall n \in \mathbb{N}$. We next use Fact 2.2.1 so let $I$ be an ideal. Because $B$ is countable there is a one-one map $i: B \to \mathbb{N}$. Then $(LC \{ \{b \} \cup \{ b \in B \mid b \in I \land i(b) \leq n\} \})_n$ is a chain of finite elements with least upper bound $I$.

The notion of singletons for powersets motivates defining $\{d\}_R = \{b \in B \mid b \in d\} = LC(\{d\})$ whenever $d \in D$. It is straightforward to show that $\lambda d . \{d\}_R$ is a continuous function from $D$ to $\mathcal{P}(D)$.

**Theorem 3.1.2 /PloLN/**. Let $D$ be an algebraic cpo and $L$ a complete lattice. For a continuous function $f: D \to L$ there is precisely one continuous and additive function $f^+: \mathcal{P}(D) \to L$ such that $f^+(\{d\}_R) = f(d)$ for all $d \in D$. It is given by $f^+(I) = \bigcup \{f(b) \mid b \in I\}$ and $\lambda f . f^+$ is an isomorphism from $D \to L$ to $\mathcal{P}(D) \to L$ (the cpo of additive as well as continuous functions) with inverse $\lambda g . \lambda d . g(\{d\}_R)$.

**Proof** The explicitly defined $f^+$ is continuous as is shown by

$$f^+(\bigcup_{n} I_n) = \bigcup f(b) \mid b \in \bigcup_{n} I_n \} = \bigcup f(b) \mid b \in I \} \mid n \in \mathbb{N}\} = \bigcup f^+(I_n)$$

and a similar calculation shows that $f^+$ is additive. Next

$$f^+(\{d\}_R) = \bigcup f(b) \mid b \in B \land b \in d\} = f(d)$$

because $f$ is continuous and $\{d\}_R$ is a countable and directed subset of $D$. For any other $g$ satisfying the conditions we show $g(I) = f^+(I)$ for $I$ an ideal of $B$. By $g$ continuous it suffices to consider ideals that are finite elements and by $g$ additive it suffices to consider $I = \{b\}_R$ for $b \in B$. The result is then immediate. It follows that $\lambda f . f^+$ is a bijection as stated and it and its inverse are easily seen to be monotonic so that $\lambda f . f^+$ is an isomorphism (in CPO). //

95
The above result is useful for defining \( \rho(\ldots) \) as a functor. Let \( \text{ALG} \) be the category with algebraic cpo's as objects and continuous functions as morphisms. It is an admissible subcategory of \( \text{CPO} \) (see the proof for \( \text{ACC} \) in section 2.2). For a morphism \( f:D \rightarrow D' \) we define \( \rho(f) = (\lambda d. \{ f(d) \})_R^T \) and the definition on objects has already been given. This specifies a locally continuous and covariant functor over \( \text{ALG} \) as is now proved. That \( \rho(f \circ g) = \rho(f) \circ \rho(g) \) is by the "precisely one" result of the theorem: for \( \rho(f) \circ \rho(g) \) is continuous and additive and \( \{(f \circ g)(d)\}_R = \rho(f) \{g(d)\}_R = \{f(g)(d)\}_R \). One shows \( \rho(\text{id}) = \text{id} \) in a similar way. That \( \rho \) is continuous from \( D \rightarrow D' \) to \( \rho(D) \rightarrow \rho(D') \) is because \( \lambda f. f^{-1} \) is continuous from \( D \rightarrow \rho(D') \) to \( \rho(D) \rightarrow \rho(D') \); the latter claim follows from the theorem because additivity is an admissible predicate. The theorem states that \( \rho(D) \) is a complete lattice and it is immediate that \( \rho(f) \) is strict iff \( f \) is. Hence \( \rho(\ldots) \) is a locally continuous and covariant functor over all of \( \text{ALG, ALGs, ACC, ACCs} \).

There are other isomorphic ways of defining the relational powerdomain. One that may have more "intuitive appeal" is the following where members of \( \rho(D) \) do not only contain elements of \( B_D \). A subset \( X \) of \( D \) is closed iff it equals \( \{d \in D \mid \text{LC}(\{d\}) \subseteq \text{LC}(X)\} \). Then \( \rho(D) \) is isomorphic to \( \{\text{nonempty closed subsets of } D\}, \subseteq \}/\text{PloLN} \) but this result will not be used subsequently.

**Other characterisations**

The collecting interpretation uses powerdomains but the general treatment of abstract interpretation in chapter 4 will also consider other algebraic complete lattices. This means we will use the
category $\mathcal{ACL}$ that has algebraic complete lattices as objects and continuous functions as morphisms. It is an admissible subcategory of $\mathfrak{CPO}$ (as follows from an easy modification of the proof for $\mathfrak{ACC}$).

In the remainder of this section we consider what special there is about powerdomains, i.e. we identify the image in $\mathcal{ACL}$ of $\mathfrak{P}$ acting on $\mathcal{ALG}$.

For this we need to define the ideal completion of a countable and pointed quasi order $B = (B, \sqsubseteq, \bot)$. A directed ideal of $B$ is an ideal of $B$ that is a directed subset.

**Definition** For a pointed quasi order $B$ define

$$\overline{B} = (\{\text{directed ideals of } B\}, \sqsubseteq).$$

**Theorem 3.1** ($\mathcal{PLOLN}$). If $B$ is a countable and pointed quasi order then $\overline{B}$ is an algebraic cpo. The finite elements of $\overline{B}$ are those of the form $\text{LC}(\{b\})$ for $b \in B$.

**Proof** Clearly $\overline{B}$ is a partially ordered set with $\{\bot\}$ as least element. It is a cpo because for $(J_n)_n$ a chain of directed ideals $\bigcup J_n$ is a directed ideal and therefore the least upper bound. Clearly an element $\text{LC}(\{b\})$ for $b \in B$ is finite. We now once more use fact 2.2: 1.

Let $i: \mathbb{N} \to B$ be onto and $J$ a directed ideal. Set $b_0 = \bot$ and $b_{n+1} = b_n$ if $i(n+1) \notin J$ and otherwise $b_{n+1}$ is chosen as some upper bound in $J$ of $b_n$ and $i(n+1)$. (This uses the countable axiom of choice.) Clearly $(\text{LC}(\{b_n\}))_n$ is a chain of finite elements with $J$ as the least upper bound.

**Fact 3.1:4**. If $D$ is an algebraic cpo then $D$ is isomorphic to $\overline{D}$. The isomorphism from $D$ to $\overline{D}$ is $\lambda d. \text{LC}(\{d\})$ and its inverse is $\lambda J. iJ$. 

---

97
The characterisation of powerdomains builds on the notion of a prime element. (The definition of prime and irreducible below is dual to that of [GHKLMS80].) Let $D = (D, \leq)$ be an algebraic cpo.

**Definition** An element $d \in D$ is a **prime** iff $d \not\leq d_1 \cup d_2$ implies that $d = d_1$ or $d = d_2$ and it is **irreducible** iff $d = d_1 \cup d_2$ implies that $d = d_1$ or $d = d_2$.

Write $P_D^b = \{ b \in B \mid b$ is a prime$\}$ and $I_D^b = \{ b \in B \mid b$ is irreducible$\}$. If $D$ is consistently complete as well as algebraic and if $d$ is finite then $d_1$ and $d_2$ may be assumed finite in the above definition without changing the concept.

**Theorem 3.1:** An algebraic complete lattice $L$ is isomorphic to a powerdomain (of an algebraic cpo) iff

$$B_L = \{ b_1 \cup \ldots \cup b_n \mid n > 0, b_i \in P_D^b \}$$

and then $L \cong (P_D^b)$.

**Proof** To show "only if" we may without loss of generality assume $L = \emptyset(D)$ for $D$ an algebraic cpo. The finite elements of $L$ are by 3.1.1 those of the form $\bigcup_{i \in \mathbb{N}} L C(D, \{ b_i \})$ for $n > 0$ and $b_i \in B_D$. It therefore suffices to show that the finite primes are those of the form $L C(D, \{ b \})$ for $b \in B_D$. Clearly such an element is prime and if a finite element $d = \bigcup_{i \in \mathbb{N}} L C(D, \{ b_i \})$ is not of this form then there is $j < k$ such that $b_j$ and $b_k$ are maximal in $\{ b_1, \ldots, b_n \}$ but neither $b_j \leq b_k$ nor $b_k \leq b_j$. Then using $d_1 = L C(D, \{ b_i \mid i \in \mathbb{N}, b_i \neq b_j \})$ and $d_2 = L C(D, \{ b_i \mid i \in \mathbb{N}, b_i \neq b_k \})$ in the definition of prime shows that $d$ is not prime.

For "if" let $L$ be an algebraic complete lattice satisfying the condition. Then $D = \overline{P_D^b}$ is an algebraic cpo and we will show $L \cong (D)$.
Since $L^\perp B^L_D$ and $B_D$ is isomorphic to $PB_L$ (in a category with partially ordered sets and continuous functions) it suffices to show that $L' = \{\text{directed ideals of } B^L\}$ is isomorphic to $L'' = \{\text{ideals of } PB_L\}$.

Define $\theta: L' \rightarrow L''$ by $\theta(J) = \bigwedge PB_L$ and $\theta^{-1}: L'' \rightarrow L'$ by $\theta^{-1}(I) = \{b_1 \wedge \cdots \wedge b_n \mid n \neq 0 \land b_k \in I\}$. Note that $b_1 \wedge \cdots \wedge b_n$ exists in $L$ (because $L$ is a complete lattice) and gives an element of $B_L$ because each $b_i$ is a finite element of $L$. So $\theta^{-1}(I)$ is a directed ideal of $B_L$ and $\theta(J)$ is an ideal of $PB_L$. It is straightforward to show that $\theta(\theta^{-1}(I)) = I$ and $\theta^{-1}(\theta(J)) = J$ using the connection between $B_L$ and $PB_L$. Finally, both $\theta$ and $\theta^{-1}$ are monotonic.

Further information about powerdomains is given by:

**Fact 3.1.6.** A prime of an object of $ACL$ is irreducible but not necessarily conversely.

For the proof note that the complete lattice $V$ defined by

![Diagram of lattice](image)

has $PB_V = \{1\}$, $IB_V = \{-, 0, +, 1\}$ and $B_V = V$.

**Theorem 3.1.7.** If $D$ is an algebraic cpo and $I$ is an element of $P(D)$ then it is a prime iff it is irreducible iff it is a singleton ($I = \{d\}_R$ for some $d \in D$).

**Proof** If $d \in D$ then $\{d\}_R$ is a prime and hence irreducible: for let $d = \bigcup b_n$ where $(b_n)$ is a chain of finite elements and suppose that $\{d\}_R \subseteq I_1 \cup I_2$. Then without loss of generality $I_1$ contains $b_n$ for infinitely many $n$, hence $I_1$ contains all $b_n$ and $\{d\}_R \subseteq I_1$. 

99
Next we show that an irreducible ideal $I$ must be a singleton. If $I$ is directed then $I = \{UI\}_R$ so we will show that $I$ must be directed. For a contradiction assume that $I$ is not directed but is irreducible.

There are $b_1, b_2 \in I$ that have no upper bound in $I$. Define the ideals $I_1 = \text{LC}(\{b \in I \mid b \leq b_1\})$ and $I_2 = \text{LC}(\{b \in I \mid b \leq b_2\})$. Then $I_1$ contains $b_1$ but not $b_2$ and $I_2$ contains $b_2$ but not $b_1$. But $I = I_1 \cup I_2$ is immediate and since $I \neq I_1$ and $I \neq I_2$ the desired contradiction has been established.

In chapter 4 we shall consider algebraic complete lattices $L$ satisfying $B_L = \{b_1 \ldots b_n \mid n \geq 0 \land b_i \in L\}$. The previous two theorems show that the powerdomains are included (as is $V$ incidentally). The condition of the previous theorem cannot be used to characterise powerdomains:

**Fact 3.1**: The powerdomains are a proper subset of the set of $\text{ACL}$ objects where primes and irreducible means the same.

For the proof note that a non-powerdomain satisfying the condition is the following: the elements are $\{\emptyset, V\}$ and $u \subseteq v$ iff $u = \bot$ or $u$ is a finite string over $\{0, 1\}$ with $v$ as a prefix.

Turning to functions we have:

**Theorem 3.1**: Let $D$ and $D'$ be algebraic cpo's and $g : \mathcal{P}(D) \to \mathcal{P}(D')$ a continuous function. There exists a continuous function $f : D \to D'$ such that $g = \mathcal{P}(f)$ iff $g$ is additive and prime-preserving (i.e. that $f(d)$ is a prime whenever $d$ is).

**Proof** By a previous theorem the primes are the singletons and this shows that $\mathcal{P}(f)$ is prime-preserving and it is clearly additive. Next given $g$ define $f(d) = \bigcup_j (\{a\}_R)$ as a continuous function. Using 3.1.2 it follows that $g = \mathcal{P}(f)$.
3.2 TENSOR PRODUCTS

Consider a type $\Lambda_1 \# \ldots \# \Lambda_k$ where $\#$ is one of the symbols $\times$, $\star$, $+$ or $\perp$. In the standard semantics the corresponding domain is $\Lambda_1 \# \ldots \# \Lambda_k$ and in the collecting semantics it will be $\wp(\Lambda_1 \# \ldots \# \Lambda_k)$. The problem is how to build $\wp(\Lambda_1 \# \ldots \# \Lambda_k)$ in a structural way. This entails finding an operator (H) such that $\wp(\Lambda_1 \# \ldots \# \Lambda_k)$ is at least isomorphic to $\wp(\Lambda_1 \perp \ldots \perp \Lambda_k)$. The operator should preferably work for all algebraic complete lattices rather than just powerdomains in order to facilitate the development of abstract interpretation in chapter 4. When $\#$ is $\perp$ one may use $\perp$ for $\perp$ and when $\#$ is $+$ one may use $\times$ for $\times$. It is less clear what to do when $\#$ is $\times$ or $\star$ and this motivates a study of tensor products /ArMa75/. A further "pay off" of this is given in chapter 4 where it is claimed that the tensor product generalises the "relational method" described in chapter 1.

**DEFINITION AND CONSTRUCTION**

Before giving the categorical definition of the tensor product it is convenient to state some auxiliary results. The categories to be considered in this section are $\text{ACL}$, $\text{ACL}_{\perp}$, $\text{ACL}_{\times}$ and $\text{ACL}_{\star}$ where the $\perp$ means that morphisms are additionally additive.

**Lemma 3.2:** $\text{ACL}$, $\text{ACL}_{\perp}$, $\text{ACL}_{\times}$ and $\text{ACL}_{\star}$ are admissible subcategories of $\text{CPO}$.

**Proof** For $\text{ACL}_{\perp}$ and $\text{ACL}_{\times}$ this is much as for $\text{ACC}$ in section 2.2. To get the rest of the result consider a chain $((D_n, e_n)_n)$ in $\text{ACL}_{\perp}$ (or equally $\text{ACL}_{\times}$) and let $(D_n, (r_n)_n)$ be the limiting cone in $\text{CPO}_{\perp}$. To see that each $r_n$ is a morphism of $\text{ACL}_{\perp}$ it suffices to show that...
each $r_n^U$ is additive. But each $e_n^U$ is additive and therefore $D$ as constructed in 2.2:2 has componentwise binary least upper bounds. It then follows from the formula for $r_n^U$ that the function is additive. //

Define an (upper) semi-lattice as a partially ordered set $L = (L, \leq)$ where all binary least upper bounds exist.

---

(3.2.2) A morphism of $ACL$ is in $ACL_A$ iff it preserves least upper bounds of all non-empty families. //

---

The notion of separate additivity is central to the definition of the tensor product. A function $f: L_1 \times \cdots \times L_k \rightarrow L$ (for $k \geq 2$) is separately additive (respectively separately continuous, separately strict) iff for each $i$ and $(1, \ldots, l_k)$ the function $\lambda_l f(1, \ldots, l_i, \ldots, 1_k)$ is additive (respectively continuous, strict). Separate strictness implies strictness but not conversely whereas separate additivity is implied by additivity but not conversely. It follows from section 2.2 that separate continuity is the same as continuity.
The definition of the tensor product is concerned with transforming a separately additive function to an additive function.

**Definition** A tensor product in $\mathbb{ACL}$ for $\times$ with respect to additivity assigns to any $k$ objects $L_1, \ldots, L_k$ an object $L_1 \otimes \ldots \otimes L_k$ and a separately additive morphism

$$\text{cross}_\times : L_1 \times \cdots \times L_k \to L_1 \otimes \cdots \otimes L_k$$

that is called the inclusion. Furthermore, for every separately additive morphism $f : L_1 \times \cdots \times L_k \to L$ of $\mathbb{ACL}$ there is precisely one additive morphism $f^\times : L_1 \otimes \cdots \otimes L_k \to L$ such that $f^\times \circ \text{cross}_\times = f$. The morphism $f^\times$ is called the extension of $f$.

It should be clear that similar definitions may be given by using other entities for $\mathbb{ACL}$, $\times$ and additivity. In this way one may define a tensor product $\otimes$ in $\mathbb{ACL}_{\otimes}$ for $\times$ with respect to additivity. The inclusion is written $\text{cross}_{\otimes}$ and the extension of $f$ as $f^\otimes$. In this section the theory will be developed for both $\otimes$ and $\otimes$ but the presentation will focus upon $\otimes$. The index $\times$ to cross will be omitted when confusion is not likely to result.

**Example** The smash product $\ast$ may be viewed as the tensor product in $\mathbb{CPO}$ of $\times$ with respect to strictness. The inclusion is smash and the extension of $f$ is the function $f$ restricted to the domain $L_1 \ast \ldots \ast L_k$. This amounts to saying that for every continuous and separately strict $f : L_1 \times \cdots \times L_k \to L$ there is precisely one strict and continuous function $f^\ast : L_1 \ast \cdots \ast L_k \to L$ such that

$$\begin{array}{ccc}
L_1 \times \cdots \times L_k & \xrightarrow{\text{smash}} & L_1 \\
\downarrow f & & \downarrow f^\ast \\
L_1 \ast \cdots \ast L_k & \xrightarrow{\ast} & L
\end{array}$$
The two major questions to be addressed are whether a tensor product $\otimes$ exists and if more than one tensor product exists then what is their relationship. The second question is answered by:

**Fact 3.2.4.** Let $\otimes$ be a tensor product with inclusion $\bigtimes^\chi$ and extension $f^\chi$. Similarly $\otimes'$, $\bigtimes^\chi'$, and $f'^\chi$. Then for all $L_1$, \ldots, $L_k$ there is an isomorphism

$$\theta: L_1 \otimes \ldots \otimes L_k \rightarrow L_1 \otimes' \ldots \otimes' L_k$$

such that $\theta \cdot \bigtimes^\chi = \bigtimes^\chi'$, and $f^\chi \cdot \theta^{-1} = f'^\chi$. It is given by

$$\theta = (\bigtimes^\chi')^\chi$$

The proof is similar to the one showing that two limiting cones are isomorphic.

### Construction

To show that a tensor product always exists we give a particular construction that uses the algebraicity of objects. The idea /PloPS/ is first to construct a certain logic, then obtain a countable and pointed quasi order from it and finally define the tensor product as the ideal completion.

Let $L_1$, \ldots, $L_k$ be ACL objects and note that all of $B_{L_1}, \ldots, B_{L_k}$ are semi-lattices with a least element. The logic $T_{\chi}$ is now constructed using ordinary logical notions /Men63/. It is convenient to formulate it as a $k+1$ sorted logic, where the sorts are $L_1, \ldots, L_k$ and $L_0$ for the tensor product. The constant terms are $b_i$ (or $b_i : L_i$) of sort $L_i$ whenever $b_i$ is a finite element of $L_i$. The function symbols

104
are $\otimes: L_1 \times \ldots \times L_k \rightarrow L_0$ and $\omega: L_0 \times L_0 \rightarrow L_0$. For formulae there is only one predicate $=\,$ that takes two terms of sort $L_0$ as arguments. These are subject to the following

axioms 1. \( \vdash x = x \)
2. \( \vdash x \omega x = x \)
3. \( \vdash x \omega y = y \omega x \)
4. \( \vdash x \omega (y \omega z) = (x \omega y) \omega z \)
5. \( \vdash \otimes(b_1, \ldots, b_i, \ldots, b_k) \omega \otimes(b_1, \ldots, b'_1, \ldots, b_k) = \otimes(b_1, \ldots, b_i \omega b'_i, \ldots, b_k) \)

inference rules

1. \( \vdash x = y \)
   \( \vdash y = x \)
2. \( \vdash x = y \quad \vdash y = z \)
   \( \vdash x = z \)
3. \( \vdash x = y \)
   \( \vdash x \omega z = y \omega z \)

Axiom 1 and the inference rules express that $=\,$ is a congruence. Axioms 2, 3 and 4 give properties that hold for a binary least upper bound operator. Axiom 5 expresses that $\otimes$ is separately additive. The convention is used that $b_i \omega b'_i$ is the constant term $b'_i$ it equals rather than assuming $\omega$ is a function $L_1 \times L_1 \rightarrow L_1$. Generally $t$ will denote a term of sort $L_0$ and \( \vdash t_1 = t_2, \ldots, t_{n-1} = t_n \) may be abbreviated to \( \vdash t_1 = \ldots = t_n \). The logic $T^\ast$ is constructed similarly but with function symbol $\otimes$ instead of $\otimes$ and with the additional

axiom 6. \( \vdash \otimes(b_1, \ldots, b_1, \ldots, b_k) = \otimes(b_1, \ldots, b_k) \)
Proofs in one logic can sometimes be transformed to proofs in another. Suppose that $T_{X'}$ is constructed similarly to $T_X$ but from objects $L_1', ..., L_k'$.

**Lemma 3.2.5.** If $r_i : B_i \to B_i'$ are additive functions then $\frac{t_1 t_2}{T_X}$ implies $\frac{t_1 [r = t_2]^r}{T_X}$ where $[r]$ is defined by $(t_1 \cdot t_2)^r = t_1 ^r \cdot t_2 ^r$ and $\Delta(b_1, ..., b_k)^r = \Delta(r_1(b_1), ..., r_k(b_k))$.

The similar result for $T_{X'}$ also needs $r_i$ to be strict. The proof of the lemma is formally by induction on the proof $\frac{t_1 = t_2}{T_X}$. Informally one must show that the axioms of $T_{X'}$ still hold in $T_{X'}$ when transformed by $[r]$ and that the transformed inference rules are derived rules in $T_{X'}$. We omit the details.

The countable and pointed quasi order $P_X (P_{X'})$ is now constructed from the logic $T_X (T_{X'})$. The countable set of elements is the set of terms of sort $L_0$ and $\Delta(L, ..., L)$ ($\Delta(L, ..., L)$) will be the least element. The quasi order is defined by $t_1 \preceq t_2$ iff $\frac{t_1 \cdot t_2}{T_X}$. That this gives a quasi order is straightforward using axioms 2 and 4. It is not a partial order as two syntactically different terms may well be proved equal. Because of axiom 4 we may omit parentheses around terms so any term is of the form $\Delta(b_1, ..., b_k) u ... u \Delta(h_1, ..., b_k)$. That the designated element is least now follows from the following two results.

**Lemma 3.2.6.** The term $t_1 \cdot t_2$ is a least upper bound of terms $t_1$ and $t_2$ with respect to $\preceq$.

**Proof** That $t_2 \preceq t_1 \cdot t_2$ is by $\frac{t_2 \cdot (t_1 \cdot t_2)}{T_X}$ and that $t_1 \preceq t_1 \cdot t_2$ is similar. If $t_1 \preceq t_2$ and $t_2 \preceq t_2$ then $t_1 \cdot t_2 \preceq t_2$ is because
Lemma 3.2: \( \forall (b_1, \ldots, b_k) \subseteq (b_1', \ldots, b_k') \) iff \( b_i \leq b_i' \) for all \( i \) (or, in the case of \( P^* \), if some \( b_j \) is \( \omega \)).

Proof For "if" it suffices to illustrate the case \( k=2 \):

\[
\vdash \forall (b_1, b_2) \subseteq (b_1', b_2') = \forall (b_1, b_2) \subseteq (b_1', b_2') = \forall (b_1', b_2')
\]

using axiom 5 several times. For "only if" we use a soundness result: a model specifies a set \( B \) (for \( L_0 \)) and functions \( \forall' \) and \( \Sigma' \) and the predicate \( ='. \) Consider the model where \( B \) is \( L_1 \), \( \forall' \) is \( \leq, \) \( \Sigma' \) is the binary least upper bound operation of \( L_1 \) and \( '=' \) is equality. Since \( \leq \) is (separately) additive all axioms are true and the inference rules preserve truth. Hence (by induction on the proof of) \( \vdash t_1 = t_2 \) implies \( [t_1] = [t_2] \), where \( [t] \) gives the value of a term in this model. So \( \forall (b_1, \ldots, b_k) \subseteq (b_1', \ldots, b_k') \) gives

\[
\vdash \forall (b_1, b_2) \subseteq (b_1', b_2') = \forall (b_1', b_2')
\]

so \( b_1 \leq b_1' \) and then \( b_1 \leq b_1' \) as was to be shown. In the case of \( P^* \) we use \( \forall'(b_1, \ldots, b_k) \) that is \( b_i \) if all \( b_j \leq \bot \) and \( \bot \) otherwise.

The "tensor product" \( L_1 \ldots \forall L_k \) is now defined to be \( P^X \) (where \( \ldots \) is the completion by ideals of section 3.1) and similarly \( L \ldots \forall L_k \) is \( P^X \). It follows from section 3.1 that these definitions give algebraic cpo's. That they are objects of \( ACL \) (ACLs) follows from lemmas 3.2:2, 3.2:6 and

Lemma 3.2:8. If a countable and pointed quasi order has least upper bounds of pairs then its completion by ideals is a semi-lattice.
Proof Define $J_1 \cup J_2 = \{ b : \exists b_1 \in J_1, b_2 \in J_2 : b \leq b_1 \land b \geq b_2 \}$ and note that this gives a directed ideal that is an upper bound of $J_1$ and $J_2$. Furthermore it is contained in any other upper bound that is also a directed ideal.

In a similar way one shows that the least upper bound operation in $L_1 \otimes \ldots \otimes L_k$ is

$$U_X = \{ \langle t, \ldots, t \rangle \cup \{ t \mid \exists n > 0 : t_1, \ldots, t_n \in \text{U} : t_1 \preceq u \ldots u t_n \} \}$$

and similarly for $L_1 \otimes \ldots \otimes L_k$.

To verify that the above construction gives a tensor product we must define the inclusion and extension of functions. Define

$$\text{cross}_x((l_1, \ldots, l_k)) =$$

$$\{ t \mid \exists b_1 \in B_1, \ldots, b_k \in B_k : (\forall i : b_i \leq l_i) \land t \in \text{O}(b_1, \ldots, b_k) \}$$

and similarly for $\otimes$. It is straightforward to verify that $\text{cross}_x$ and $\text{cross}_\otimes$ are strict and continuous functions that produce directed ideals.

Lemma 3.2: $\otimes_{x}$ and $\otimes_\otimes$ are separately additive. Proof Let $l_1', l_2' \in L_1$ and $b \in B_1$. Then there exists $b' \in B_1$ such that $b \leq l_1' \leq b'$. The proof is then straightforward using the formula for least upper bounds in $L_1 \otimes \ldots \otimes L_k$.

Similarly for $\otimes$.

Given an $ACL$ morphism $f : L_1 \otimes \ldots \otimes L_k \to L$ define $f^X$ by

$$f^X(J) = \bigcup \{ f(b_1, \ldots, b_k) \mid \otimes(b_1, \ldots, b_k) \in J \}$$

This defines a continuous function that satisfies

$$f^X(\text{cross}(l_1, \ldots, l_k)) = f(l_1, \ldots, l_k)$$

because by continuity it suffices to consider finite $l_i$ and for these the result is immediate. For an
ACLs morphism \( f: L_k \rightarrow L \) define \( f^k \) by

\[
f^k(J) = \bigcup \{ f(\text{smash}(b_1, \ldots, b_k)) \mid b_1, \ldots, b_k \in J \}
\]

This gives a strict and continuous function such that \( f^k \cdot \text{cross}^k = f \).

**Lemma 3.2:** \( f^k \) is additive if \( f \) is separately additive and similarly for \( f^k \).

**Proof** We calculate

\[
f^k(J \cup J') = 
\bigcup \{ f(b_1, \ldots, b_k) \mid \text{if } b_1, \ldots, b_k \in J \text{ then } f(b_1, \ldots, b_k) \in J \}
\]

(see below)

\[
\bigcup \{ f(b_1, \ldots, b_k) \mid b_1, \ldots, b_k \in J \}
\]

For the middle equality it is immediate that \( \Sigma \) holds and for \( \Sigma \) we shall use a soundness result as in the proof of 3.2:7. The model has \( B \) to be \( B_L \), \( \otimes \) to be \( f \), \( \wedge \) to be binary least upper bound in \( L \) and \( = \) is equality. It then follows that

\[
\text{if } b_1, \ldots, b_k \in J \text{ then } f(b_1, \ldots, b_k) \in J
\]

For the correctness of the construction now follows:

**Theorem 3.2:** The constructed \( L \otimes \cdots \otimes L_k \) is a tensor product with inclusion \( \text{cross}^k \) defined above and extension of \( f \) given by \( f^k \) above. Similarly for \( \otimes \).
Proof It remains to show that \( f^X \) is the unique continuous and additive function \( h \) such that \( h' \text{cross} = f \). So let \( h \) be some other candidate and show \( h(\mathbf{J}) = f^X(\mathbf{J}) \). By continuity it suffices to consider arguments \( \mathbf{J} \) of the form \( \{t \mid t \in \mathbf{T}'\} \) and by additivity one may assume \( t' = (b'_1, \ldots, b'_k) \). For such arguments the result follows from \( h' \text{cross} = f^X \text{cross} \).

The function \( \lambda L.1^X \) is continuous because it is an isomorphism from the continuous and separately additive functions \( L_1^X \times \cdots \times L_k^X \to L \) to the continuous and additive functions \( L_1 \otimes \cdots \otimes L_k \to L \). Similarly \( \lambda f.f^X \) is an isomorphism from strict, continuous and separately additive functions to completely additive functions.

**Extending it to a functor**

All domain constructors have been defined as functors and we now consider how to do this for tensor products. For \( \text{ACL} \) morphisms \( f_i : L_i \to M_i \) define the \( \text{ACL} \) morphism

\[
(f \otimes \cdots \otimes f_k)' = (\text{cross}_X f_1^X \cdots f_k^X)'^X
\]

This gives an additive (respectively strict) morphism if all the \( f_i \) are additive (respectively strict) because of 3.2:10 and \( \text{cross}'(f_1^X \cdots f_k^X) \) then is separately additive. Similarly for \( \text{ACLs} \) morphisms \( f_i : L_i \to M_i \) define the \( \text{ACLs} \) morphism

\[
(f \otimes \cdots \otimes f_k)' = (\text{cross}_X f_1^X \cdots f_k^X)'^X
\]

Again this gives an additive morphism if all the \( f_i \) are. Both \( \otimes \) and \( \otimes \) are locally continuous and covariant functors over \( \text{ACLs} \). This will be a consequence of a later result but may easily be proved directly using the following idea: since \( \times \) (and \( \times \)) are functors one may use 3.2:11 to show the functor laws in much the same way the result was shown for \( \varnothing \) in section 3.1. Neither of \( \otimes \) or \( \otimes \) gives a functor over.
ACLs, however, because the lack of additivity means that the law for composition may fail as is shown by the following example.

Example Let $L = P\{0,1,2\}$, $M = P\{-1,0,1\}$ and recall $V$ defined in section 3.1. Define $h: L \rightarrow V$ and $g: V \rightarrow M$ as the naming of the elements suggests, e.g. $h(\{1\}) = 1$, $h(\{1,2,3\}) = +$ etc. Consider $J = \text{cross}(\{1\}_R, \{0\}_R) \cup \text{cross}(\{2\}_R, \{1\}_R)$ in $\mathbb{LCL}$. Then

$$(h \circ h)(J) \circ (+, +) = \text{cross}(+, +)$$

so that

$$((g \circ g) \circ (h \circ h))(J) = \text{cross}(\{1\}_R, \{-1,0,1\})$$

This differs from

$$((g \circ h) \circ (g \circ h))(J) = \text{cross}(\{1\}_R, \{0\}_R) \cup \text{cross}(\{1\}_R, \{1\}_R)$$

$$= \text{cross}(\{1\}_R, \{0,1\})$$

Note that $g$ is not additive (but $h$ is).

For the purposes of chapter 4 it is convenient not to restrict consideration to ACLs. We therefore define a weaker notion than covariant functor and show that this includes $\otimes$ and $\circledast$. For morphisms in a sub cpo-category of $\mathbb{CPO}$ it has already been defined what additivity means. This may be extended to morphisms in a product category by defining such a morphism to be additive iff all its components are.

Definition Consider two cpo-categories with a notion of additivity defined as above. A (upper) semi-functor from one to another is a map upon objects and a map upon morphisms such that

1) $F(id) = id$

2) $F(f \circ f') \in F(f) \circ F(f')$
iii) $F(f)$ is additive if $f$ is additive

iv) equality holds in ii) when $f$ is additive

This definition includes semi-functors of more than one argument.

The functors $\ast$, $\otimes$, and $\Lambda$ are semi-functors over ACLs as follows from

Fact 3.2.12. Every covariant functor (over a cpo-category) that preserves additivity is a semi-functor.

Fact 3.2.13. Every semi-functor gives rise to a covariant functor on the subcategory of additionally additive morphisms.

Theorem 3.2.14. $\otimes$ is a locally continuous semi-functor over ACL and ACLs and $\otimes$ is over ACLs.

Proof We only consider $\otimes$ and the only non-trivial result is that

$$(f_1 \otimes \ldots \otimes f_k) \cdot (f'_1 \otimes \ldots \otimes f'_k) \exists (f_1, f'_1, \otimes(f_k, f'_k))$$

and that equality holds when all $f_i$ are additive. Abbreviating the inequality to $g' \otimes g$ we have

$$(g' \otimes g)(J) = g(\bigcup \text{cross}(f'_1(b_1), \ldots, f'_k(b_k)) \mid \otimes(b_1, \ldots, b_k) \in J)$$

$$\exists \bigcup g(\text{cross}(f'_1(b_1), \ldots, f'_k(b_k)) \mid \otimes(b_1, \ldots, b_k) \in J)$$

$$= \bigcup \text{cross}(f'_1(b_1), \ldots, f'_k(b_k)) \mid \otimes(b_1, \ldots, b_k) \in J$$

$$= g''(J)$$

If all $f_i$ are additive then so is $g$. Since $g$ is continuous and all objects are algebraic we have

$$g(\bigcup \chi) = \bigcup \{g(J) \mid J \in \chi\}$$

whenever $\chi$ is not empty. This shows that equality holds in the above calculation when all $f_i$ are additive.

It is straightforward to show that the composition of locally continuous semi-functors gives a locally continuous semi-functor and that
the tupling of locally continuous semi-functors gives a locally continuous semi-functor.

**Other Characterisations**

In the remainder of this section we shall study the tensor product of powerdomains, view the tensor product as a subcopo of a powerdomain, view the tensor product as a function space and study commutativity and associativity of $\mathcal{O}$ and $\mathcal{O}$.

**Powerdomains**

The motivation for studying the tensor product was to achieve that $\mathcal{P}(\{X_1, \ldots, X_k\})$ is isomorphic to $\mathcal{P}(A_1) \mathcal{O} \cdots \mathcal{O} \mathcal{P}(A_k)$. That this is the case follows from fact 3.2.4 and

**Theorem 3.2.15.** $\mathcal{P}(D_1^X \cdots D_k^X)$ with inclusion $\times^X = \lambda(I_1, \ldots, I_k) . \lambda^X D_1^X \cdots D_k^X$ is a tensor product of $\mathcal{P}(D_1), \ldots, \mathcal{P}(D_k)$

with respect to $X$ in $\text{ACL}$ and the extension of $f$ is given by

$$f^X = \lambda (f I_1, \ldots, f I_k) . \{(b_1, \ldots, b_k), (b_1, \ldots, b_k) e I\}$$

**Proof** Clearly $\times^X$ is continuous and separately additive and $f^X$ is continuous and additive and satisfies $f^X \times^X = f$ (because this is immediate for finite arguments). Any other candidate for $f^X$ equals $f^X$ on arguments of the form $\{(b_1, \ldots, b_k), (b_1, \ldots, b_k) e I\}$ and by additivity on all finite elements and by continuity always.

A similar result holds for $\times$. Here $\mathcal{P}(D_1^\times \cdots D_k^\times)$ is a tensor product (of $\times$ in $\text{ACL}$) with inclusion

$$\text{cross} \times = \lambda(I_1, \ldots, I_k) . \{(\text{smash} (b_1, \ldots, b_k), b_1 e I_1)\} .$$
and the extension $f^*$ of $f$ is defined similarly to $f^!$.

The isomorphism between $\rho(A_1 \times \ldots \times A_k)$ and $\rho(A_1) \otimes \ldots \otimes \rho(A_k)$ can be strengthened to include morphisms.

**Definition /ArMa75/.** A natural transformation from a functor $F : L \to K$ to a functor $G : L \to K$ is a family $\text{nat}$ of $K$ morphisms indexed by the objects of $L$ such that whenever $f : L \to L'$ is a morphism of $L$ the equality $G(f)' \text{nat}(L) = \text{nat}(L')'F(f)$ holds. This means that

\[
\begin{array}{ccc}
F(L) & \xrightarrow{\text{nat}(L)} & G(L) \\
\downarrow F(f) & & \downarrow G(f) \\
F(L') & \xrightarrow{\text{nat}(L')} & G(L')
\end{array}
\]

commutes. A natural equivalence is a natural transformation $\text{nat}$ where each $\text{nat}(L)$ is an isomorphism.

**Theorem 3.2:16.** There is a natural equivalence from $\rho^!(\text{cross}) : \text{Alg}_k \to \text{Acl}_k$ to $\rho^!(\rho_1, \ldots, \rho_k)$. Similarly for $\rho^!$ and $\otimes^!(\rho_1, \ldots, \rho_k)$. ///

**Proof** Let $\text{cross}$ and $f^X$ refer to the tensor product $\rho(D_1 \times \ldots \times D_k)$ and $\text{cross}'$ and $f^{X'}$ refer to $\rho(D_1) \otimes \ldots \otimes \rho(D_k)$. Define $\text{nat}(D_1, \ldots, D_k) = (\text{cross}')^X$ and note that fact 3.2:4 shows that this is an isomorphism. It is straightforward to show that

\[
\rho(f_1 \times \ldots \times f_k)(I) = (\text{cross}' \rho(f_1) \times \ldots \times \rho(f_k))^X(I)
\]

holds for morphisms $f_i : D_i \to D_i'$. It then follows that

\[
(\text{cross}')^X \rho(f_1, \ldots, f_k) \text{cross} \text{equals } \rho(f_1) \otimes \ldots \otimes \rho(f_k) (\text{cross}')^X \text{cross}
\]

and by the tensor product property of $\rho(D_1 \times \ldots \times D_k)$ this shows the result. ///
The tensor product as a subcpo of a powerdomain

We next study the quasi order in $P_x$ and $P_y$ with the aim of giving a different characterisation of it and thereby of the tensor product. The basic idea is to consider a function $\text{rep}$ that associates a term $t$ with the tuples $(b_1, \ldots, b_k)$ such that $\Theta(b_1, \ldots, b_k) \subseteq t$. The definition will be more "explicit", however, and the above description will follow from the next theorem. Define

$$\text{init}(t) = \text{LC}\{ (b_1, \ldots, b_k) \mid \Theta(b_1, \ldots, b_k) \text{ mentioned in } t \}$$

and note that this gives an ideal of $B_1 \times \ldots \times B_k$ hence an element of $\mathcal{P}(B_1 \times \ldots \times B_k)$. Next define

$$\text{step}_i(I) = \{ (b_1', \ldots, b_k') \mid \exists b_1', b_k' : b_i \in b_i' \land (b_1', \ldots, b_i', \ldots, b_k') \in I \}$$

$$\text{close}(I) = \text{LFP}(\lambda I. \text{step}(I_1'))$$

This defines continuous and extensive (e.g. $\text{step}(I) \subseteq I$) functions over $\mathcal{P}(L_1 \times \ldots \times L_k)$ and close is the least fixed point of step that contains $I$. Finally define

$$\text{rep}(t) = \text{close}(\text{init}(t))$$

For $\Theta$ the change is to replace $(b_1, \ldots, b_k)$ in init and step$_i$ by $\text{smash}(b_1, \ldots, b_k)$ so as to operate on $\mathcal{P}(L_1 \times \ldots \times L_k)$.

**Theorem 3.2:17.** $\vdash t_1 = t_2$ iff $\text{rep}(t_1) = \text{rep}(t_2)$  

**Proof** "Only if" is by induction on the proof of $\vdash t_1 = t_2$. The result is immediate for axioms 1, 2, 3, 4, (6) and for axiom 5 it suffices to observe that $\text{init}(t_1) \subseteq \text{init}(t_2)$ and $\text{step}(\text{init}(t_1)) \subseteq \text{init}(t_2)$.
It is straightforward to verify the result for inference rules 1 and 2 and for inference rule 3 note that

\[ \text{rep}(t u t') = \]
\[ \text{close}(\text{init}(t) u \text{init}(t')) = \]
\[ \text{close}(\text{rep}(t) v \text{rep}(t')) \]

For "if" it suffices to show that \( t \Theta(b_1, \ldots, b_k) u t_2 = t_2 \) holds for every \( \Theta(b_1, \ldots, b_k) \) mentioned in \( t_1 \). For then \( t_1 u t_2 = t_2 \) and by symmetry \( t_2 u t_1 = t_1 \) so that \( t_1 = t_2 \). If \( \Theta(b_1, \ldots, b_k) \) is mentioned in \( t_1 \) the tuple \( (b_1, \ldots, b_k) \) is an element of \( \text{rep}(t_1) \) and hence of \( \text{step}^n(\text{init}(t_2)) \) for some \( n \). It follows that it suffices to prove

\[ (b_1, \ldots, b_k) \in \text{step}^n(\text{init}(t_2)) \]
implies \( t \Theta(b_1, \ldots, b_k) u t_2 = t_2 \)

by induction on \( n \). The result is immediate for \( n = 0 \) so let \( (b_1, \ldots, b_k) \) be an element of \( \text{step}^{n+1}(\text{init}(t_2)) \). This means that there is \( i, b_i^1 \) and \( b_i^2 \) such that \( b_i^1 \in b_i \) and \( (b_1, \ldots, b_i^1, \ldots, b_k) \) and \( (b_1, \ldots, b_i^2, \ldots, b_k) \) are both elements of \( \text{step}^n(\text{init}(t_2)) \). The inductive hypothesis then gives

\[ t_2 = t_2 u \Theta(b_1, \ldots, b_i^1, \ldots, b_k) u \Theta(b_1, \ldots, b_i^2, \ldots, b_k) \]

and since

\[ t \Theta(b_1, \ldots, b_i^1, \ldots, b_k) u \Theta(b_1, \ldots, b_i^2, \ldots, b_k) \]
\[ = \Theta(b_1, \ldots, b_i^1, \ldots, b_k) u \Theta(b_1, \ldots, b_i^2, \ldots, b_k) u \Theta(b_1, \ldots, b_i^2, \ldots, b_k) \]

it follows that \( t_2 = t_2 u \Theta(b_1, \ldots, b_i^1, \ldots, b_k) \) as was to be shown.

(For \( \Theta \) one must take special care with tuples \( (b_1, \ldots, b_k) \) where some but not all components are \( \bot \).)

Corollary 3.2:18. \( t_1 \in t_2 \) iff \( \text{rep}(t_1) \subseteq \text{rep}(t_2) \).

The proof uses that \( \text{rep}(t_1 u t_2) = \text{close}(\text{rep}(t_1) v \text{rep}(t_2)) \) and that
this equals \( \text{rep}(t_2) \) iff \( \text{rep}(t_1) \subseteq \text{rep}(t_2) \). For later reference note the formulae

\[
\text{rep}(\bigotimes(b_1, \ldots, b_k)) = \{(b'_1, \ldots, b'_k) \mid \forall i: b'_i \in b_i \}
\]

\[
\text{rep}(\bigodot(b_1, \ldots, b_k)) = \{ \text{smash}(b'_1, \ldots, b'_k) \mid \forall i: b'_i \in b_i \}
\]

This characterisation of the quasi orders in \( \mathcal{P} \mathcal{X} \) and \( \mathcal{P} \mathcal{X} \) is convenient for characterising the irreducible elements and the primes.

**Lemma 3.2:19.** \( \bigotimes(b_1, \ldots, b_k) \) is irreducible iff all \( b_i \) are (or, in the case of \( \bigotimes \), if some \( b_i \) is \( \bot \)). 

**Proof** "Only if" is straightforward using 3.2:7 so consider "if" for \( \mathcal{P} \mathcal{X} \). Let all \( b_i \) be irreducible and suppose for the sake of contradiction that \( \bigotimes(b_1, \ldots, b_k) \) is not. Then there are \( t_1, t_2 \) such that

\[
\mathcal{F}(b_1, \ldots, b_k) = t_1 \lor t_2 
\]

but neither \( \mathcal{F}(b_1, \ldots, b_k) \) nor \( \mathcal{F}(b_1, \ldots, b_k) = t_1 \) nor \( \mathcal{F}(b_1, \ldots, b_k) = t_2 \). From the previous theorem \( (b_1, \ldots, b_k) \) is neither an element of \( \text{rep}(t_1) \) nor \( \text{rep}(t_2) \) but is an element of \( \text{rep}(t_1 \lor t_2) = \text{close}(\text{rep}(t_1) \cup \text{rep}(t_2)) \). For the contradiction we prove by induction on \( n \) that \( (b_1, \ldots, b_k) \) is not an element of \( \text{step}^n(\text{rep}(t_1 \lor \text{rep}(t_2)) \). The base case is immediate so suppose \( b_i \in b_i \land b_i \) with \( (b_1, \ldots, b_i, \ldots, b_k) \) and \( (b_1, \ldots, b_i, \ldots, b_k) \) elements of \( \text{step}^n(\text{rep}(t_1 \lor \text{rep}(t_2)) \). Since the latter set is a subset of \( \text{rep}(t_1 \lor t_2) = \text{LC}(\{(b_1, \ldots, b_k)\}) \) it follows that \( b_i \land b_i \) and hence \( b_i = b_i \). By \( b_i \) irreducible this contradicts the inductive hypothesis and therefore the proof of the inductive step is complete. The proof of "if" for \( \mathcal{P} \mathcal{X} \) is similar.///

**Lemma 3.2:20.** \( \bigodot(b_1, \ldots, b_k) \) is a prime iff all \( b_i \) are (or, in the case of \( \bigotimes \), if some \( b_i \) is \( \bot \)). 

**Proof** "Only if" is straightforward using 3.2:7. "If" may be proved by adapting the proof of the previous lemma or by a soundness result as
in the proof of 3.2:7: the model has $B=\{T\}_L$ and $\varnothing'(b_1', \ldots, b_k')=T$
iff all $b'_1 \sqsubseteq b_1$ (or, in the case of $\varnothing$, if some $b_1$ is $\bot$). That axiom 5 holds is because the $b_1$ are primes.

From this lemma it immediately follows that a tensor product of powerdomains is (isomorphic to) a powerdomain. For by theorem 3.1: 5 an object $L$ of $\Lambda CL_{\equiv}$ is a powerdomain iff $PB_L$ generates $B_L$, i.e. $B_L = \{b_1 \cdot \ldots \cdot b_n \mid n>0 \land b_i \in PB_L\}$, and the lemma shows that the finite elements $\{t \mid t \in t\}$ of the tensor product fulfills this condition. In section 4.5 it will be of interest to know that lemma 3:2.19 implies that the tensor product preserves the property that $IB_L$ generates $B_L$.

Another application of theorem 3.2:17 is to give a pictorial representation of a tensor product $L_1 \otimes \ldots \otimes L_k$. The method illustrated in the example below works whenever $L_i$ are complete lattices of finite cardinality such that $IB_{L_i}$ is a flat cpo that generates $B_{L_i}$.

Example The five element complete lattice $V$ of section 3.1 is the smallest complete lattice that is not (isomorphic to) a powerdomain.

The set $A_V = IB_V \setminus \{\bot\}$ contains only incomparable elements (i.e., if $b$ and $b'$ are elements then $b \sqsubseteq b'$ implies $b=b'$) and furthermore for every element of $V$ there is a subset of $A_V$ whose least upper bound equals that element.

The elements $\{t', t''\}$ of $V \otimes V$ can be uniquely represented by $\text{rep}(t) \cap A_V \times A_V$. Such a representation may be depicted by black squares on a $3 \times 3$ chess board. As an example $t=\varnothing(0,+)$ gives

```
  0 1 +
  + 1 .
+ . .
```
The partial order on \( V \oplus V \) gives rise to the following ordering of chess boards: \( \text{cb}_1 \leq \text{cb}_2 \) iff each black square of \( \text{cb}_1 \) is also black in \( \text{cb}_2 \). The function step may be defined on such chess boards. This implies that the chess board for \( \text{rep}(t)\text{init}(A) \) may be obtained from that for \( \text{init}(t)\text{init}(A) \) by repeated application of this function. The analogue of step operates upon a row as follows. Let \( H \) be the set of elements corresponding to black squares. For every element \( b \) in \( A \) such that \( b \in H \) the square it corresponds to is painted black. As an example

Based on these considerations it is straightforward, but tedious, to prove that \( V \oplus V \) is isomorphic to the partial order depicted below. Counting from the bottom the elements are divided into levels 0 to 4. A chess board is on level \( i \) iff \( i \) is the minimal number of black squares so that the chess board can be obtained by repeated application of step to a chess board with that many black squares. The \( T \) shaped chess board above therefore is on level 3. On level 2

abbreviates all solutions to the 2 rook problem on a 3x3 chess board, i.e. all ways of placing two rooks so neither captures the other. Similarly for the 3 rook problem on level 3.

Between two adjacent levels there should be lines that represent the partial order. The partial order satisfies that if \( x \) and \( y \) are different chess boards such that \( x \preceq y \) then the level of \( x \) is strictly less than the level of \( y \).
The function rep was defined as a map from the terms of the logic \(T^X\) to \(\mathcal{P}(L_1^X \ldots \times L_k^X)\). We now extend this to show that \(L_1^\otimes \ldots \otimes L_k^\otimes\) is embedded in \(\mathcal{P}(L_1^X \ldots \times L_k^X)\). A similar development can be performed for \(\otimes\) and \(\#\) as well. Define

\[
\text{Fix}_{\text{step}} = \{ \{ I \in \mathcal{P}(L_1^X \ldots \times L_k^X) \mid \text{step}(I) = I \} \}, \leq
\]

and note that by /Tar55/ this is a complete lattice.

**Theorem 3.2.21.** \(L_1^\otimes \ldots \otimes L_k^\otimes\) is isomorphic to \(\text{Fix}_{\text{step}}\).

**Proof** We will explicitly construct the isomorphism in both directions. Define \(\Theta : L_1^\otimes \ldots \otimes L_k^\otimes \to \text{Fix}_{\text{step}} \) by \(\Theta(J) = \bigcup \{ \text{rep}(t) \mid t \in J \}\) and note that this specifies a monotonic function into \(\mathcal{P}(L_1^X \ldots \times L_k^X)\). To see this is a fixed point of step consider \((b_1, \ldots, b_k) \in \text{step}(\Theta(J))\). There is \(i\) and \(t', t'' \in J\) and \(b_i^1, b_i^2\) such that \(b_i^1 \leq b_i^2 \leq \text{rep}(t)^1\) and \((b_1, \ldots, b_i^1, \ldots, b_k) \leq \text{rep}(t')\) and similarly for \(b_i^2\) and \(t''\). By \(J\) a directed ideal it contains \(t' \wedge t''\) and \((b_1, \ldots, b_i^1, \ldots, b_k) \leq \text{rep}(t' \wedge t'')\). This shows that
step(θ(J)) ≤ θ(J) and the other inclusion is immediate.

In the other direction θ⁻¹: Fix_{step} L_1 \odot ... \odot L_k is defined as a monotone function by

$$\theta^{-1}(I) = \left\{ (b_1^1, ..., b_k^1), ..., (b_1^n, ..., b_k^n) \mid n > 0 \land (b_1^i, ..., b_k^i) \in I \right\}$$

This is a directed ideal because I is a fixed point of step. It then remains to show that θ⁻¹θ = id and θ⁻¹φ = id. That θ⁻¹(I) ≥ I is immediate. If (b_1, ..., b_k) ∈ θ(θ⁻¹(I)) then there is (b_1^i, ..., b_k^i) ∈ I such that

$$(b_1, ..., b_k) \in close(\{(b_1^1, ..., b_k^1), ..., (b_1^n, ..., b_k^n)\})$$

Since I is a fixed point of step it follows that (b_1, ..., b_k) ∈ I. That θ⁻¹(θ(J)) ≥ J is immediate. If t ∈ θ⁻¹(θ(J)) then init(t) ≤ θ(J).

The set \{rep(t) | t ∈ J\} is directed and since init(t) is the left-closure of a finite set it follows that there is a term t' ∈ J such that init(t) ≤ rep(t'). This shows that t ∈ t' and therefore t ∈ J. //

The isomorphism θ defined in the above proof satisfies that

$$\theta(b_1, ..., b_k) ∈ J \iff (b_1, ..., b_k) ∈ θ(J).$$

Using this and fact 3.2.4 one can show that Fix_{step} is a tensor product with inclusion

$$cross(l_1, ..., l_k) = \left\{ (l_1, ..., l_k) \right\}_{R}$$

and extension of a separately additive function f given by

$$f^x(I) = f^I(I)$$

The definition as a functor upon additive functions f_i is

$$(f_1 \odot ... \odot f_k)(I) = close\left( \bigcup \left\{ f_i(b_1), ..., f_k(b_k) \right\}_R \mid (b_1, ..., b_k) \in I \right\}$$

This follows because the least upper bound of a nonempty set X is $U \odot \delta(l_1 \times ... \times l_k)$ and close(UX) in Fix_{step}. If X is a directed
set then close may be omitted because $U\chi$ is already a fixed point of step. This shows that

$$\theta : L_1 \odot \ldots \odot L_k \rightarrow \mathcal{P}(L_1 \chi \ldots \chi L_k)$$

is continuous and it follows that $\text{Fix}_{\text{step}}$ is a sub-cpo of $\mathcal{P}(L_1 \chi \ldots \chi L_k)$, i.e. that it has the same least element and the same least upper bounds of chains.

Note: For the similar results for $\otimes$ one should insert smash the "obvious" places.

The tensor product $\otimes$ as a function space

It is also possible to view the tensor product $\otimes$ as a certain function space. To formulate this define $D^{op} = (D, \Sigma)$ whenever $D = (D, E)$. Recall that $A \cong B$ means that $A$ and $B$ are isomorphic (as objects of CPO) and that $\rightarrow_{as}$ means the space of completely additive functions.

**Theorem 3.2.22.** $L_1 \odot \ldots \odot L_k \cong (L_1 \rightarrow_{as} \ldots \rightarrow_{as} L_{k-1} \rightarrow_{as} (L_k^{op})^{op} \quad 

**Proof** We prove the isomorphism in two steps. First note that

$$(L_1 \rightarrow_{as} \ldots \rightarrow_{as} (L_k^{op})^{op})^{op} \cong (B_{L_1 \rightarrow_{as} \ldots \rightarrow_{as} (L_k^{op})})^{op}$$

where $\rightarrow_{as}$ on the righthand side means completely additive functions from a semi-lattice to a complete lattice. When $k=2$ the isomorphism from right to left is

$$\lambda \xi. \lambda \eta. \Pi \{ \xi(b) \mid b \in B_{L_1 \wedge \eta} \}$$

Throughout this proof all $\xi$ and $\Pi$ refer to the $L_1$ so $\Pi$ is the least upper bound operator on $L_k^{op}$. 

122
Define $\theta: L_1 \otimes \cdots \otimes L_k \rightarrow (B_1 \rightarrow \cdots \rightarrow (L_k^{\text{op}}))^{\text{op}}$ by

$$\theta(J) = \lambda b_1, \ldots, \lambda b_{k-1}, \bigcup b_k \theta(b_1, \ldots, b_k) \in J \}.$$  

We shall assume $k=2$ below as this simplifies the notation but this is not otherwise important. First we must show that $\theta(J)$ is completely additive. Strictness is immediate as $\theta(J)(\lambda) = \lambda$ which is the least element of $L_2^{\text{op}}$. Additivity amounts to showing

$$\theta(J)(b'b) = \theta(J)(b') \cap \theta(J)(b'').$$

It is immediate to show $\theta$ because $b' \in J$ and $\theta(b'', b_2) \in J$ implies that $\theta(b', b_2) \in J$ so $\theta(J)(b') \cap \theta(J)(b'')$. For $\lambda$ the algebraicity of $L_k$ ensures that if suffices to show for $b_k \in B_k$ that $b_k \in \theta(J)(b') \cap \theta(J)(b'')$ implies $b_k \in \theta(J)(b' \cup b'')$.

The set $\{ b_k \mid \theta(b', b_k) \in J \}$ is directed for all choices of finite elements $b_2$ so for a finite element $b_k$ the condition $b_k \in \theta(J)(b_k)$ is equivalent to $\theta(b', b_k) \in J$. The result then follows because $\theta(b' \cup b'', b_k) \in J$ whenever $\theta(b', b_k)$ and $\theta(b'', b_k)$ are elements of $J$. Finally, $\theta$ is monotonic because $J_1 \subseteq J_2$ implies that $\theta(J_1)(b) \subseteq \theta(J_2)(b)$ holds in $L_k$.

In the other direction define

$$\theta^{-1}(f) = \{ \theta(b_1, \ldots, b_k) \mid n > 0 \forall \text{fin}: \exists b_k \in f(b_1, \ldots, b_{k-1}) \}.$$  

This is a directed ideal: it is clearly nonempty and it is left-closed because $f: B_1 \rightarrow \cdots \rightarrow L_k^{\text{op}}$ is monotonic. The function $\theta^{-1}$ is monotonic because $\forall b_1: f(b_1, \in) \in \theta^{-1}(f(b_1))$ implies that $\theta^{-1}(f) \subseteq \theta^{-1}(f')$.

It remains to verify that $\theta$ is a bijection with inverse $\theta^{-1}$.

That $\theta^{-1}(\theta(J)) \supseteq J$ is straightforward as both sides are directed ideals so it suffices to consider terms of the form $\theta(b_1, \ldots, b_k)$.
The converse inclusion follows because $\otimes(b_1,\ldots,b_k)$ is an element of $J$ iff $b_k \in \Theta(J)(b_1)\ldots(b_{k-1})$ as was remarked above for $k=2$. That $\Theta(\Theta^{-1}(f))\beta f$ is straightforward using the algebraicity of $L_k$ because if $b_k \in \Theta(f)(b_1)\ldots(b_{k-1})$ then $\otimes(b_1,\ldots,b_k) \in \Theta^{-1}(f)$ so that $b_k \in \Theta(\Theta^{-1}(f))(b_1)\ldots(b_{k-1})$. For the converse relation suppose that $b_k \in \Theta(\Theta^{-1}(f))(b_1)\ldots(b_{k-1})$ so that $\otimes(b_1,\ldots,b_k) \in \Theta^{-1}(f)$ and it follows that $b_k \in \Theta(f)(b_1)\ldots(b_{k-1})$.

For this formulation of the tensor product one may again use fact 3.2:4 to show that

$$\text{cross}(l_1,\ldots,l_k) = \lambda_{l_1}^l \ldots \lambda_{l_{k-1}}^l (\bigwedge_{j<k} l_j^l) \rightarrow l_k^l \perp$$

$$f = \lambda_q \bigcup f(l_1,\ldots,l_k) \mid l_k \in g(l_1)\ldots(l_{k-1}) \}$$

$$\otimes f_k \otimes \ldots \otimes f_k = \lambda_q \lambda_{l_1}^l \ldots \lambda_{l_{k-1}}^l$$

$$\bigcup \{ f_k \in g(l_1)\ldots(l_{k-1}) \mid \forall j<k: l_j^l \in g_j(l_j) \}$$

where $f$ is separately additive and strict, each $f_i$ additive and strict and $\bigcup$ and $\bigwedge$ refer to the $L_k$. It follows from the theorem that for a countable set $S$ we have the isomorphism

$$\mathcal{P}(S) \cong L$$

because both sides are isomorphic to $(S \rightarrow L^\text{op})^\text{op}$. This in turn might suggest a connection to the reduced cardinal power of /CoCo79/ (see example 10.2.0.2) but it remains to formally investigate the connection.

**Note** It is not clear how to obtain a "nice" analogue of the previous theorem for the tensor product $\otimes$. In the search the isomorphism

$$(L \otimes M)_\perp \cong L_\perp \otimes M_\perp$$

may possibly be of use.
Remark It turns out that the previous theorem is not novel (see e.g. /GHKLMS80,Ban80/). For another well known result let an adjunction from L to M be a pair of monotonic adjoined functions (as in the introduction) and let adjunctions be ordered as in CP02s.

Then (f,g) is an adjunction from L to M iff f is completely additive and g(m) = \bigcup \{1 \mid f(1) \leq m\} (see e.g. /CoCo79/). This shows that

$$L \otimes M \subseteq (\text{adjunctions from } L \text{ to } M^\text{op})^\text{op}$$

and by defining Galois connections as in /CoCo79/ it follows that

$$L \otimes M \subseteq (\text{Galois connections from } L \text{ to } M)$$

This result is stated in /Joh82/ for products of the so-called locales and is attributed to a 1979 paper by D. Wigner.

Commutativity and associativity

Finally, we comment upon the use of k-ary rather than just binary domain constructors. This is mainly a matter of convenience. It is well known that this is so for cartesian product and smash product as e.g. \(L_1 \times L_2 \times L_3\) is isomorphic to \((L_3 \times L_1) \times L_2\) and there is a similar close connection between \(f_1 \times f_2 \times f_3\) and \((f_3 \times f_1) \times f_2\). Formally there is a natural equivalence from \(x = x'(P_1, P_2, P_3)\) to \(x'(x'(P_3, P_1), P_2)\).

We shall, somewhat informally, phrase this as saying that \(x\) (as a family\(x^k\) of functors) is commutative and associative. This holds for \(y\) as well.

Theorem 3.2:23. \(\otimes\) is commutative and associative (as a family of functors) over ACLa and similarly \(\boxdot\) is over ACLas.

Proof We concentrate on \(\otimes\) and begin with defining some concepts. A list of terms over \(L_1, \ldots, L_k\) is a list \(g_1, \ldots, g_m\) of terms where
each $L_i$ occurs exactly once and $\emptyset$ is the only domain constructor used (possibly used with different arities). For $k=3$ examples include $L_1 L_2 \emptyset L_3$ and $L_3 \emptyset L_2$, $L_1$ but not $L_3, L_2 \emptyset L_2$. We shall not distinguish between the term and the object it denotes. Two lists $g_1, \ldots, g_m$ and $g'_1, \ldots, g'_m$, over $L_1, \ldots, L_k$ are similar iff

- for each object $L$ of ${\mathcal{ACL}}$ there is an isomorphism $\theta_L$ from the separately additive morphisms $g_1 \times \ldots \times g_m$ to the separately additive morphisms $g'_1 \times \ldots \times g'_m$.
- the isomorphisms satisfy that $h \cdot \theta_L(f) = \theta_L(h \cdot f)$ whenever $h: L \to L'$ is an additive morphism.

This concept is of interest because $L_1, \ldots, L_k$ and $g_t$ are similar iff $g_t$ is a tensor product of $L_1, \ldots, L_k$. The proof of "if" is straightforward so consider "only if". The claim is that $g_t$ is a tensor product with inclusion cross $\theta^{-1}_t(id)$ and $\theta_L(f)$ the extension of $f: L_1 \times \ldots \times L_k \to L$. To see that $f = \theta_L(f)^\times$ cross it suffices to show that

$$\theta_L(f) = \theta_L(f)^\times \theta_t(\theta_t^{-1}(id)) = \theta_L(\theta_L(f)^\times \theta_t^{-1}(id))$$

and the remaining conditions are straightforward.

The concept of similarity admits the following commutative property. If $p$ is a permutation of $1, \ldots, m$ then $g_1, \ldots, g_m$ and $g_t p(1), \ldots, g_t p(m)$ are clearly similar lists. For associativity we state the property that $g_1, \ldots, g_m$ is similar to $g_1 \emptyset \ldots \emptyset g_n, g_{n+1}, \ldots, g_m$. To sketch the proof it is convenient to assume $n=2$ and $m=3$ and define

$$\theta(f) = \lambda(x_{12} x_3) \cdot \lambda(x_1 x_2) \cdot f(x_1 x_2 x_3) x_{12}$$

It is straightforward to check that $\theta$ satisfies the conditions. If
\( \text{gt}_1 \) and \( \text{gt}_2 \) are any two "reorderings" of \( L_1 \ldots L_k \) then repeated application of the above rules show that \( \text{gt}_1 \) and \( \text{gt}_2 \) are similar. Hence both \( \text{gt}_1 \) and \( \text{gt}_2 \) are tensor product objects.

Each tensor product \( \text{gt}_i \) gives rise to a functor \( G_i : \text{ACL}_{\{1\ldots k\}} \rightarrow \text{ACL}_{\{1\ldots k\}} \) such that \( G_i(L_1, \ldots, L_k) = \text{gt}_i \). Since tensor products are isomorphic there is an isomorphism

\[
\text{nat}(L_1, \ldots, L_k) : G_1(L_1, \ldots, L_k) \rightarrow G_2(L_1, \ldots, L_k)
\]

As in the proof of 3.2:16 it follows that \( \text{nat} \) is a natural equivalence from \( G_1 \) to \( G_2 \).

The theorem can be extended to apply to \( \emptyset \) (and \( \emptyset \)) when viewed as semi-functors over \( \text{ACL}_{\{1\ldots k\}} \) (\( \text{ACL}_{\{1\ldots k\}} \)) but we shall not go into the proof of this. Also the functions take \( i \) and functional tuple \( \ldots \) can be shown to be related by the natural equivalence. This shows that also for tensor products a binary domain constructor would suffice for the entire development.

### 3.3 Collecting Interpretation

In this section the collecting interpretation \( C \) is defined and its properties are studied. The objects in the bottom-level are essentially the powerdomains of the corresponding objects in the standard interpretation \( S \). The functions in the bottom-level are essentially obtained by applying the powerdomain functor to the corresponding functions in \( S \). We begin with defining the type part of \( C \) and then formally prove a relationship with the type part of \( S \). Then the expression part is treated in a similar manner. The relationships are not quite as trivial as the previous overview might
suggest because the equality between objects / functions in $\mathbb{C}$ and $\mathbb{P}$ of objects / functions in $\mathbb{S}$ only hold for base types / constants of contravariantly pure type.

**THE TYPE PART**

The type part of $\mathbb{C}$ is defined by specifying the following three components:

- the category $\mathbb{C}(\mathbb{B})$ is $\mathbb{ACL}$ and $\mathbb{C}(\mathbb{B}_{\mathbb{P}})$ is $\mathbb{ACL}_{\mathbb{S}}$,
- the domain constructors are $\mathbb{C}(\mathbb{P})=\mathbb{0}$, $\mathbb{C}(\mathbb{X})=\mathbb{0}$, $\mathbb{C}(\mathbb{Y})=\mathbb{X}$ and $\mathbb{C}(\mathbb{L})=\mathbb{1}$,
- the constant functors are $\mathbb{C}(\mathbb{A}_{\mathbb{P}}) = \mathbb{0}$, $\mathbb{S}(\mathbb{A}_{\mathbb{B}})$

This definition satisfies the conditions required in section 2.3.

In particular $\mathbb{ACL}$ is a sub cpo-category of $\mathbb{CPO}$ and from lemma 3.2:1 it follows that $\mathbb{ACL}_{\mathbb{S}}$ is an admissible subcategory of $\mathbb{CPOs}$. It follows from theorem 3.2:14 and fact 3.2:13 that $\mathbb{0}$ and $\mathbb{0}$ are locally continuous and covariant functors over $\mathbb{ACL}_{\mathbb{S}}$ and it is immediate from section 2.2 that also $\mathbb{X}$ and $\mathbb{L}$ are.

The definition of the constant functors should come as no surprise. The use of $\mathbb{0}$ and $\mathbb{0}$ was motivated by theorem 3.2:16, namely that $\mathbb{P}(\mathbb{D}\times\mathbb{E})$ essentially is $\mathbb{P}(\mathbb{D})\mathbb{0}\mathbb{P}(\mathbb{E})$, and the proof of theorem 3.3:4 states similar results that motivate the use of $\mathbb{X}$ and $\mathbb{L}$. The choice of using $\mathbb{ACL}$ for $\mathbb{C}(\mathbb{B})$ will be discussed in the next paragraph. Given the choice, however, the use of $\mathbb{ACL}_{\mathbb{S}}$ for $\mathbb{C}(\mathbb{B}_{\mathbb{P}})$ is "forced". For $\mathbb{0}$ and $\mathbb{0}$ are not functors over $\mathbb{ACL}_{\mathbb{S}}$ but are when strictness and additivity are assumed.

The category $\mathbb{ACL}$ is not the only possible choice for $\mathbb{C}(\mathbb{B})$, another being the subcategory that is the image of $\mathbb{ACC}$ under $\mathbb{P}$. The use of $\mathbb{ACL}$ is motivated by section 4.1, where it will be argued that $\mathbb{ACL}$
is a natural category to use for abstract interpretation and indeed data flow analysis in general. In the introduction it was stated that the collecting semantics should be thought of as the most precise of all approximating interpretations. It is to facilitate this view that ACL is used in the collecting interpretation. It is a consequence of this decision that the tensor product of non power-domains is needed already in this chapter.

Remark Another possibility might be to use ACL for C(0) because, as is shown in section 4.1, this is a possible category for a certain class of approximating interpretations. To make this feasible it is essential that top-level objects are not required to be algebraic. This is because there exists a domain M such that the cpo M→M of continuous and additive functions is not algebraic.

Take M = \{⊥, T, 0, 1, ..., 5\} where x ≤ y if x = ⊥ or y = T or x = y. Then the identity id is neither finite nor the least upper bound of a chain of finite elements. To see that id is not finite consider the chain (g_n)_n defined by

\[ g_n(⊥) = ⊥ \]
\[ g_n(T) = T \]
\[ g_n(m) = (m ≤ n) → T, m+1 \]

and note that id ∉ ∪_n g_n even though each g_n is incomparable with id.

Next suppose by way of contradiction that id = ∪_n h_n for a chain (h_n)_n of finite elements. Then there exists n_0 such that n_0 ≥ n implies that h_n is the identity on ⊥, T, 0, 1. Since each h_n is strictly less than id the set \{m | h_n(m) = ⊥\} must be nonempty and in fact infinite for each n. When n ≥ n_0 and m and m' are chosen such that m ≠ m' and h_n(m) = ⊥ = h_n(m') this shows that h_n is not additive and the desired contradiction has
been established.  

To formulate and prove the relationship between the bottom-level domains of $S$ and $C$ it is helpful to state the following notations and facts concerning natural equivalences. Most of the notation makes sense for natural transformations in general.

**Fact 3.3:1.** A natural equivalence $\text{id}^F$ from the covariant functor $F$ to itself is defined by $\text{id}^F(L) = \text{id}_F(L)$. When $F$ is the identity functor we shall write this simply $\text{id}$. If $\text{nat}$ is a natural equivalence from $F$ to $G$ then $\text{nat}^{-1}$ defined by $\text{nat}^{-1}(L) = \text{nat}(L)^{-1}$ is from $G$ to $F$. If additionally $\text{nat}'$ is a natural equivalence from $G$ to $H$ then $\text{nat}' \cdot \text{nat}$ defined by $(\text{nat}' \cdot \text{nat})(L) = \text{nat}'(L) \cdot \text{nat}(L)$ is from $F$ to $H$.

**Fact 3.3:2.** If $\text{nat}_i$ are natural equivalences from $F_i$ to $G_i$ then $(\text{nat}_1, \ldots, \text{nat}_k)$ defined by

$$(\text{nat}_1, \ldots, \text{nat}_k)(L) = (\text{nat}_1(L), \ldots, \text{nat}_k(L))$$

is a natural equivalence from $(F_1, \ldots, F_k)$ to $(G_1, \ldots, G_k)$.

**Fact 3.3:3.** Let $\text{nat}_i$ be natural equivalences from $F_i$ to $G_i$ and $H$ and $H'$ covariant functors. Then $H[\text{nat}_i]$ defined by $H[\text{nat}_i](L) = H(\text{nat}_i(L))$ is a natural equivalence from $H'F_i$ to $H'G_i$ and $\text{nat}_i[H']$ defined by $\text{nat}_i[H'](L) = \text{nat}_i(H'(L))$ is from $F_i \cdot H'$ to $G_i \cdot H'$. Furthermore $\text{nat}_2[G_1] \cdot F_2[\text{nat}_1]$ is a natural equivalence from $F_2 \cdot F_1$ to $G_2 \cdot G_1$ and it equals $G_2[\text{nat}_1] \cdot \text{nat}_2[F_1]$.

The equality in the last fact is illustrated by

```
\[ 
\begin{array}{ccc}
F_2(F_1(L)) & \rightarrow & G_2(F_1(L)) \\
\downarrow & & \downarrow \\
F_2(\text{nat}_1(L)) & \rightarrow & G_2(\text{nat}_1(L)) \\
\end{array}
\]
```
and follows because nat₂ is a natural transformation. (In the terminology of category theory /ArMa75/ nat₂ ° nat₁ is the horizontal composition of nat₂ and nat₁ whereas nat₂[GT₁]° P₂[nat₁] is the vertical composition.)

The connection between the bottom-level domains of S and C can now be stated.

Theorem 3.3.4. Assume that V ⊨ gt, V ⊨ gt', card(V) = N and that X has index 1 in V. Then there is a natural equivalence nat_gt from the functor

\[ \varphi^S[gt] : \text{ACC}_N \rightarrow \text{AClas} \]

to

\[ C[gt']((\varphi^P_1, \ldots, \varphi^P_N)) \]

It satisfies that nat_gt[gt'/X] equals

\[ C[gt][(\text{nat}_{gt}, \text{id}[\varphi^P_2], \ldots, \text{id}[\varphi^P_N])] \cdot \text{nat}_{gt}[(C[gt'], P_2, \ldots, P_N)] \]

Recalling lemma 2.3.2 the latter equality may be illustrated as the commutativity of

\[ \varphi^S[gt[gt'/X]] = \varphi^S[gt']((\text{C}[gt'], P_2, \ldots, P_N)) \]

\[ \text{nat}_{gt[gt'/X]} \]

\[ \text{nat}_{gt}[(C[gt'], P_2, \ldots, P_N)] \]

\[ C[gt']((\varphi^S[gt'], \varphi^P_2, \ldots, \varphi^P_N)) \]

\[ C[gt][(\text{nat}_{gt}, \text{id}[\varphi^P_2], \ldots)] \]

\[ C[gt'[gt'/X]] \cdot \varphi^P = C[gt']((C[gt'], P_2, \ldots, P_N)) \cdot \varphi^P \]

where \( \varphi^P \) abbreviates \((\varphi^P_1, \ldots, \varphi^P_N)\). This result will be used later in lemma 3.3.11 and may be viewed as a companion to lemma 2.3.2.
Proof The proof of the theorem, and the construction of $\text{nat}_{gt}$, is by structural induction on $gt$.

Case $gt = A_i$: The natural equivalence $\text{nat}_{gt}$ will be chosen to be $\text{nat}_{gt}(D_1, \ldots, D_N) = \text{id}_\rho(S(A_i))$. This works because $\rho^{\ast} S(A_i) \cdot K\ NIL = \rho^{\ast} S(A_i) \cdot K\ NIL \cdot p = \rho^{\ast} D\ p$

For the desired equality between the natural equivalences one has

$\text{nat}_{gt}(D_1, \ldots, D_N) = \text{id}_\rho(S(A_i))$

and the result follows.

Case $gt = Y$. Assuming that the index of $Y$ in $V$ is $i$ the natural equivalence is $\text{nat}_{gt} = \text{id}[\rho^{\ast} P_i]$. Verification of the formula for $\text{nat}_{gt}[gt'/X]$ is straightforward: there are two cases depending on whether $X \subseteq Y$ or not.

Case $gt = gt_1 \ldots x gt_k$. Let $\text{nat}'$ be the natural equivalence from $\rho^{\ast} X$ to $\rho^{\ast}(\rho^{\ast} P_1, \ldots, \rho^{\ast} P_k)$ guaranteed by theorem 3.2:16. Then define $\text{nat}_{gt}$ as illustrated by

\[
\begin{align*}
\rho^{\ast} [gt] & = \rho^{\ast} X (S[gt_1], \ldots, S[gt_k]) \\
\text{nat}_{gt} & \downarrow \quad \text{nat}'[(S[gt_1], \ldots, S[gt_k])] \\
\rho^{\ast} (\rho^{\ast} [gt_1], \ldots, \rho^{\ast} [gt_k]) & \downarrow \quad \Theta[(\text{nat}_{gt_1}, \ldots, \text{nat}_{gt_k})] \\
\Theta[\rho^{\ast} D\ p] & = \Theta[\rho^{\ast} D_1\ p, \ldots, \rho^{\ast} D_k\ p] \\
i.e. \text{nat}_{gt} = \Theta[(\text{nat}_{gt_1}, \ldots, \text{nat}_{gt_k})] \cdot \text{nat}'[(S[gt_1], \ldots, S[gt_k])].
\end{align*}
\]
Verification of the formula for $\text{nat}_{\text{gt}[\text{gt}]/X}$ is a straightforward but tedious calculation using that $\emptyset$ is a functor over $\text{ACL}_\text{alc}$.

Case $\text{gt}=\text{gt}_1 \ldots \text{gt}_k$ is similar.

Case $\text{gt}=\text{gt}_1 + \ldots + \text{gt}_k$ is similar because a natural equivalence $\text{nat}'$ from $\emptyset'$ to $X'((\emptyset' P_1, \ldots, \emptyset' P_k)$ can be defined by

$$\text{nat}'(D_1, \ldots, D_k)(I) = (\text{out}_1(d) | d \epsilon I), \ldots, (\text{out}_k(d) | d \epsilon I)$$

Case $\text{gt} = \text{gt}_0 \perp$ is similar because a natural equivalence $\text{nat}'$ from $\emptyset'$ to $L'P$ can be defined by

$$\text{nat}'(D)(\{1'(D)\}) = \{1'(\emptyset(D))\}$$
$$\text{nat}'(D)(I) = \text{up}(\{\text{down}(d) | d \epsilon I\}) \text{ when } I \neq \{1'(D)\}$$

Case $\text{gt} = \text{rec} Y \cdot \text{gt}_0$. To define $\text{nat}_{\text{gt}}$ we shall assume that $Y$ is an element of $V$ and has index $N$. It is straightforward to adapt the definition if this is not so. Define the functors $F_{\text{gt}_0,n}$ and $G_{\text{gt}_0,n}$ inductively by

$$F_{\text{gt}_0,0} = \emptyset, \quad F_{\text{gt}_0,n+1} = \text{rec} Y \cdot F_{\text{gt}_0,n}$$
$$G_{\text{gt}_0,0} = \emptyset, \quad G_{\text{gt}_0,n+1} = \text{rec} Y \cdot G_{\text{gt}_0,n}$$

A natural equivalence $\text{NAT}_{\text{gt}_0,n}$ from $\emptyset' F_{\text{gt}_0,n}$ to $G_{\text{gt}_0,n} \cdot P$ is defined by

$$\text{NAT}_{\text{gt}_0,0}(D_1, \ldots, D_N) = \{P(U) \rightarrow U\}$$
$$\text{NAT}_{\text{gt}_0,n+1} = \text{rec} X \cdot \text{NAT}_{\text{gt}_0,n} \cdot \text{nat}_{\text{gt}_0} [(P_1, \ldots, F_{\text{gt}_0,n})]$$

Now consider $\text{rec} X (D_1, \ldots, D_N)$ which is the object in the limiting cone $(D, (r_n))$ for the chain

$$(F_{\text{gt}_0,n}(D_1, \ldots, D_N), (F_{\text{gt}_0,n}(\text{id}_{D_1}, \ldots, \text{id}_{D_N})))$$

as follows from the definition of $\text{REC}_N$ in section 2.2. Similarly
The definition

$$\text{nat}_{\rho}(D_1, \ldots, D_N) = \bigsqcup_n s_n \cdot \text{NAT}_{\rho_0, n}(D_1, \ldots, D_N) \cdot \rho(r_n)^U$$

clearly gives an isomorphism because each $\text{NAT}_{\rho_0, n}(D_1, \ldots, D_N)$ is an isomorphism.

To show that $\text{nat}_{\rho}$ is a natural equivalence note that

$$id_{\rho(D)} = \bigsqcup_n \rho(r_n)^U$$

follows from sections 2.2 and 3.2: see e.g. 2.2:2 and recall that $\rho$ is a locally continuous functor. Considering $f_1 : D_1 \to D'_1$ it therefore suffices to show that

$$\mathcal{C} \text{gt}\{1(D_1), \ldots\} \cdot \text{nat}_{\rho}(D_1, \ldots) \cdot \rho(r_n)^U$$

equals

$$\text{nat}_{\rho}(D_1, \ldots) \cdot \rho(\mathcal{C} \text{gt}\{1(D_1), \ldots\}) \cdot \rho(r_n)^U$$

for all values of $n$. Using the definitions of $\mathcal{C} \text{gt}\{1, 2\}$ and $\text{nat}_{\rho}$ lemmas 2.2:11 and 2.2:6 can be used to reduce this to the desired equality of

$$s_n \cdot \text{NAT}_{\rho_0, n}(\rho(F), \ldots) \cdot \text{NAT}_{\rho_0, n}(D_1, \ldots)$$

and

$$s_n \cdot \text{NAT}_{\rho_0, n}(D'_1, \ldots) \cdot \rho(\rho_{\rho_0, n}(F_1, \ldots))$$

This result is immediate, however, because $\text{NAT}_{\rho_0, n}$ is a natural equivalence from $\rho^* F_{\rho_0, n}$ to $G_{\rho_0, n} \cdot \rho^P$.

It remains to verify the formula for $\text{nat}_{\rho}[\rho'/X]$. If $X$ and $Y$ are the same variable then this is straightforward so assume otherwise.

We shall assume that $Y$ is not free in $\rho'$ (otherwise renaming could
achieve this). First we shall see that a similar formula holds for each \( \text{NAT}_{gt_0}[gt'/x], n \) namely that

\[
\text{NAT}_{gt_0}[gt'/x], n = \text{G}_{gt_0,n}[\text{n(at}_{gt}, \text{id}[P_2], \ldots, \text{id}[P_N]], n]
\]

This can be verified by induction on \( n \). In each case both sides are instantiated at \((D_1, \ldots, D_N)\) and the case \( n=0 \) is immediate. The proof in the inductive case is by a lengthy calculation which will be omitted except for mentioning that the following facts are used in the calculation:

- lemma 2.3:2 and \( \text{F}_{gt_0}[gt'/x], n = \text{F}_{gt_0,n}(\text{GT}[gt], P_2, \ldots, P_N) \)
- the formula for \( \text{n(at}_{gt}, /x) \)
- \( Y \) is not free in \( gt' \) so \( \text{GT}[gt], \text{GT}[gt] \) and \( \text{n(at}_{gt} \) do not depend on the \( N'\)th argument
- \( \text{GT}[gt] \) and \( \text{GT}[gt] \) are functors
- the inductive hypothesis in the induction on \( n \).

Turning to the formula for \( gt[gt'/x] \) we have the equality

\[
\text{n(at}_{gt}[gt'/x], (D_1, \ldots, D_N)) = \bigcup_n \text{n(at}_{gt_0}[gt'/x], n(D_1, \ldots, D_N)) \cup (r_n^U)
\]

Write \( L_1 = \emptyset(D_1) \) and assume \( N = 2 \) below. By lemma 2.3:2 and \( \text{G}_{gt_0}[gt'/x], n = \text{G}_{gt_0,n}(\text{GT}[gt], P_2, \ldots, P_N) \) the functionalities are

\[
s_n : \text{G}_{gt_0,n}(\text{GT}[gt], L_1, L_N) \rightarrow \text{GT}[\text{GT}[gt], (L_1, L_N), L_N]
\]

\[
r_n : \text{F}_{gt_0,n}(\text{GT}[gt], (D_1, D_N)) \rightarrow \text{GT}[\text{GT}[gt], (D_1, D_N), D_N]
\]

If similarly

\[
s_n' : \text{G}_{gt_0,n}(\text{GT}[gt], (D_1, D_N)) \rightarrow \text{GT}[\text{GT}[gt], (D_1, D_N), L_N]
\]

we have
\[ C_{gt} [(\text{nat}_{gt}, \text{id}[\varnothing], \ldots)] (D_1, \ldots, D_n) \]
\[ = \bigcup_n s_n \cdot G_{gt_0, n} (\text{nat}_{gt}, (D_1, \ldots, D_n), \text{id}[\varnothing], \ldots) \cdot s_n^U \]

and
\[ \text{nat}_{gt} [(\sigma_{gt'} \varnothing, \ldots)] (D_1, \ldots, D_n) \]
\[ = \bigcup_n s_n' \cdot \text{NAT}_{gt_0, n} (\sigma_{gt'} \varnothing, (D_1, \ldots, D_n), D_2, \ldots) \cdot \varnothing(r_n) \]

By the formula for \( \text{NAT}_{gt_0[gt'/\varnothing]} \), and lemma 2.2:6 the formula for \( \text{nat}_{gt[gt'/\varnothing]} \) follows. 

This theorem does not generalise to top-level types. This is immediate for a type \( t \) such that \( P(\varnothing, t) \) (see section 2.1) as \( \sigma_{gt'} \varnothing \)
then equals \( C_{gt} \varnothing \). Also in the case of \( gt \rightarrow gt' \) one has \( \varnothing(\sigma_{gt \rightarrow gt'}) \)
to be of the form \( \varnothing(X \rightarrow Y) \) whereas \( C_{gt \rightarrow gt'} \) is of the form \( \varnothing(X) \rightarrow \varnothing(Y) \).

In a sense \( \rightarrow \) is treated in an independent attribute method rather
than a relational method as will be discussed in chapter 6.

**Expression Part**

We now turn to defining the expression part of \( C \). The transformations strict, lin and view \( t \) that are used in the definition will be explained afterwards. The first of the three components specifies the functions

\[ C(\text{take}_i) = \lambda l. \text{id}^X(1) \cdot l \]
\[ C(\text{smashtake}_i) = \lambda l. \text{id}^X(1) \cdot l \]
\[ C(\text{in}_i) = \lambda l. (1, \ldots, 1) \]
\[ C(\text{up}) = \lambda l. (0, 1) \]
\[ C(\text{fold}) = \varnothing \quad -- \text{see below} \]
\[ C(\text{unfold}) = \varnothing^{-1} \quad -- \text{see below} \]

where \( \varnothing \) is the isomorphism from \( C_{gt[\text{rec}X \cdot gt/X]} \) to \( C_{\text{rec}X \cdot gt} \). (As was discussed in section 2.3 the intended \( gt \) is left implicit.)
The second component specifies the functionals

\[ C(\text{tuple}) (g_1, \ldots, g_k) = \text{lin}(\text{cross}_x^* \lambda l_k (g_1 (l_1), \ldots, g_k (l_k))) \]

\[ C(\text{smashtuple}) (g_1, \ldots, g_k) = \text{lin}(\text{cross}_x^* \text{smash}^* \lambda l_k (g_1 (l_1), \ldots, g_k (l_k))) \]

\[ C(\text{case}) (g_1, \ldots, g_k) = \lambda l_k . \text{g}_1 (l_1) \downarrow \ldots \downarrow \text{g}_k (l_k) \]

\[ C(\text{lift}) (g) = \lambda l . \text{def}(l) \rightarrow g(\text{down}(l))_l \]

\[ C(\text{cond}) (g_1, g_2, g_3) = \text{strict}(g_2)^* \text{filter}_{tt} (g_1) \cup \text{strict}(g_3)^* \text{filter}_{ff} (g_1) \]

where \( \text{filter}_x (g) = \text{lin}(\lambda l . \text{g}(l) \downarrow \{ x \})_{l R} \rightarrow l, L \)

\[ C(\emptyset) (g_1, g_2) = g_1^* g_2 \]

As an example \( C(\text{cond}) \) will be explained below. Finally, the third component defines

\[ C(f) = \text{view}_t (\emptyset) (S(f)) \]

for each constant \( f \) of contravariantly pure type \( t \).

The transformation \( \text{lin} \) is defined for any \( ACL \) morphism \( g : L \rightarrow M \) by the formula

\[ \text{lin}(g) = \lambda l . \bigcup \{ g(i) \mid i \in 1 \wedge i \in IB_L \} \]

which clearly gives a continuous function and depends continuously on \( g \). For the collecting interpretation one is especially interested in the case where \( L \) (and \( M \)) are powerdomains. Then

\[ \text{lin}(g) = \lambda l . \bigcup \{ g(i) \mid i \in 1 \wedge i \in IB_L \} = (g^* \lambda d . \{ d \})_R \]

follows from theorem 3.1:7. It is seen that \( \text{lin}(g) \) is an additive function that agrees with \( g \) upon singletons and it is this that motivates the use of \( \text{lin} \) (as will become clear when \( \text{filter}_x \) is explained below). Of these three characterisations of \( \text{lin} \) only the first two work for all \( ACL \) morphisms and the definition chosen is
the one that will be useful in section 4.4.

Next consider filter \( \lambda (g) \). It follows from the type considerations of section 2.1 that \( g \) will be an element of \( \mathcal{C}[g_t \rightarrow \mathcal{T}] \) for some \( g_t \), i.e. \( g : L \rightarrow \mathcal{O}(T) \) for some \( \text{ACL} \) object \( L \). By theorem 3.3:4 there will be a powerdomain \( \mathcal{P}(D) \) that is isomorphic to \( L \) so let us assume that \( L = \mathcal{P}(D) \). Then for \( x=\text{tt} \) or \( x=\text{ff} \) we have

\[
\text{filter}_x(g)(I) = \text{LC}(\{J\} \cup \{b \in I \mid g(\{b\}_R) \exists x\})
\]

and in the case where \( g = \mathcal{P}(f) \) this gives

\[
\text{filter}_x(\mathcal{P}(f))(I) = \text{LC}(\{J\} \cup \{b \in I \mid f(b) = x\})
\]

so that \( \text{filter}_x(g)(I) \) specifies the subset of \( I \) for which the first branch is appropriate and similarly for \( \text{filter}_x(g)(I) \). It is straightforward that \( \text{filter}_x(g) \) is continuous and is continuous in \( g \).

The conditional then executes each branch with the appropriate set of arguments. So the idea is that

\[
\mathcal{P}(S(\text{cond})(f_1, f_2, f_3)) = \mathcal{C}(\text{cond})(\mathcal{P}(f_1), \mathcal{P}(f_2), \mathcal{P}(f_3))
\]

which agrees with what was said in the introduction. For this to hold it is necessary to use the continuous function \( \text{strict} \) defined by

\[
\text{strict}(g) = \forall l. l = \bot \rightarrow \top, g(l)
\]

unless \( f_2 \) and \( f_3 \) are both strict. (This relates to the remark in section 2.3.) Clearly \( \text{strict}(g) \) is a strict and continuous function that agrees with \( g \) except possibly on \( \bot \).

Finally the transformation \( \text{view}_t() \) used for the third component must be specified. Consider as an example a function \( f: \mathcal{C}[gt \rightarrow gt'] \).
so that $f: \mathcal{S}[t] \to \mathcal{S}[t']$. Had it been the case that $\mathcal{S}[t] = \rho(\mathcal{S}[t])$ and likewise for $t'$ then the desired function would be

$$\rho(f) = \lambda t. LC\{ f(b) | b \in I \}$$

This is not quite the case but by theorem 3.3:4, and recalling that $\emptyset t - t$ and $\emptyset t' - t'$, we can use

$$v_{\mathcal{S}[t]}(f) = \text{nat}_{\mathcal{S}[t]}(f) \cdot \rho(f) \cdot \text{nat}_{\mathcal{S}[t]}^{-1}$$

instead. It is now necessary to extend this definition to all constants of contravariantly pure type. This will be accomplished by the continuous function

$$\text{view}_{\mathcal{S}[t]}() : \mathcal{S}[t] \to \mathcal{S}[t]$$

It will be defined structurally on $t$ so types will be met that are not contravariantly pure but only $\text{CP}(V, t)$ for some $V$ such that $V t$. When $\text{card}(V) = N$ and $h_i : D_i \to L_i$ are morphisms of $\text{CPO}$ we therefore have

$$\text{view}_{\mathcal{S}[t]}(h_1, \ldots, h_N) : \mathcal{S}[t](D_1, \ldots, D_N) \to \mathcal{S}[t](L_1, \ldots, L_N)$$

Clearly $\text{view}_{\mathcal{S}[t]}()$ is to be $v_{\mathcal{S}[t]}$ but it will sometimes be convenient to make this explicit by writing

$$\text{view}_{\mathcal{S}[t]}[(v_{\mathcal{S}[t]})_{\mathcal{S}[t]}](h_1, \ldots, h_N)$$

This makes it possible to perform the development in sufficient generality that it is not necessary to give a similar development in chapter 4.

The definition of $\text{view}_{\mathcal{S}[t]}$ closely resembles the definition of the functor $\mathcal{S}[t]$ in section 2.3 (see theorem 2.2:10 for recX.t). It is worth repeating that it is assumed that $V t$, $\text{CP}(V, t)$ and $\text{card}(V) = N$. 

139
That this definition makes sense is by a structural induction showing that $\text{view}_t \llbracket \text{vw} \rrbracket (h_1, \ldots, h_N)$ is continuous and depends continuously on $h_i$ (provided that $\text{vw}_f t$ and $h_i$ are all continuous). In the case $t_1 \to t_2$ lemma 2.3:3 is needed to show that $P(V, t_1) \wedge V \wedge t_1$ guarantees that $\sum_{t_1} \bigotimes (D_1, \ldots, D_N)$ equals $\bigotimes_{t_1} (L_1, \ldots, L_N)$. It is to achieve this that the constants are required to be of contravariantly pure type for when
If \( S[t_1] \) differs from \( C[t_1] \) it is not clear how to define \( \text{view}_{t_1 \to t_2} \). In the case \( \text{rec}X.t \) lemma 2.3:2 shows that the functionality is correct.

The resemblance of \( \text{view}_t[vw] \) to the functor \( [v] \) defined in section 2.3 can be formalised in the following way.

**Lemma 3.3:5.** Let \( t_1 \) be a type such that \( V \vdash t_1 \), \( CP(V,t) \) and \( \text{card}(V)=N \).
Let \( f_{t_1}, \ldots, f_{t_n} \) be all the \( ft \)-types mentioned in \( t \) and let \( t'=t_1[X_{N+1}/f_{t_i}] \) have each \( f_{t_i} \) replaced by \( X_{N+1} \). Then for all choices of \( \text{CPO} \) morphisms \( h_1, \ldots, h_N \) and \( q_{N+1} \) and for all interpretations \( \text{I} \) it is the case that

\[
\text{view}_t[vw](h_1, \ldots, h_N)
\]

equals

\[
\text{I}[t']((h_1, q_1), \ldots, (h_N, q_N), (vw_{f_{t_1}}, q_{N+1}), \ldots, (vw_{f_{t_n}}, q_{N+n})))
\]

**Proof** This is a structural induction and we sketch two cases. Case \( t=t_1 \to t_2 \) uses lemma 2.3:3 and that \( \Theta[t_1] \) is a functor to show that

\[
\text{I}[t']((h_1, q_1), \ldots, (vw_{f_{t_n}}, q_{N+n}))) = \text{id}
\]

Case \( t=\text{rec}X.t_0 \) uses lemma 2.2:10 in the case where \( P(V, \text{rec}X.t_0) \) fails and is straightforward in the case where it holds.

This result makes it immediate to prove the following property of \( \text{view}_t[vw] \). Call a function \( h \) **strongly strict** iff \( h(\text{v})=\text{v} \iff \text{v} \).

**Lemma 3.3:6.** If \( vw_{f_t} \) and \( h_1 \) are all strongly strict then so is

\[
\text{view}_t[vw](h_1, \ldots, h_N)
\]

**Proof** This is a structural induction and the only non-trivial case is \( t=\text{rec}X.t_0 \) when \( P(V, \text{rec}X.t_0) \) fails. From the previous lemma it is not
difficult to see that

$$\text{view}_{\text{recX}, t_0}^{[vw]}(h_1, \ldots, h_N) = \bigcup_n s_n \cdot (\lambda h.\text{view}_{t_0}^{[vw]}(h_1, \ldots, h_N, h))^n(\lambda) \cdot r_n$$

for appropriate embeddings $s_n$ and $r_n$ in CPOs. Since $\mathbb{I}: U \to U$ is strongly strict and it easily follows that also all

$$(\lambda h.\text{view}_{t_0}^{[vw]}(h_1, \ldots, h_N, h))^n(\lambda)$$

are we get that $\text{view}_{\text{recX}, t_0}^{[vw]}(h_1, \ldots, h_N)$ is strongly strict. ///

A consequence of this lemma is that the smash in $\text{view}_{t_1 \times \ldots \times t_k}$ is not always necessary. It will emerge shortly that strongly strictness is only of interest because of the presence of the domain constructor $\chi$ (see a remark in section 2.1).

Returning to the definition of the expression part of $C$ it can be seen that it satisfies the requirements of section 2.3: For $C(\text{in}_1)$ etc. in the first component this is a straightforward verification using lemma 2.3:2 for $C(\text{fold})$ and $C(\text{unfold})$. The functionals are clearly continuous and continuous in the $g_i$ arguments. Also the constants of contravariantly pure type are as required. In the remainder of this section we therefore consider the relationship between the semantics of expressions as given by $S$ and $C$.

**Relating the interpretations $S$ and $C$**

In the explanation of $C(\text{cond})$ it was hinted at the desired connection between the denotations for expressions in the standard and collecting semantics. The formal statement of this connection amounts to defining a predicate $\text{sim}_t()$ on $S[t] \times C[t]$. As for $\text{view}_t$ it is convenient to consider
\( \text{sim}_t \left( (\text{sm}_t^t, s_t^t) \right)(Q_1, \ldots, Q_N) \)

that is a predicate on \( \mathcal{S}(t)(D_1, \ldots, D_N) \times \mathcal{C}(t)(L_1, \ldots, L_N) \) whenever each
\( Q_i \) is a predicate on \( D_i \times L_i \) and \( \text{sm}_t^t \) on \( \mathcal{S}(t) \times \mathcal{C}(t) \). It is assumed
that \( \forall t \vdash t \) and \( \text{card}(V) = N. \)

The predicate is defined structurally much as was the case for
\( \text{view}_t \).

\( \text{sim}_{A_1} \left( \text{sm} \right)(Q_1, \ldots, Q_N)(v, w) \equiv v = w \)

\( \text{sim}_{t_1 \times \ldots \times t_k} \left( \text{sm} \right)(Q_1, \ldots, Q_N)(v, w) \equiv \)

\( \forall i: \text{sim}_{t_i} \left( \text{sm} \right)(Q_1, \ldots, Q_N)(v_i, w_i) \)

\( \text{sim}_{t_1 \times \ldots \times t_k} \left( \text{sm} \right)(Q_1, \ldots, Q_N)(v, w) \equiv \)

\( \forall i: \text{sim}_{t_i} \left( \text{sm} \right)(Q_1, \ldots, Q_N)(v_i, w_i) \)

\( \text{sim}_{t_1 + \ldots + t_k} \left( \text{sm} \right)(Q_1, \ldots, Q_N)(v, w) \equiv (v = w^t = 1) \lor \)

\( \exists i: v = (i, w') \wedge w = (i, w') \wedge \text{sim}_{t_i} \left( \text{sm} \right)(Q_1, \ldots, Q_N)(v', w') \)

\( \text{sim}_{t_1 \rightarrow t_2} \left( \text{sm} \right)(Q_1, \ldots, Q_N)(v, w) \equiv \)

\( \forall (v', w'): \text{sim}_{t_1} \left( \text{sm} \right)(Q_1, \ldots, Q_N)(v', w') \rightarrow \text{sim}_{t_2} \left( \text{sm} \right)(Q_1, \ldots, Q_N)(v(v'), w(w')) \)

\( \text{sim}_{\text{rec}X.t} \left( \text{sm} \right)(Q_1, \ldots, Q_N)(v, w) \equiv \)

\( \forall n: \text{SIM}_{t, n}[\text{sm}](Q_1, \ldots, Q_N)(r_n^U(v), s_n^U(w)) \)

where \( \text{SIM}_{t, 0}[\text{sm}](Q_1, \ldots, Q_N)(v', w') \) is true

\( \text{SIM}_{t, n+1}[\text{sm}](Q_1, \ldots, Q_N) = \)

\( \text{sim}_{t_i}[\text{sm}](Q_1, \ldots, Q_N) \text{SIM}_{t, n}[\text{sm}](Q_1, \ldots, Q_N) \)

and it is assumed that \( X \) has index \( N+1 \) in \( V \cup \{x\} \) and that \( r_n \) and
\( s_n \) are the CPO embeddings of the respective limiting cones

\( \text{sim}_X[\text{sm}](Q_1, \ldots, Q_N)(v, w) \equiv Q_i(v, w) \quad \text{if } X \text{ has index } i \text{ in } V \)

\( \text{sim}_{gt \rightarrow gt} \left( \text{sm} \right)(Q_1, \ldots, Q_N)(v, w) \equiv \text{sm}_{gt \rightarrow gt}(v, w) \)

143
The sm that is of interest in this section is

$$sm_{gt \rightarrow gt', (v, w)} = w=\text{nat}_{gt'}() \cdot \rho(v) \cdot \text{nat}_{gt}()^{-1}$$

It is a straightforward structural induction to prove that this definition gives rise to a predicate as stated. Furthermore it is convenient to state the following information about the predicate.

First an analogue of lemma 3.3:6:

**Lemma 3.3:7.** If each $sm'_{gt}$ and $Q_i$ is an admissible predicate then also

$$\text{sim}_{t[sm']}(Q_1, \ldots, Q_N)$$

is. If further $sm'_{gt}$ and $Q_i$ satisfies that

when true of $(v, w)$ then $v=\bot$ iff $w=\bot$ \hfill (x)

then also $\text{sim}_{t[sm']}(Q_1, \ldots, Q_N)$ does. \hfill ///

The proof is by structural induction with an additional numerical induction for the case $\text{recX.t}$. Both admissibility and (x) hold for $sm'_{gt}$ and as with "strongly strict" the property (x) is only of interest because of the domain constructor X. Next an analogue of lemma 2.3:2 is:

**Lemma 3.3:8.** If $V \vdash t$, $V \vdash t'$, $\text{card}(V)=N$ and $X$ has index $N$ in $V$ then

$$\text{sim}_{t[t'/X]}[sm'](Q_1, \ldots, Q_N) = \text{sim}_{t[sm']}(Q_1, \ldots, Q_{N-1}, \text{sim}_{t'[sm']}(Q_1, \ldots, Q_N))$$

This result is proved by induction as for the previous lemma. Finally an analogue of lemma 2.3:3 is:

**Lemma 3.3:9.** If $V \vdash t$, $P(V,t)$ and $\text{card}(V)=N$ then

$$\text{sim}_{t[sm']}(Q_1, \ldots, Q_N)(v, w) \text{ holds iff } v=w$$

///
\textbf{Proof} This is a corollary of the following result that is proved by structural induction on $t$.

Let $V\vdash t$, $P(V', t)$, $\text{card}(V) = N$, $\text{card}(V') = n$ be such that $V'$ is the subset of variables in $V$ with index at most $n$ in $V$. If $\text{eqv}(v, w)$ is defined by $v = w$ then $\text{sim}_t[sm'](Q_1, \ldots, Q_n, \text{eqv}, \ldots, \text{eqv}) = \text{eqv}$. We omit the proof. ///

The purpose of $\text{view}_t$ is to transform a function into one that is related to it by $\text{sim}_t$. That this holds is made precise in:

\textbf{Lemma 3.3:10.} Considering a contravariantly pure and closed type $t$ we have

$$\text{sim}_t(()(f, \text{view}_t(()(f)))$$

for every $f \in S[t]$. In general

$$\text{sim}_t[sm'](()(f, \text{view}_t[\text{vw'}]())(f))$$

holds provided that $sm'_{ft}(f', \text{vw'}_{ft}(f'))$ is guaranteed and all $\text{vw'}_{ft}$ are strongly strict. ///

\textbf{Proof} Clearly the explicitly defined $sm$ and $\text{vw}$ satisfy the conditions stated for $sm'$ and $\text{vw'}$. The lemma then follows from the more general result that

$V \vdash t$ and $\text{CP}(V, t)$ and $\text{card}(V) = N$ implies

$$\text{sim}_t[sm'](Q_1, \ldots, Q_N)(f, \text{view}_t[\text{vw'}](h_1, \ldots, h_N)(f))$$

whenever $sm'_{ft}(f', \text{vw'}_{ft}(f'))$ and $Q_i(f', h_i(f'))$ are guaranteed and $\text{vw'}_{ft}$ and $h_i$ are strongly strict.

This result is proved by structural induction on $t$ such that $V \vdash t$ and $\text{CP}(V, t)$.  

145
Case $t = \lambda_1$ is immediate.

Case $t = t_1 \ldots \ldots t_k$ is straightforward from the inductive hypotheses.

Case $t = t_1 \ldots \ldots t_k$ is as above because lemma 3.3:6 shows that smash has no effect.

Case $t = t_1 + \ldots + t_k$ and case $t = t_0 l$ are straightforward.

Case $t = t_1 \rightarrow t_2$ is straightforward because lemma 3.3:9 shows that $\text{sim}_{t_1} [\ldots] (...) (v', w')$ holds iff $v' = w'$.

Case $t = \text{rec}X. t_0$ There are two cases. If $P(V, \text{rec}X. t_0)$ holds then lemma 3.3:9 shows that $\text{sim}_{t_0} [\ldots] (...) (v', w')$ holds if $v' = w'$. Since $\text{view}_{t} [vw'](h_1, \ldots, h_N)(v) = v$ the result is immediate. Suppose next that $P(V, \text{rec}X. t_0)$ is false. The formulation of $\text{view}_{t}$ given in the proof of lemma 3.3:6 makes it convenient to abbreviate (assuming that $X$ has index $N+1$ in $V_x \{x\}$)

$$H(h) = \text{view}_{t_0} [vw'](h_1, \ldots, h_N, h)$$

It then follows that (e.g. by lemmas 3.3:5 and 2.2:11)

$$s_{n}^{U}(\text{view}_{t} [\ldots] (...) (v)) = H^{n}(\Lambda)(r_{n}^{U}(v))$$

Since $\Lambda : U \rightarrow U$ is strongly strict also $H^{n}(\Lambda)$ is and it suffices to prove that

$$\text{SIM}_{t_0 \cdot n^{[s[m]}}(Q_1, \ldots, Q_N)(r_{n}^{U}(v), H^{n}(\Lambda)(r_{n}^{U}(v)))$$

This is by induction on $n$ using the hypotheses of the structural induction.

Cases $t = X$ and $t = ft$ are immediate. ///

We are now ready to prove a relationship between the expression parts of the standard and collecting interpretations. Henceforth $[vw]$ and $[sm]$ will be left implicit for $\text{view}_{t}()$ and $\text{sim}_{t}()$. 
Lemma 3.3:11. For every entry $z$ of type $t$ in one of the components of the expression part of an interpretation we have $\text{sim}_t(S(z), C(z)) ///$

Proof There are three components in the expression part and by lemma 3.3:10 the result is immediate for the third component. For the other two components this is by explicit calculations. In these the parentheses are omitted from the natural equivalences $\text{nat}_{gt}$ because $\emptyset \vdash gt$.

The calculation for $\text{take}_i$ starts with

$$\text{nat}_{gt_i}^{-1} \cdot C(\text{take}_i) \cdot \text{nat}_{gt} \cdot x \ldots x g t_k$$

The definition of $\text{nat}$ in theorem 3.3:4 then gives

$$\text{nat}_{gt_i}^{-1} \cdot \downarrow i \cdot \text{id}^X \cdot \text{nat}_{gt} \cdot \otimes \ldots \otimes \text{nat}_{gt_k} \cdot \text{nat}'(S[gt_1], \ldots, S[gt_k])$$

and the definition of the functor $\otimes$ gives

$$\downarrow i \cdot (\text{nat}_{gt_1}^{-1} \cdot x \ldots \cdot \text{id}^X \cdot \text{cross} \cdot (\text{nat}_{gt} \cdot x \ldots) \cdot X \cdot \text{nat}'(S[gt_1], \ldots))$$

Since $f^X \cdot \text{cross} = f$ and $g \cdot f^X = (g \cdot f)^X$ whenever $g$ is additive this gives

$$\downarrow i \cdot ((\text{nat}_{gt_1}^{-1} \cdot x \ldots) \cdot (\text{nat}_{gt} \cdot x \ldots) \cdot X \cdot \text{nat}'(S[gt_1], \ldots))$$

and because $X$ is a functor this is

$$\downarrow i \cdot \text{id}^X \cdot \text{nat}'(S[gt_1], \ldots)$$

But $\text{nat}'$ was constructed as an isomorphism between two tensor products (theorem 3.2:16) so this equals

$$\downarrow i \cdot \lambda i \{ b | b \in I \}, \ldots, \{ b | b \in I \}$$

using fact 3.2:4 and the extension of functions for the other tensor product. It is immediate that this equals

$$\emptyset(S(\text{take}_i))$$

The calculations for $\text{smashtake}_i$, $\text{in}_i$ and up follow this pattern.
Next consider the calculation for unfold. The desired result

$$C(\text{unfold}) = \text{nat}_{\text{rec}x. \text{gt}}^{\text{gt}}(S(\text{unfold})) \cdot \text{nat}_{\text{rec}x. \text{gt}}^{-1}$$

is by theorem 3.3:4 equivalent to

$$C(\text{unfold}) \cdot \text{nat}_{\text{rec}x. \text{gt}}$$

$$= C(\text{gt}) \cdot (\text{nat}_{\text{rec}x. \text{gt}}) \cdot \text{nat}_{\text{gt}}(S(\text{rec}x. \text{gt})) \cdot \rho(S(\text{unfold}))$$

which will be abbreviated to

$$c u ' n r = G(n r) \cdot \text{nat}_{\text{gt}}(D) \cdot \rho(s u)$$

Because $\rho(D) = \bigcup_n \rho(r_n) \cdot \rho(r_n)^U$ for embeddings $r_n$ of CPO given as in section 2.2 this follows if

$$\rho(s^n(U)) \quad \rho(r_n) \quad \rho(D) \quad nr \quad s_n \quad cu$$

$$\rho(su) \quad \rho(F(D)) \quad G(L) \quad G(nr) \quad G(\rho(D))$$

in CPOs

can be shown to commute where $F = S(\text{gt})$ and $L = C(\text{rec}x. \text{gt})$. This is immediate for $n = 0$ and for $n > 0$ it follows from the following calculation.

It makes use of the definition of $\text{nat}$ in theorem 3.3:4 and also theorem 2.2:4. For the upper path

$$c u ' n r \cdot \rho(r_n) = G(s_{n-1}) \cdot \text{NAT}_{\text{gt}, n}$$

where $\text{NAT}_{\text{gt}, n}$ is defined in theorem 3.3:4. For the lower path

$$G(nr) \cdot \text{nat}_{\text{gt}}(D) \cdot \rho(su) \cdot \rho(r_n) =$$

$$G(nr) \cdot \text{nat}_{\text{gt}}(D) \cdot \rho(F(r_{n-1})) =$$

by $\text{nat}_{\text{gt}}$ a natural equivalence
\[ G(nr)^*G(\rho(r_{n-1}))' \text{nat}_0 (r^{n-1}(U)) = \]

and by \( G \) a functor

\[ G(s_{n-1})'G(\text{NAT}_{qt, n-1})' \text{nat}_0 (r^{n-1}(U)) \]

But by the definition of \( \text{NAT} \) this is the result along the upper path.

Finally, the result for \text{fold} is immediate from the result for \text{unfold}.

It remains to verify the connection between the functionals. The calculation for \text{cond} has been sketched previously (under the assumption that the natural equivalences are the identities). The calculation for \text{tuple} is illustrative for the other functionals as well and is the only one that will be sketched. Consider functions \( f_i \in \text{S} \) and abbreviate

\[ \text{ct} = C(\text{tuple}) \text{view}_{\text{gt}_1} () (f_1), \ldots, \text{view}_{\text{gt}_k} () (f_k) \]

\[ = \text{lin}(\text{cross}^* \lambda_1. ((\text{nat}_1' \rho(f_i) \text{nat}_0^{-1})(1), \ldots) \]

and

\[ \text{st} = \text{view}_{\text{gt}_1} x_1 \ldots x_k (f_1) (C(\text{tuple}) (f_1), \ldots, f_k) \]

\[ = \text{nat}_1 x_1 \ldots x_k \rho(\lambda_1. (f_1(d), \ldots, f_k(d)))' \text{nat}_0^{-1} \]

Both \( \text{ct} \) and \( \text{st} \) are continuous and additive so it suffices to consider a prime basis element \( p \) of \( C[\text{gt}] \) and show that \( \text{ct}(p) = \text{st}(p) \). Defining \( x_i = \rho(f_i) (\text{nat}_0^{-1}(p)) \) this gives

\[ \text{ct}(p) = \text{cross}(\text{nat}_1(x_1), \ldots, \text{nat}_k(x_k)) \]

\[ \text{st}(p) = \text{nat}_1 x_1 \ldots x_k \rho(\lambda_1. (f_1(d), \ldots, f_k(d)))' \text{nat}_0^{-1} \]

(for \( x \) cartesian product of sets). Using the definition of \( \text{nat}_1 x_1 \ldots x_k \) in theorem 3.3:4 we get

\[ \text{nat}_1 x_1 \ldots x_k = (\text{cross}^* (\text{nat}_1 x) \ldots) x \text{nat}'(S[\text{gt}_1], \ldots) \]

Here \( \text{nat}' \) is given by theorem 3.2:16 so as for \( \text{take}_1 \) above.
\[
\text{st}(p) = (\text{cross'}(\text{nat}_\text{gt} x \ldots)) x'(x_1 \ldots x_k)
\]

where \(x'\) and \(\text{cross'}\) relate to the tensor product \(P(\text{gt}_1, \ldots, x')\).

But \(\text{cross'}(x_1, \ldots, x_k) = x_1 x' \ldots x_k\) so \(\text{st}(p) = \text{ct}(p)\) follows. //

The interplay between \(S\), \(C\) and \(\text{sim}\)

Before undertaking the structural induction that \(\text{sim}_L()\) holds it is convenient to consider the interplay between the functors \(S[t]\) and \(C[t]\) and the predicate \(\text{sim}_E\). A similar situation arises in chapter 4 so to make the development immediately applicable there we consider arbitrary interpretations \(I\) and \(J\). For \(\text{sim}_E\) we assume that there are admissible predicates \(\text{sm'}_E\) on \(I[t] \times J[t]\) fulfilling that

\[
\text{sm'}_E(v, w) \text{ implies that } v = L \text{ if } w = L
\]

As has been said previously the need to assume \((*)\) is because of the presence of the top-level domain constructor \(*\).

Define the category \(\text{SIM}\) as follows. The objects are triples \((D, Q, L)\) where \(D\) and \(L\) are cpo's and \(Q\) is an admissible predicate on \(D \times L\) that satisfies \((*)\). A morphism from \((D, Q, L)\) to \((D', Q', L')\) is a pair \((f, g)\) of cpo's morphisms \(f : D \rightarrow D'\) and \(g : L \rightarrow L'\) such that the following "naturality" conditions hold:

\[
Q(d, 1) \Rightarrow Q'( (f \downarrow 1)(d) , (g \downarrow 1)(1) )
\]

\[
Q( (f \downarrow 2)(d') , (g \downarrow 2)(1') ) \Leftrightarrow Q'(d', 1')
\]

Composition is defined componentwise and the identity on \((D, Q, L)\) is \((\text{id}_D, \text{id}_L)\) where \(\text{id}_D\) and \(\text{id}_L\) are the identities in \(\text{cpo}_s\). It is straightforward to verify that this gives a category.
The notions of chain, cone, limiting cone and mediating morphism were explained in section 2.2 and also applies to SIM#. The category SIM# admits the following analogue of theorem 2.2:2 and lemmas 2.2:5 and 2.2:6 for the characterisation of limiting cones. The result also holds for the variant of SIM where the predicates are only required to be admissible.

Theorem 3.3:12. Consider a chain \( E = ((D, Q, L), (e, f)) \) in SIM such that \( e_n \) and \( f_n \) are embeddings of CPO2s with upper adjoints \( e_n^R \) and \( f_n^R \). A cone \( R = ((D, Q, L), (r_n, s_n)) \) such that \( r_n \) and \( s_n \) are embeddings of CPO2s with upper adjoints \( r_n^R \) and \( s_n^R \) is limiting iff

\[
\begin{align*}
\text{id}_D &= \bigcup_n r_n^* r_n^R \\
\text{id}_L &= \bigcup_n s_n^* s_n^R
\end{align*}
\]

in CPO2s

\[
Q(d, 1) \subseteq \forall n: Q_n( (r_n^2)(d) , (s_n^2)(1) )
\]

A limiting cone of this form always exists and the unique mediating morphism to another cone \( R' = ((D', Q', L'), (r'_n, s'_n)) \) is

\[
(\bigcup_n r_n^1 r_n^R, \bigcup_n s_n^1 s_n^R).
\]

Proof First note that \(((D_n, e_n_n))\) is a chain in CPO2s for which lemma 2.2:5 is applicable. It follows that a cone \((D, (r_n))\) with \( r_n^U = r_n^R \) is limiting iff

\[
\text{id}_D = \bigcup_n r_n^* r_n^R
\]

Further, such a limiting cone always exists and the unique mediating morphism to a cone \((D', (r'_n))\) is

\[
\bigcup_n r_n^1 r_n^R.
\]

Similarly for limiting cones \((L, (s_n))\) of the chain \(((L_n, f_n_n))\).

We begin with proving the existence of a limiting cone \( R \) that satisfies the requirements. Let \((D, (r_n))\) and \((L, (s_n))\) be the limiting cones for \(((D_n, e_n))\) and \(((L_n, f_n))\) as indicated above. Define \( Q \) by the formula displayed in the theorem and note that this defines an admissible predicate that fulfils property (X). To see
that \(((D, Q, L), (r_n, s_n))\) is a cone reduces to showing
\[ Q_n(d_n, l_n) \Rightarrow Q_m((r_n \downarrow 2)((s_n \downarrow 1)(d_n)), \ldots) \]
For \(m \geq n\) we have \(r_m \downarrow 2 \cdot r_n \downarrow 1 = e_{m-1} \downarrow 1 \cdot \ldots \cdot e_n \downarrow 1\) and for \(m < n\) we have
\(r_m \downarrow 2 \cdot r_n \downarrow 1 = e_m \downarrow 2 \cdot \ldots \cdot e_{n-1} \downarrow 2\) as follows from consideration of the cone
\(((D, (r_n \downarrow 1)), (D, (s_n \downarrow 1)))\) for the chain \(((D_n), (e_n \downarrow 1))\) \(\in \mathbb{CPo}_1\). Similarly for \(s_n\) and the result then follows because \(E\) is a chain in \(\mathbb{SIM}\).

We next show that the cone is limiting with the mediating morphism as stated in the theorem. Let \(R'\) be as displayed in the theorem and let \((r, s)\) be the claimed mediating morphism. If \((r', s')\) is another mediating morphism then in particular \(r'\) mediates from \((D, (r_n \downarrow n))\) to \((D', (r_n \downarrow n))\) \(\in \mathbb{CPo}_2\) and this shows that \(r = r'\). Similarly \(s = s'\) so that \((r, s)\) is the only candidate for the mediating morphism. To see it is the mediating morphism it suffices to show that \((r, s)\) is a morphism of \(\mathbb{SIM}\) as the rest follows because \(r\) and \(s\) are mediating in \(\mathbb{CPo}_2\).

To see
\[ Q(d, l) \Rightarrow Q'( (r \downarrow 1)(d) , (s \downarrow 1)(1) ) \]
note that \(Q(d, l)\) implies that
\[ \forall n: Q_n((r_n \downarrow 2)(d) , (s_n \downarrow 2)(1)) \]
so by \(R'\) a cone
\[ \forall n: Q'( (r_n \downarrow 2)^R \downarrow 1(d) , (s_n \downarrow 2)^R \downarrow 1(l) ) \]
and the admissibility of \(Q'\) then gives \(Q'( (r \downarrow 1)(d) , (s \downarrow 1)(1) )\).

To see
\[ Q'(d', l') \Rightarrow Q( (r \downarrow 2)(d') , (s \downarrow 2)(1') ) \]
note that \(Q'(d', l')\) implies that
\[ \forall n: Q_n((r_n \downarrow 2)(d') , (s_n \downarrow 2)(1')) \]
so by $R$ a cone

$$\forall n: Q((r^1_n r^R_n) d^2) (d'), \ldots)$$

and the admissibility of $Q$ then gives $Q((r^2 d^2) (d'), (s^2 (l')) )$.

The statements of the theorem now holds for any limiting cone

$R'' = ((D'', Q'', L''), (r'', s''))$ that satisfies $r_n = r_n^U$ and $s_n = s_n^U$.

This is because any such cone is isomorphic to $R$ that was explicitly constructed above (see section 2.2). For an example calculation let $(r'', s'')$ be the isomorphism from $R$ to $R''$. Then $Q''(d'', l'')$ iff

$$Q((r''^2 d'')) (d''), \ldots)$$

iff $\forall n: Q_n((r''^2 d'')) (d''), \ldots)$ iff

$$\forall n: Q_n((r_n''^2 d'')) (d''), \ldots)$$

The main use of the category $\text{SIM}$ is as a convenient tool for expressing that $\text{IFT}_t$, $\text{IFT}_l$ and sim$_l [s']$ work together in a certain sense. This is formally expressed by proposing the following definition of a functor $\text{IFT}_t$ over $\text{SIM}$. The effect on objects is

$$\text{IFT}_t((D_1, Q_1, L_1), \ldots, (D_N, Q_N, L_N)) = (\text{IFT}_t(D_1, \ldots, D_N), \text{sim}_l [s'](Q_1, \ldots, Q_N), \text{IFT}_l(L_1, \ldots, L_N))$$

and the effect upon morphisms is

$$\text{IFT}_t((f_1, g_1), \ldots, (f_N, g_N)) = (\text{IFT}_t(f_1, \ldots, f_N), \text{IFT}_l(g_1, \ldots, g_N))$$

**Theorem 3.3:13.** $\text{IFT}_t$ is a covariant functor over $\text{SIM}$. ///

**Note** If $t$ does not mention the top-level smash product one could prove a similar theorem using the variant of $\text{SIM}$ where the predicates are only assumed to be admissible.

**Proof** By lemma 3.3:7 $\text{IFT}_t((D_1, Q_1, L_1), \ldots)$ is an object of $\text{SIM}$.

Assuming that $\text{IFT}_t((f_1, g_1), \ldots)$ is a morphism of $\text{SIM}$ it is immediate
to verify the functor laws using that $\mathcal{I}(t)$ and $\mathcal{J}(t)$ are functors and
the componentwise definition of composition and the identities.
Clearly $\mathcal{I}(t)(f_1, \ldots, f_n)$ and $\mathcal{J}(t)(g_1, \ldots, g_n)$ are morphisms of $\mathbf{CPO}_0$s
so to show that $\mathcal{I}(t)((f_1, g_1), \ldots)$ is a morphism of $\mathbb{S}$ it suffices
to prove the "naturality" conditions.

For this let $(f_i, g_i)$ be $\mathbb{S}$ morphisms with domain $(D_i, Q_i, L_i)$ and
range $(D'_i, Q'_i, L'_i)$. Let $(D, Q, L)$ be $\mathcal{I}(t)((D_1, Q_1, L_1), \ldots)$ and let
$(D', Q', L')$ be defined similarly and let $(f, g)$ be $\mathcal{J}(t)((f_1, g_1), \ldots)$.
It is to be shown that

$$Q(d, 1) \Rightarrow Q'((f \downarrow 1)(d), (g \downarrow 1)(1))$$

$$Q'((f \uparrow 2)(d'), (g \uparrow 2)(1')) \Rightarrow Q'(d', 1')$$

This is proved by structural induction on $t$.

**Case** $t = \lambda$. is immediate because $f = \text{id}_D$ and $g = \text{id}_L$ (in $\mathbf{CPO}_0$s) and because
$Q$ equals $Q'$.

**Case** $t = \times_{i=1}^k$. Let $d$ be an element of $\mathcal{I}(t)(D_1, \ldots)$, i.e. of
$\mathcal{I}(t)(D_1, \ldots, D_N) \times \mathcal{I}(t)(D_1, \ldots, D_N)$ and similarly for $l$. If $Q(d, l)$
then for all $i$ sim $\mathcal{I}(t){[\mathcal{J}(t)]}(Q_i', \ldots, Q_N')(d_i, 1_i)$. By the inductive
hypothesis we have for all $i \leq k$ that

$$\text{sim}_{t_i} [\mathcal{J}(t)](Q_i', \ldots, Q_N')(d_i, 1_i), \ldots)$$

so that $Q'(((f \downarrow 1)(d), (g \downarrow 1)(1))$. The other implication is similar.

**Case** $t = \times_{i=1}^k$. As above $Q(d, l)$ gives sim $\mathcal{I}(t){[\mathcal{J}(t)]}(d_i, 1_i)$
for $d_i = (\mathcal{I}(t)(f_1, \ldots) \downarrow 1)(d_i)$ and $1_i$ defined similarly. If no $d_i$ or
$l_i$ is $\bot$ then the result follows as before. Otherwise suppose without
loss of generality that $d_i = \bot$. By lemma 3.3:7 it follows that $l_i = \bot$
so $(f \downarrow 1)(d) = \bot$ and $(g \downarrow 1)(1) = \bot$. Because $Q'(\bot, \bot)$ the result follows.
The other implication is similar.

Cases $t=t_1 + \ldots + t_k$ and $t=t_0 \bot$ are straightforward.

Case $t=t_1 \rightarrow t_2$. Suppose $Q(d, l)$ and note that $d$ is a continuous function from $I[t_1](D_{1}, \ldots)$ to $I[t_2](D_{1}, \ldots)$. Furthermore $(f \downarrow 1)(d)$ equals

$$(I[t_2](f_{1}, \ldots) \downarrow 1) \cdot d' (I[t_1](f_{1}, \ldots) \downarrow 2)$$

(which will be abbreviated to $i_2 \cdot d' i_1$) and is a continuous function from $I[t_1](D_{1}, \ldots)$ to $I[t_2](D_{1}, \ldots)$. Similarly for $(g \downarrow 1)(l)$ and to show $Q'((f \downarrow 1)(d), (g \downarrow 1)(l))$ we must consider $(v, w)$ such that

$$\text{sim} \left[ sm' \right] (Q_{1}, \ldots, Q_{N}) (v, w) \text{ holds and show that}$$

$$\text{sim} \left[ sm' \right] (Q_{1}, \ldots, Q_{N}) ( (f \downarrow 1)(d)(v) , (g \downarrow 1)(l)(w) )$$

holds. By the inductive hypothesis

$$\text{sim} \left[ sm' \right] (Q_{1}, \ldots, Q_{N})(i_1(v), \ldots)$$

so that $Q(d, l)$ gives

$$\text{sim} \left[ sm' \right] (Q_{1}, \ldots, Q_{N})(d(i_1(v)), \ldots)$$

and by the inductive hypothesis the desired result follows. (Note that this proof used that the "naturality" conditions express the effect of "going in both directions" because of the contravariance of the function space construction in $\text{CPO}$.) The other implication is similar.

Case $t=\text{rec} X. t_0$. It is convenient to assume $V \vdash t$ for $\text{card}(V)=N$ and that $X$ has index $N+1$ in $V \in \{X\}$. (The proof can easily be rectified if this is not the case.) Define the following functors over $\text{CPO}$s:

$$F_0 = \text{K}_U, \quad F_{n+1} = \square t_0 (P_1, \ldots, P_N, F_n)$$

$$G_0 = \text{K}_U, \quad G_{n+1} = \square t_0 (P_1, \ldots, P_N, G_n)$$
Let the limiting cone of the CP02s chain

\[ ((F_n(D_1, \ldots, D_n), (F_n(id_{D_1}, \ldots, id_{D_n}))_n) \]

be \((D, (r_n)_n)\) and recall that \(r_n \overset{U}{=} r\). Similarly \((D', (r'_n)_n), (L, (s_n)_n)\)
and \((L', (s'_n)_n)\) are defined as limits for the obviously intended chains.
Define predicates \(R_n\) on \(F_n(D_1, \ldots, D_n) \times G_n(L_1, \ldots, L_n)\) and similarly \(R'_n\)
by the formulae

\[
R_0(v, w) \overset{\text{def}}{=} \sim t, \quad R_{n+1} = \sim t \left[ \sum_{i=1}^{m} (Q_1, \ldots, Q_{N'} R_n) \right]
\]

\[
R'_0(v', w') \overset{\text{def}}{=} \sim t, \quad R'_{n+1} = \sim t \left[ \sum_{i=1}^{m} (Q'_1, \ldots, Q'_{N'} R'_n) \right]
\]

It is an easy numerical induction that

\[
(F_n(f_1, \ldots, f_N), G_n(g_1, \ldots, g_N))
\]
is a \(\text{SIM}\) morphism from

\[
(F_n(D_1, \ldots, D_n), R_n, G_n(L_1, \ldots, L_n))
\]
to

\[
(F_n(D'_1, \ldots, D'_n), R'_n, G_n(L'_1, \ldots, L'_n))
\]

To prove the "naturality" condition that \(Q(d, l)\) implies \(Q'(f_1(d), (g_1(l)))\) suppose \(Q(d, l)\). By definition of \(Q\)

\[
\forall n: R_n((r_n l_2)(d), (s_n l_2)(l))
\]
so because \((F_n(f_1, \ldots, f_N), G_n(g_1, \ldots, g_N))\) is a morphism

\[
\forall n: R'_n((r'_n l_2)(d)(f_1(d)), \ldots)
\]

By lemma 2.2:11 (second half) and the definition of \(f\)

\[
\forall n: R'_n((r'_n l_2)((f l_1)(d)), \ldots)
\]
so that by definition of \(R\) the result follows. The other "naturality"
is similar.

Cases \(t = X\) and \(t = gt \rightarrow gt'\) are straightforward. //

156
This theorem may be compared with lemma 13.65 in /Sto77/ and the results in /Rey74/ about how to obtain a so-called relational functor. The local continuity of $I_{f(t)}$ and $J_{f(t)}$ ensure that the similar condition holds for $I_{f(t)}$. This is sufficient for obtaining a continuity result analogous to lemma 2.2:3, namely that $I_{f(t)}$ preserves limiting cones for chains like those mentioned in theorem 3.3:12. (The details may be extracted from the proof of the following theorem.)

We are now ready to perform the structural induction that shows $\text{sim}_t()$ and thereby establishes the relationship between $S$ and $C$.

**Theorem 3.3:14.**

1. If $\varnothing \vdash t : e$ then $\text{sim}_t()$ and $\text{C}(e)$. 
2. In general consider interpretations $I$ and $J$ and admissible predicates $\text{sm}_{f(t)}$ on $I_{f(t)}$ and $J_{f(t)}$. Suppose $\varnothing \vdash t : e$ and that an analogue of lemma 3.3:11 holds for $I$, $J$, and $\text{sim}_t[\text{sm}]$. Then $\text{sim}_t()$ holds if

   (a) $t$ contains no top-level smash product
   or if each $\text{sm}_{f(t)}$ satisfies that
   (b) $\text{sm}_{f(t)}(v,w)$ implies that $v = _L$ iff $w = _L$ (X) ///

**Proof** Clearly (1) is a special case of (2b). The proof of (2a) is similar to the proof of (2b) but uses the variant of $\text{SIM}$ where predicates are only assumed to be admissible. To prove (2b) the following more general result is proved by structural induction on $e$:

if $\text{tenv} \vdash e : t$ and for all $x$: $\text{sim}_{\text{tenv}(x)}[\text{sm}](\text{tenv}(x), \text{jenv}(x))$
then $\text{sim}_t[\text{sm}](\text{tenv}(x), \text{jenv}(x), \text{sm}(\text{tenv}(x), \text{jenv}(x)))$

157
Most cases are straightforward using that an analogue of 3.3:11 holds. The cases considered below are among the harder ones.

**Case e = (e₁, ..., eₙ, X).** The inductive hypothesis gives

\[ \text{sim}_{t_i}^{[\text{sm}']}() (If_{e_i}J(\text{ienv}), J_{e_i}J(jenv)) \]

so that

\[ \text{sim}_{t}^{[\text{sm}']}() (If_{e}J(\text{ienv}), J_{e}J(jenv)) \]

is immediate in the case where no \( I_{e_i}J(\text{ienv}) \) or \( J_{e_i}J(jenv) \) is \( \bot \).

If one is suppose it is \( I_{e_i}J(\text{ienv}) \). Lemma 3.3:7 shows that \( J_{e_i}J(jenv) \) also is so the result follows because \( \text{sim}_{t}^{[\text{sm}']}() (L, L) \).

**Case e = Y e'.** Here \( I_{e}J(\text{ienv}) = \bigcup_n (I_{e'}J(\text{ienv}))^n(L) \) and similarly for \( J \). Clearly \( \text{sim}_{t}^{[\text{sm}']}() (L, L) \) holds so by the inductive hypothesis

\[ \text{sim}_{t}^{[\text{sm}']}() ( (I_{e'}J(\text{ienv}))^n(L), \ldots ) \]

and admissibility then gives \( \text{sim}_{t}^{[\text{sm}']}() (I_{e}J(\text{ienv}), \ldots ) \).

**Cases e = mkrec e' and e = unrec e".** Let \( t_0 \) be given such that \( \text{tenv} e' : t_0 [\text{recX.t}_0] \) and \( \text{tenv} e'' : \text{recX.t}_0 \) and note that \( \{X\} \vdash t_0 \).

Write \( D = \{\text{recX.t}_0\} \) and \( L = \{\text{recX.t}_0\} \) and let \( i : \{\text{recX.t}_0\}(D) \to D \) and \( j : \{\text{recX.t}_0\}(L) \to L \) be the isomorphisms of CPOs such that (by lemma 2.3:2)

\[ \{\text{mkrec e'}\}(\text{ienv}) = i(\{\text{e'}\}(\text{ienv})) \]

\[ \{\text{unrec e}"\}(\text{ienv}) = i^{-1}(\{\text{e}"\}(\text{ienv})) \]

and similarly for \( j \). The result is straightforward from the inductive hypothesis and lemma 3.3:8 once it has been shown that

\[ \text{sim}_{\text{recX.t}_0}^{[\text{sm}']}() (i(v), j(w)) \]

iff \( \text{sim}_{t_0}^{[\text{sm}']} (\text{sim}_{\text{recX.t}_0}^{[\text{sm}']}()) (v, w) \)
To show this we consider \( IJt_0 \) over \( \text{SIM} \). Define the predicate true by \( \text{true}(v, w) = tt \). Using theorem 3.3:13 it is easy to see that

\[
((IJt_0)_n(U, \text{true}, U), (IJt_0)_n(L, L))
\]
is a chain in \( \text{SIM} \). The limiting cone is

\[
((D, Q, L), (r_n, s_n))
\]
for \( Q = \text{sim} \) and \( r_n, s_n \) as in theorem 3.3:12. It follows from section 2.2 that \( (D, (r_n)_n) \) is the limiting cone in \( \text{CP02s} \) of the chain \((IJt_0)_n(U), (IJt_0)_n(L)) \) and similarly for \( (L, (s_n)_n) \). Next define \( r'_0 = \bot \) and \( r'_{n+1} = IJt_0(r_n) \) and similarly for \( s'_n \). From theorem 3.3:13 it follows that

\[
R = ((IJt_0)_D, \text{sim} t_0 (sm'), (IJt_0)_L), (r'_n, s'_n)_n)
\]
is also a cone. Therefore there is a unique mediating morphism \( (r, s) \) from the limiting cone to \( R \). Theorem 3.3:12 and lemma 2.2:5 show that \( r \) is the unique mediating morphism in \( \text{CP02s} \) from \( (D, (r_n)_n) \) to \((IJt_0)_D, (r'_n)_n) \). Since \( IJt_0 \) is locally continuous \( r \) is an isomorphism and in fact \( r = (i^{-1}, i) \). Similarly \( s = (j^{-1}, j) \) and the desired result follows because \( (r, s) \) is a morphism of \( \text{SIM} \). (This argument is easily extended to show that \( R \) is also a limiting cone and thereby proves the continuity result for \( IJ(t) \) that was mentioned above.)

This theorem may be compared to /Nie82/ where a similar theorem (but with program points) is proved for a simple imperative language. The theorem there is based on a store semantics /MiSt76/ rather than a standard semantics and it is said (p.274 lines 1 to 4) that the theorem would not hold for the obvious standard semantics. The corresponding remark here is that it is essential to use the tensor
product for $C(x)$ rather than the cartesian product. For use of the cartesian product might suggest:

$$C(\text{tuple}) (g_1, g_2) = A_1 (g_1(1), g_2(1))$$

$$C(\text{add}) = A(Y,Z), \{y+z \mid y \in Y, z \in Z\} \_R$$

(assuming that the natural equivalences are the identities). Then $C\[\text{add} \ast \text{tuple}(id, id)\] \{2,3\} \_R$ gives $\{4,5,6\} \_R$ instead of the correct $\{4,6\} \_R$. This corresponds to what is done in /Don78/. The problem is that the connection between the pairs of values is lost. A similar problem is likely to occur if $t::=gt$ is allowed (e.g. $C[\lambda x. \text{add}(x,x)]$).

Remark We conclude this section with a remark about the definition of $\text{sim}_{\text{rec}X.t}(d,1)$. Its definition was of the form

$$V_n: \text{SIM}_{t,n} (r_n^U(d), s_n^U(1))$$

for embeddings $r_n$ and $s_n$ of $C\text{POS}$. The definition of $\text{REC}$ in section 2.2 was also given using embeddings but it was shown that a least fixed point definition could be used instead. Similar remarks apply to $\text{view}_{\text{rec}X.t}$. When predicates are ordered by "is implied by" the analogue of a least fixed point definition is

$$V_n: Q_n(d,1)$$

where $Q_0(d,1)$ is always true and

$$Q_{n+1}(d,1) \equiv \text{sim}_t(Q_n)(r^{-1}(d), s^{-1}(1))$$

for isomorphisms $r$ and $s$ in $C\text{POS}$.

Theorem 3.3: 13 and induction can be used to relate $\text{SIM}_{t,n}$ and $Q_n$ by showing that

$$((r_n^U, s_n^U), (r_n^U, s_n^U)) : (G[t]^n(U), \text{SIM}_{t,n}, C[t]^n(U))$$

$$\rightarrow (G[\text{rec}X.t], Q_n, C[\text{rec}X.t])$$
is a morphism of $SIM$. It follows that (1) is equivalent to

$$\forall n: Q_n(\sigma_n^{U}(d), \nu_n^{U}(l))$$

and that (2) implies (1). The converse implication may fail as will be sketched below. Intuitively this should not be surprising because the definition of $\text{sim}_{t_1 \rightarrow t_2}$ is such that $\text{sim}_{t}(Q)$ does not necessarily depend monotonically on $Q$.

For a sketch of an example where (1) does not imply (2) consider

$$t = (U_1+N_\perp \cdot X) + N_\perp + (X \rightarrow X)$$

Intuitively $S[recX.t]$ can code (among other things) lists of integers, integers and functions from lists of integers to integers. Let $d$ be the coding of the function that maps the coding of the list $(i,n_1,\ldots,n_k)$ to the coding of $n_i$ if $i$ is between 1 and $k$ and gives the coding of 1 otherwise. Since $C[recX.t] = S[recX.t]$ we may take $l = d$ and (1) is straightforward. On the other hand (2) fails because already $Q_2(d,l)$ fails. The codings of the lists $(1,2)$ and $(1,3)$ are related by $Q_1$ but this is not the case for the codings of the results (2 and 3). The problem is that $d$ and $l$ are not restricted in the number of unfoldings they are allowed to perform.

The failure of the attempt (2) at defining $\text{sim}_{recX.t}()$ as a least fixed point may be circumvented by placing suitable restrictions on $recX.t$. However, even in the general case $\text{sim}_{recX,t}()$ is essentially the only solution to the following equation

$$Q \equiv (\forall d,l: Q(d,l) \iff \text{sim}_{t}(Q)(r^{-1}(d),s^{-1}(l)))$$

To be precise: if an admissible predicate $Q$ is a solution and $Q(d,l)$ implies that $d=\perp$ iff $l=\perp$ then $Q$ equals $\text{sim}_{recX.t}()$. It follows from the proof of theorem 3.3:14 that $\text{sim}_{recX.t}()$ is a solution. Next if
Q is an solution it follows that

\[((r, r^{-1}), (s, s^{-1})): \]
\[
(S[t[rec\times t/x]], sim_t(Q), C[t[rec\times t/x]]) \rightarrow \\
(S[rec\times t], Q, C[rec\times t])
\]
is a morphism of SIM. One can then construct a cone

\[
R = ((S[rec\times t], Q, C[rec\times t]), \ldots )
\]
for the chain

\[
((S[t^n)(u))_n, (S[t^n)(u))_n)
\]
Since the mediating morphism from the limiting cone to R is \((id, id)\)
the result follows. \(/ / / \)
In this chapter abstract interpretation is developed for the meta-language. This builds on a study of the relation "is a safe approximation to" between (forward) data flow analyses specified as interpretations. As in chapter I this is done using pairs of abstraction and concretization functions. Conditions upon such pairs are studied in section 4.1.

The overall framework is developed in section 4.2. In this development certain problems arise because of the presence of the top-level smash product which therefore needs to be abandoned in some parts of the development. Using a family \((\text{con}_{\text{gt}})_{\text{gt}}\) of concretization functions a relation is defined between two interpretations (say \(I\) and \(J\)) that specify data flow analyses. The relation is sufficient and (essentially) necessary for

\[ I \subseteq D \iff \text{con}_{\text{gt}} \subseteq \text{con}_{\text{gt}}, \quad J \subseteq D \]

to hold whenever \(\emptyset \vdash e : gt \rightarrow gt'\). Given \(I\) and additionally a family \((\text{abs}_{\text{gt}})_{\text{gt}}\) of abstraction functions one can define an interpretation \(I'\) that gives as faithful a picture of \(I\) as is possible using the (generally more approximate) spaces of \(I'\). The interpretation \(I'\) is said to be induced from \(I\) and is a convenient way of specifying data flow analyses (not least if \(I\) is the collecting interpretation \(C\)).

In the above it was not required that \(\text{con}_{\text{gt}}\) was defined structurally on \(gt\). In keeping with the principle of compositionality of
denotational semantics we study in section 4.3 how a compositional definition can be given. In part this resembles the construction of the natural equivalence in the proof of theorem 3.3:4. In section 4.4 the induced functionals are studied. Taking composition \( \circ \) as an example one might expect it to be functional composition. Similarly for other functionals expected formulations may be given. Because "inducing" is very precise the induced version of an expected definition need not, however, itself be an expected definition. It is shown that under certain conditions it is safe to replace the induced functionals with the expected definitions. Finally, section 4.5 discusses the role of the tensor product in generalising the relational flow analysis method.

4.1 PAIRS OF ADJOINED FUNCTIONS

In the introduction we considered approximation between complete lattices. The partial orders were viewed as representing the notion of "can safely be approximated by". To describe approximation a concretization function \( \text{con} \) and an abstraction function \( \text{abs} \) were used. It was argued that they must satisfy that

\[
\text{abs}(l) \preceq m \iff 1 \in \text{con}(m)
\]

This is equivalent to assuming \( \text{abs} \) and \( \text{con} \) to be monotonic and adjoined: \( \text{con}'\text{abs} \preceq \text{id} \) as is illustrated by

![Diagram of concretization and abstraction]

and \( \text{abs}'\text{con} \preceq \text{id} \). If additionally \( \text{abs}'\text{con} = \text{id} \) we shall say that \( \text{abs} \) and \( \text{con} \) are exactly adjoined /Nie81a/. This has often been assumed.
(e.g. /CoCo77b/) because then con is one-one and M contains no superfluous elements. Part of the development in this chapter can be performed using weaker assumptions than adjoined (see /Nie82/) but this will not be considered here.

These concepts can be defined relative to any cpo-category. A pair \((f:L\rightarrow M, g:M\rightarrow L)\) of morphisms is said to be adjoined iff \(f'g\leq \text{id}\) and \(g'f\geq \text{id}\). A morphism \(f\) is a lower adjoint iff there exists a morphism \(g\) such that \((f,g)\) is an adjoined pair. The \(g\) need not exist but if it does it is uniquely determined: for if both \(g\) and \(g'\) would do then

\[ g = g'\text{id} \geq g'(f'g') = (g'f)'g' \geq \text{id}'g' = g' \]

and similarly \(g' \geq g\). A morphism \(g\) is an upper adjoint if there exists a (necessarily unique) morphism \(f\) such that \((f,g)\) is an adjoined pair. An adjoined pair \((f,g)\) with \(f'g=\text{id}\) is said to be exactly adjoined.

It will be seen that in the previous paragraph a category \(\text{Cf}\) of complete lattices and monotonic functions was presupposed.

In most of this chapter we shall assume pairs \((\text{abs}, \text{con})\) of abstraction and concretization functions to be adjoined relative to \(\text{ACLc}\). The use of this category will be discussed below. It is straightforward to show for such a pair of adjoined functions that the lower adjoint \(\text{abs}\) is completely additive and the upper adjoint \(\text{con}\) is completely multiplicative (the dual notion of preserving greatest lower bounds). Further the formulae /CoCo79/

\[
\text{abs}(l) = \prod \{m \mid l \subseteq \text{con}(m)\}
\]

\[
\text{con}(m) = \bigcup \{l \mid \text{abs}(l) \subseteq m\}
\]

are immediate.
We now argue that it is not overly restrictive to assume that all pairs \((\text{abs,con})\) of abstraction and concretization functions are adjoined relative to ACLs. The restrictions must be compared with those that are commonly made in abstract interpretation and monotone frameworks.

Abstract interpretation (as explained in the Introduction and e.g. /CoCo77b/) may be viewed as using \(\text{CL}_m\) instead of ACLs. The use of ACLs means that (precisely) two additional restrictions are imposed. One is that the objects (complete lattices) must be algebraic. The other is that the (monotonic) concretization functions must be strict and continuous.

The formulation of monotone frameworks /KaUl77/ considers objects that are semi-lattices of finite height that have a least element. Such objects are complete lattices and they are algebraic iff they are countable (as will be the case in applications). Functions upon such objects are assumed to be monotonic but because of the finite height this is equivalent to assuming that they are continuous.

The assumption that concretization functions be continuous is motivated by the desire to stay within the usual "continuous" domain theory. It is a restriction with respect to abstract interpretation but hardly in practice, i.e. not with respect to monotone frameworks. The assumption of strictness is motivated by the presence of domain constructors (like \(\otimes\)) that are only functors over a category of strict functions. This should be no serious restriction as an adjoined situation

\[(\text{abs}: L \rightarrow M, \text{con}: M \rightarrow L)\]
may be replaced by.

(strict(up'abs):L→M, lift(con):M→L)

and here lift(con) is strict.

Algebraicity of objects is motivated by the construction of the tensor product in section 3.2. To see this is not overly restrictive consider first objects corresponding to bottom-level types in the collecting interpretation C. These are algebraic complete lattices because they are \( \text{isomorphic to} \) the powerdomain of the corresponding objects in the standard interpretation. Consider next the corresponding object in an interpretation that specifies a data flow analysis. If it is related to the object in C by a pair of exactly adjoined functions then it will be algebraic too. This is a consequence of:

**Lemma 4.1.** Let \( L \) be an algebraic complete lattice and \( M \) a partially ordered set. Let \( \text{abs}:L→M \) and \( \text{con}:M→L \) be monotonic functions such that

\[
\text{con'}\text{abs}≤\text{id} \quad \text{and} \quad \text{abs'}\text{con}≤\text{id}
\]

Then \( M \) is a complete lattice with least upper bound of a subset \( X \) given by \( \text{abs}(\bigcup\{\text{con}(m)\mid m∈X\}) \). If additionally \( \text{con} \) is continuous then \( M \) is algebraic with \( B_M = \{\text{abs}(l)\mid l∈B_L\} \).

**Proof** For the first result we calculate

\[
\forall m∈X: m∈m',
\forall m∈X: \text{con}(m)≤\text{con}(m'),
\bigcup\{\text{con}(m)\mid m∈X\} ≤ \text{con}(m'),
\text{abs}(\bigcup\{\text{con}(m)\mid m∈X\}) ≤ m'.
\]

and this shows that \( M \) is a complete lattice with least upper bounds.
as stated. For the second result let \( b \in B_L \) and show \( \text{abs}(b) \in B_M \). To see this let \( \text{abs}(b) \in \bigcup_{m_1} m_1 \) so that \( b \in \text{con}(\bigcup_{m_1} m_1) = \bigcup_{m_1} \text{con}(m_1) \). Then there is \( i \) such that \( b \in \text{con}(m_1) \) hence \( \text{abs}(b) \in m_1 \). To see that \( M \) is algebraic with \( B_M \) as stated we use fact 2.2:1 so it suffices to note for \( m \in M \) that

\[
m = \text{abs}(\text{con}(m)) = \text{abs}\left(\bigcup_{b \in B_L} b \in \text{con}(m)\right)
= \bigcup \{\text{abs}(b) \mid b \in B_L \land b \in \text{con}(m)\}
= \bigcup \{\text{abs}(b) \mid b \in B_L \land \text{abs}(b) \in m\}
\]

and that \( \{\text{abs}(b) \mid b \in B_L \land \text{abs}(b) \in m\} \) is directed. ///

The lemma may fail if \( \text{abs} \cdot \text{con} = \text{id} \) was weakened to \( \text{abs} \cdot \text{con} \neq \text{id} \). As an example let \( M \) be a cpo that is not algebraic and not a complete lattice. Taking \( L=U \), \( \text{abs}=1 \), \( \text{con}=\bot \) the assumptions of the lemma are fulfilled. If \( \text{con} \) is continuous, however, we have that \( \{\text{abs}(b) \mid b \in B_L\} \) is a subset of \( B_M \).

Taking it for granted that objects should be algebraic one can give the following intuitive argument for why they should be complete lattices. First they should be cpo's if iteration and recursion (\( Y \) in the metalanguage) is still to be treated by a least fixed point approach as is also the case in the MFP solution of data flow analysis. Secondly objects should be semi-lattices so that the effect of conditional (\( \text{cond} \) in the metalanguage) may be expressed using a least upper bound as in \( L(\text{cond}) \). Lemma 3.2:2 then shows that objects should be complete lattices.
Considering further restrictions

We now concentrate on a pair \((\text{abs:} L \rightarrow M, \text{con:} M \rightarrow L)\) of abstraction and concretization functions that is adjoined relative to ACLs. The abstraction function is strict, continuous and additive (hence completely additive) and the concretization function is strict and continuous but not additive. It follows from the discussion after the previous lemma that abs sends a finite element to a finite element. This is equivalent to saying that it specializes to a continuous function \(\text{abs:} B_L \rightarrow B_M\). We now investigate the cases where it specializes to \(\text{abs:} PB_L \rightarrow PB_M\) or \(\text{abs:} IB_L \rightarrow IB_M\). (Recall that PB is the set of finite elements that are primes and similarly IB for irreducible.)

Preservation of primes is particularly of interest for powerdomains.

Lemma 4.1:2. Let \((\text{abs:} L \rightarrow M, \text{con:} M \rightarrow L)\) be a pair of adjoined functions where \(L\) is isomorphic to a powerdomain. Then abs specializes to \(\text{abs:} PB_L \rightarrow PB_M\) iff con is additive.

Proof For "if" suppose that con is additive and that \(b\) is a finite prime of \(L\). Clearly \(\text{abs}(b)\) is finite and to see that it is a prime suppose that \(\text{abs}(b) \subseteq m_1 \cup m_2\). Then \(b \in \text{con}(m_1) \cup \text{con}(m_2)\) so for some \(i\) we have \(b \in \text{con}(m_i)\) and hence \(\text{abs}(b) \subseteq m_i\). For "only if" suppose that abs specializes to \(\text{abs:} PB_L \rightarrow PB_M\). Then the calculation

\[
\text{con}(m_1 \cup m_2) = \]

(algebraicity of \(L\) and theorem 3.1:5)

\[
\bigcup \{b \in PB_L \mid b \in \text{con}(m_1 \cup m_2)\} = \]

\[
\bigcup \{b \in PB_L \mid \text{abs}(b) \subseteq m_1 \cup m_2\} = \]

\[
\text{con}(m_1) \cup \text{con}(m_2) \]

169
shows that \(\text{con} \) is additive.

"If" holds even if \(L\) is not isomorphic to a powerdomain but "only if" may fail if this is not the case.

Suppose now that \(L = \mathcal{O}(D)\) and \(M = \mathcal{O}(E)\). If \(\text{abs}\) specializes to \(\text{abs}: PB_L \rightarrow PB_M\) it is possible to define a representation function \(r: D \rightarrow E\) by

\[
\text{abs}(\{d\}_R) = \{r(d)\}_R
\]

This defines a continuous function \(r\) that is strongly strict, i.e., \(r(d) = \perp\) iff \(d = \perp\) and it specializes to \(r: B_D \rightarrow B_E\). It is immediate to verify the formulae

\[
\text{abs}(I) = LC(\{r(b) \mid b \in I\})
\]
\[
\text{con}(I') = \{b \in B_D \mid r(b) \in I'\}
\]

Conversely for any strongly strict and continuous function \(r: B_D \rightarrow B_E\) the above formulae define a pair of adjoined functions with \(\text{con}\) additive. An example representation function is \(r: Z_L \rightarrow \{-, 0, +\}_L\) defined by \(r(-1) = -\) etc. (as was used in the Introduction). It will, however, be too restrictive to assume all concretization functions to be additive as this will make it impossible to handle many data flow analyses that can otherwise be handled. Examples include constant propagation and the example in the Introduction where an independent attribute method is related to a relational method.

The situation where \(\text{abs}\) specializes to a map \(\text{abs}: IB_L \rightarrow IB_M\) will be found of use in section 4.4. We begin with motivating the intuitive role played by the irreducible finite elements. An element \(l\) of \(L\) is said to be \textbf{essential} iff whenever \(l = \bigcup X\) for a subset \(X\) of \(L\) then \(l\) is an element of \(X\) or \(l = \perp\). Intuitively \(l\) can only be
specified by itself. There is an immediate analogy with the definition of finite element in section 2.2; there $X$ was assumed to be a countable and directed subset and $1 = \bigcup X$ was weakened to $1 \in \bigcup X$.

**Lemma 4.1:** $\mathbb{E} L$ is the set of finite elements that are essential.

**Proof** An essential element is clearly irreducible so a finite and essential element is in $\mathbb{E} L$. Conversely let $i$ be an element of $\mathbb{E} L$ and $X$ a subset of $L$ such that $i = \bigcup X$. The algebraicity of $L$ gives $\bigcup X = \bigcup LC(X)$ for $LC$ as defined in section 3.1. If $LC(X)$ is empty we have $i = \mathbb{L}$ which is essential so assume otherwise. Then there is a total and onto map $k : N \rightarrow LC(X)$ and the axiom of choice guarantees a total map $k' : N \rightarrow X$ such that $k(n) \in k'(n)$. It follows that $\bigcup X = \bigcup_{n=1}^{m} k'(n)$ such that $k(n) \in k'(n)$. Now $(\bigcup_{n=1}^{m} k'(n))_m$ is a chain and $i \in \bigcup X$ gives $i \subseteq \bigcup_{n=1}^{m} k'(n)$ for some $m$ (because $i$ is finite). Clearly $i = \bigcup X$ gives $i = k'(1) \cup \ldots \cup k'(m)$ so because $i$ is irreducible there is $m'$ such that $i = k'(m')$. It follows that $i$ is an element of $X$.  

That abs specializes to $\text{abs} : \mathbb{E} L \rightarrow \mathbb{E} M$ therefore says that abstracting an essential piece of information must give an essential piece of information. It is a restriction to assume abstraction functions to satisfy this but we shall argue that it may be acceptable to do so. For this we consider conditions upon the upper adjoint $\text{con} : M \rightarrow L$.

It is said to be "irreducibly covered" iff

$$\forall M \in M: \forall i \in \mathbb{E} L: i \in \text{con}(m) \Rightarrow \exists j \in \mathbb{E} M: j \in \text{con}(j)$$

and "irreducibly generated" iff $\text{con} = \text{lin}(\text{con})$, i.e. iff

$$\forall M \in M: \text{con}(m) = \bigcup \{ \text{con}(j) \mid j \in \mathbb{E} M \land j \in \text{con}(j) \}$$

**Lemma 4.1.4.** For a pair (abs : $L \rightarrow M, \text{con} : M \rightarrow L$) of adjoined functions the statements
Abs specializes to \( \text{abs} : \mathcal{I} \mathcal{B}_L \rightarrow \mathcal{I} \mathcal{B}_M \)

Con is irreducibly covered

are equivalent and if \( L \) is isomorphic to a powerdomain they are equivalent to

Con is irreducibly generated

Proof If (1) holds then (2) is shown by "choosing" \( j = \text{abs}(i) \) in the definition of irreducibly covered. If (2) holds then (1) is shown by instantiating the definition of irreducibly covered with \( i \in \mathcal{I} \mathcal{B}_L \) and \( m = \text{abs}(i) \). Assume now that \( L \) is isomorphic to a powerdomain. If (3) holds then to show (1) consider \( i \in \mathcal{I} \mathcal{B}_L \) and calculate

\[
\text{Con} (\text{abs}(i)) = \bigcup \{ \text{con}(j) \mid j \in \mathcal{I} \mathcal{B}_M \land j \in \text{abs}(i) \}
\]

But, by theorem 3.1:7, \( i \in \mathcal{P} \mathcal{B}_L \) and therefore \( \text{Con}(j) \) for some \( j \in \mathcal{I} \mathcal{B}_M \) with \( j \in \text{abs}(i) \). This shows \( \text{abs}(i) = j \) and hence \( \text{abs}(i) \in \mathcal{I} \mathcal{B}_M \). Finally, if (1) holds then (3) is proved as follows. First

\[
\text{Con}(m) \supseteq \bigcup \{ \text{con}(j) \mid j \in \mathcal{I} \mathcal{B}_M \land j \notin m \}
\]

is immediate. For the other inclusion note that (by theorems 3.1:5 and 3.1:7)

\[
\text{Con}(m) = \bigcup \{ i \in \mathcal{I} \mathcal{B}_L \mid i \in \text{Con}(m) \}
\]

so that one may calculate

\[
\text{Con}(m) \subseteq \bigcup \{ \text{Con}(\text{abs}(i)) \mid i \in \mathcal{I} \mathcal{B}_L \land \text{abs}(i) \in m \}
\]

\[
\subseteq \bigcup \{ \text{Con}(j) \mid j \in \mathcal{I} \mathcal{B}_M \land j \notin m \}
\]

To illustrate these conditions choose \( L = \emptyset (N, L) \) and

\[
M_1 = \begin{array}{ccc}
& 1 & \\
& & 2 \\
\text{L} & & \\
& & \\
& & 1 & 2 \\
\end{array}
\]

\[
M_2 = \begin{array}{ccc}
& 1 & \\
& & 2 \\
\text{L} & & \\
& & \\
& & 1 & 2 \\
\end{array}
\]

large
Define pairs of adjoined functions \((\text{abs}_1, \text{con}_1)\) by \(\text{con}_2(\text{large}) = \{3, 4, \ldots\} \bot\) and \(\text{con}_1(2) = \{2\}\) etc. Then \(\text{con}_2\) but not \(\text{con}_1\) is irreducibly generated. Examples like \(\text{con}_1\) do come up occasionally /Jon81a/ so it is a restriction to require concretization functions to be irreducibly generated. It is the author's point of view that this restriction is acceptable. This is partly because the example in /Jon81a/ has always been "somewhat unnatural" in the view of the author and partly because the restriction is motivated by the consideration of essential elements. At any rate the restriction will be of use in a small part of the development in section 4.4. When \(L\) is not a powerdomain the consideration of essential elements suggests that "irreducibly covered" rather than "irreducibly generated" is the condition to be imposed upon concretization functions. This condition is preserved by composition of upper adjoints because (1) in the theorem is preserved when composing the corresponding lower adjoints. (The second condition is not preserved under composition: consider \(f: M_1 \rightarrow M_2\) and \(g: M_2 \rightarrow \{\bot\}\) defined by \(f(x) = x\) and \(g(x) = (x\text{large}) \rightarrow \top, \bot\).)

It is appropriate at this point to give a comparison with /WaSh77/ where the semantics of data types is studied. (In giving the comparison some technical differences will be ignored.) Starting with an algebraic cpo \(D\) they give axioms for a "type structure" \(M\) over \(D\). The axioms prescribe that the elements of \(M\) must be elements of \(\mathcal{P}(D)\) and \(M\) is partially ordered in the same way that \(\mathcal{P}(D)\) is, i.e. by subset inclusion. Further, \(M\) is a complete lattice and it is possible that \(\mathbb{P}_M = \{\bot\}\). What is interesting is that \(\mathbb{I}_M\) is isomorphic to \(\mathbb{B}_D\). This means that their notion of the "tightening" of \(h\) is nothing but \(\text{lin}(h)\). Further they use "tightening" to prove certain results by "case analysis" where a "case" is an element of
This idea is intuitively close to the arguments to be given in section 4.4 for the use of \( \text{lin} \) in the expected definitions for \( \text{filter}_x \) and \( \text{take}_i \) (for tensor products).

### 4.2 Approximating Interpretations

In this section we define a relation between those interpretations that specify data flow analyses. The relation guarantees that the semantics given to certain expressions by one interpretation is safely approximated by the other. For specifying data flow analyses the immediate possibility is to specify an interpretation in full. We explore another possibility namely to induce an interpretation, i.e., to define an interpretation that mimicks a given interpretation but using more approximate spaces.

For this development it is helpful to have additional assumptions about the interpretations that specify data flow analyses. Define an **approximating interpretation** to be an interpretation \( I \) such that:

- the categories \( I(B) \) and \( I(B_0) \) are ACL and ACLas respectively
- the bottom-level functors \( I(x), I(w), I(+), I(\downarrow) \) are locally continuous semi-functors on ACLs

The collecting interpretation is an approximating interpretation as follows from theorem 3.2:14 (and the facts listed before it). The use of ACL for \( I(B) \) is in accord with the discussion in section 4.1 and the use of ACLas is because the tensor product is not a functor in general. The requirement that the bottom-level functors should be locally continuous semi-functors over ACLs is motivated by:

**Fact 4.2:1.** If \( F : \text{ACLs}^N \rightarrow \text{ACLs} \) is a locally continuous semi-functor and
(abs, con) are pairs of adjoined functions (relative to ACLs) then also (P(abs, ..., abs), P(con, ..., con)) is a pair of adjoined functions.

For an approximating interpretation each \( I^[[gt]] \) is a locally continuous semi-functor over ACLs. This is by a structural induction where the case \( gt=\lambda i \) is immediate. The remaining cases, except \( gt=\text{rec}X.gt' \), are straightforward because composition and tupling preserve the property of being a locally continuous semi-functor (as was already remarked in section 3.2). For the case \( gt=\text{rec}X.gt' \) it should be stressed that \( \text{REC}(G) \) is defined on morphisms by the formula

\[
\text{REC}(G)(g_2, \ldots, g_N) = \bigcup_n r_n \cdot (A_{g,G}(g_2, \ldots, g_N))^{n(\lambda)} r_n U
\]

for embeddings \( r_n \) and \( r'_n \) as in section 2.2.

**Lemma 4.2.2.** If \( G: ACLs \rightarrow ACLs \) is a locally continuous semi-functor then also \( \text{REC}_1(G): ACLs \rightarrow ACLs \) is.

**Proof.** We already know that \( \text{REC}(G) \) is a locally continuous functor over ACLs. Clearly it is locally continuous in general so it suffices to prove that

\[
\text{REC}_1(G)(g_2, g'_2, \ldots, g_N, g'_N) \in \text{REC}_1(G)(g_2, \ldots, g_N) \cdot \text{REC}_1(G)(g'_2, \ldots, g'_N)
\]

holds in general and that equality holds when all \( g_i \) are additive.

This is by a straightforward calculation. ///

**Remark.** In lemma 2.2:10 a fixed point characterisation of \( \text{REC}(G)(\ldots) \) was given. The proof given is not applicable here because it was assumed that \( G \) was a functor over the category where the \( g_i \) are morphisms. If \( I^[[gt]] \) denotes the "functor" where \( \text{REC}(\ldots) \) is defined according to the fixed point characterisation then this too
defines a locally continuous semi-functor over ACLs. It agrees with $\mathbb{I}^g t_{U}$ on abstraction functions and concretization functions. The result for abstraction functions is because they are lower adjoints and hence additive. Denote by $\text{abs}^U$ the upper adjoint corresponding to $\text{abs}$. The result for concretization functions then is because they are upper adjoints and because the formula

$$\mathbb{I}^g t_{U}(\text{abs}^U) = (\mathbb{I}^g t:\text{abs})^U = (\mathbb{I}^g t:\text{abs})^U = \mathbb{I}^g t_{U}(\text{abs}^U)$$

is a consequence of the fact above.

### RELATING THE INTERPRETATIONS

A central idea in abstract interpretation is to formalise the notion of when one data flow analysis is a safe approximation of another. Consider two approximating interpretations $\mathbb{I}$ and $\mathbb{J}$ and let $g \in \mathbb{I}^g t_{\rightarrow g t'}$ and $h \in \mathbb{J}^g t_{\rightarrow g t'}$ be functions to be related. For this purpose we assume that there is a family $(\text{abs}, \text{con}) = (\text{abs}^g t, \text{con}^g t_{g t})$ of pairs of adjoined abstraction and concretization functions. As in the introduction the relation to be used is

$$\mathbb{sa}^g t_{\rightarrow g t'} (g, h) \equiv \text{g'con}^g t \mathbb{E} \text{con}^g t_{g t} h$$

This can be formulated in a number of equivalent ways, one being that $g(v) \mathbb{E} \text{con}^g t_{g t} (h(\text{abs}^g t(v)))$ holds for all elements $v$ of $\mathbb{I}^g t$.

To extend this relation to all closed types $t$ define the relation $\leq_{\text{con}, t}$ on $\mathbb{I}^g t_0 x \mathbb{I}^g t_0$ by

$$g \leq_{\text{con}, t} h \text{ iff } \text{sim}_t(\mathbb{sa}) (g, h)$$

using the $\text{sim}_t$ predicate defined in section 3.3. It follows from lemma 3.3:7 that this defines an admissible predicate. It further satisfies certain "reflexive" and "transitive" laws. To state these
Let \( \text{id} \) denote the family \( (\text{id})_{\text{gt}} \) of concretization functions and let \( (\text{con}'\text{con}')_{\text{gt}} \) be defined componentwise as \( \text{con}'_{\text{gt}} \text{con}'_{\text{gt}} \).

**Lemma 4.2:** If \( t \) is a closed and contravariantly pure type then

- \( g \preceq_{\text{id}, t} g \)
- \( g \preceq_{\text{con}, t} h \) and \( h \preceq_{\text{con}', t} k \) implies \( g \preceq_{\text{con}', t} k \)
- \( g \preceq_{\text{id}, t} h \) implies \( g \preceq_{t} h \)

**Proof** For the first and third result call an admissible predicate \( Q \) benevolent iff \( Q(g, g) \) holds and \( Q(g, h) \Rightarrow g \preceq h \) holds for all \( g \) and \( h \). The result then follows from

if all \( Q_i \) are benevolent then so is \( \text{sim}_t[\text{sa}](Q_1, \ldots, Q_N) \)

that is proved by structural induction upon types \( t \) such that \( \forall v \cdot t \) and \( \text{CP}(v, t) \) hold for some \( v \) with \( \text{card}(v) = N \). The case \( t = \text{rec}X.t_0 \) is by a straightforward numerical induction and in the case \( t = t_1 \rightarrow t_2 \) lemma 3.3: 9 is used to show that \( \text{sim}_t[\text{sa}](Q_1, \ldots, Q_N)(v, w) \) holds iff \( v = w \).

For the second result say that predicates \( Q \) and \( Q' \) force \( Q'' \) iff for all \( g, h \) and \( k \) that \( Q(g, h) \) and \( Q'(h, k) \) implies \( Q''(g, k) \). (We do not merely define \( Q'' \) as the relational composition of \( Q \) and \( Q' \) as this does not in general produce an admissible predicate.) The result then follows from

if for all \( i \): \( Q_i \) and \( Q'_i \) force \( Q''_i \)

then \( \text{sim}_t[\text{con}](Q_1, \ldots, Q_N) \) and \( \text{sim}_t[\text{con}](Q'_1, \ldots, Q'_N) \)

force \( \text{sim}_t[\text{con}', \text{con}'](Q''_1, \ldots, Q''_N) \)

that is proved by structural induction upon types \( t \) such that \( \forall v \cdot t \) and \( \text{CP}(v, t) \) for some \( v \) with \( \text{card}(v) = N \). The proof is quite straightforward.
Note The assumption about the type being contravariantly pure cannot simply be removed. To see this let \( f: \mathbb{N} \to \mathbb{N} \) be some finite element and define \( h: \mathbb{N} \to \mathbb{T} \) by \( h(f') = \text{tt} \) if \( f' \) is strictly greater than \( f \) and \( h(f') = \bot \) otherwise. Then \( h \nleq \text{id}_{\mathbb{N} \to \mathbb{T}} \) fails because \( f \nleq \text{id}_{\mathbb{N} \to \mathbb{T}} \) but \( \bot \nleq \text{id}_{\mathbb{T}} \) is false. ///

We now define a relation \( \leq_{\text{con}} \) between approximating interpretations that is analogous to the condition expressed by lemma 3.3:11. Define \( I \leq_{\text{con}} J \) iff for all entries \( z \) of type \( t \) mentioned in the expression part it is the case that \( I(z) \leq_{\text{con}, t} J(z) \). The relation is intended to express that \( J \) is a safe approximation to \( I \). It satisfies the following "reflexive" and "transitive" laws:

**Lemma 4.2.4.** For approximating interpretations \( I, J \) and \( K \):

- \( I \leq_{\text{id}} I \)
- \( I \leq_{\text{con}} J \) and \( J \leq_{\text{con}} K \) implies \( I \leq_{\text{con}, t} K \)

**Proof** Because of the previous lemma it suffices to show the result for entries \( z \) corresponding to functionals. The first result is immediate because \( I(z) \) is monotonic (in fact continuous). For the second result let abs be the lower adjoint corresponding to \( \text{con} \) and suppose

\[
g_i \cdot \text{con} \cdot \text{con}' \cdot \text{con}' \cdot k_i
\]

Define \( h_i = \text{abs} \cdot g_i \cdot \text{con} \) so that \( g_i \cdot \text{con} \cdot h_i \) and \( h_i \cdot \text{con} \cdot \text{con}' \cdot k_i \).

Then

\[
I(z)(g_1, \ldots, g_n) \cdot \text{con} \cdot \text{con}' \cdot \text{con} \cdot J(z)(h_1, \ldots, h_n) \cdot \text{con}'
\]

\[
\cdot \text{con} \cdot \text{con}' \cdot k(z)(k_1, \ldots, k_n)
\]

The "transitivity" result means that data flow analyses can be built
in stages by "stepwise coarsening".

The relation really desired between approximating interpretations \( I \) and \( J \) is that \( \forall e \in \text{con}, \exists c \in \text{con} \) for all expressions \( e \). This is almost achieved:

**Theorem 4.2: 5.**

\[
\begin{align*}
\forall e \in \text{con}, t. & \quad J[e] \leq I[e] \\
\forall e \in \text{con}, t. & \quad I[e] \leq J[e] \\
\forall e \in \text{con}, t. & \quad J[e] \leq I[e] \leq J[e] \\
\end{align*}
\]

**Proof** The first implication is because the metalanguage contains the expressions take\(_1\), ..., \( \lambda x:f_1 \times \ldots \times f_k \cdot \text{tuple}(x_1, \ldots, x_k) \), ..., \( f_1 \). The second implication is by theorem 3.3: 14.

**Note** After the proof of lemma 4.2: 3 it was said that \( h \not\leq \text{id}, t \) may fail if \( t \) is not contravariantly pure. This is not contradicted by the theorem above even though \( I \leq \text{id} \) always holds by lemma 4.2: 4. This is because the \( h \) that causes problems cannot be obtained as \( h=\exists[e] \) for some expression \( e \).

**Remark** In the statement of the theorem it was assumed that the type \( t \) of the expression \( e \) does not use the top-level smash product. To show that this is essential we shall assume that the constants of contravariantly pure type include the truth value \( \top \), the undefined function \( \bot:N \to N \) and the identity function \( \text{id}:N\to N \). Then

\[
\begin{align*}
\forall e & \leq \text{id}, N \to N, \text{id} \text{id} \\
\forall e & \leq \text{id}, T \leq \text{id} \\
\end{align*}
\]
Both hold but

\[ C[\text{take}_2(x,ttx)] \leq_{id,T} C[\text{take}_2(xid,ttx)] \]

fails. To overcome this problem we could redefine

\[ g \leq_{\text{con},gt\rightarrow gt'} h \iff g \cdot \text{con}_{gt} \leq_{gt'} h \quad \text{and} \quad g = \top \Leftrightarrow h = \top \]

but then lemma 4.2:4 fails: it is not the case that \( C \leq_{id} C \) because \( C(\text{cond})(g_1, g_2, g_3) \) may be \( \top \) without any of the \( g_i \) being \( \bot \) and clearly \( C(\text{cond})(\top, \top, \top) \) is not \( \bot \). The solution chosen was to abandon the use of smash product and this is in line with the remark in section 2.1.

Another solution might be to replace \( t::=gt\rightarrow gt' \) by \( t::=(gt\rightarrow gt')\bot \) as then the smash product will pose no problems. This may be motivated with the following discussion about least elements. Usually in denotational semantics the least element in a cpo (e.g. \( N_\bot \)) is "unreal" in the sense that when produced by a computation it really means that the computation did not terminate. On the other hand the least element in a function space is a "real" value namely the function that never terminates when executed. If this function is produced by a computation this is not the same as saying that the computation did not terminate. Therefore an "unreal" least element should be added below the least "real" element.

///

**INDUCING AN INTERPRETATION**

We now turn to ways of describing a data flow analysis. The obvious possibility is to specify all the components of an approximating interpretation. It is possible, however, to specify the expression part in a very "automatic" way by generalising the idea in /CoCo79/ of inducing predicate transformers. The type part of
the abstracting interpretation must still be specified in full.

So let I be an approximating interpretation and J the type part of an approximating interpretation. Further assume a family

\[(abs, con) = (abs_{gt : I[gt] \to J[gt]}, con_{gt : J[gt] \to I[gt]})_{gt}\]

of adjoined abstraction and concretization functions (adjoined relative to ACLs). Consider now a function \(g\) in \(I[gt \to gt']\) and how to transport it to \(J[gt \to gt']\). An obvious possibility, and the one used in /CoCo79/, is to use

\[\text{ind}_{gt \to gt'}(g) = \text{abs}_{gt'} \cdot g \cdot \text{con}_{gt}\]

as is illustrated by the diagram

\[
\begin{array}{ccc}
I[gt] & \xrightarrow{\text{con}_{gt}} & J[gt] \\
\downarrow g & & \downarrow \\
I[gt'] & \xrightarrow{\text{abs}_{gt'}} & J[gt']
\end{array}
\]

Clearly \(\text{ind}_{gt \to gt'}(g)\) is a safe approximation to \(g\), i.e.

\[g \leq_{\text{con},gt \to gt'} \text{ind}_{gt \to gt'}(g)\]

Also it is the best approximation to \(g\) in that

\[g \leq_{\text{con},gt \to gt'} h \iff \text{ind}_{gt \to gt'}(g) \leq h\]

which may be generalised to

\[g \leq_{\text{con}',gt \to gt'} h \iff \text{ind}_{gt}(g) \leq_{\text{con}',gt \to gt'} h\]

Another way of phrasing these conditions is that \(\text{ind}_{gt \to gt'}(g)\) is as faithful a representation of \(g\) that the spaces in \(J\) allow.

The above process can be extended to all contravariantly pure and closed types \(t\) using the transformation view \([\ldots]\) defined in
section 3.3. So define

\[ \text{induce}_{(\text{abs, con}), t}(g) = \text{view}_t[\text{ind}]() \]

As before induce\(_{(\text{abs, con}), t}(g)\) is the best safe approximation of \(g\).

**Lemma 4.2:6.** For any contravariantly pure and closed type \(t\) and family \((\text{abs, con})\) of pairs of adjoined functions

\[
- \ g \triangleq \text{con}', t \ \text{induce}_{(\text{abs, con}), t}(g) \\
\]

\[
- \ g \triangleq \text{con}', t \ \text{h iff induce}_{(\text{abs, con}), t}(g) \triangleq \text{con}', t \ h \quad ///
\]

**Proof** The first result is immediate by lemma 3.3:10 because the strictness of \(\text{con}_t\) ensures that \(\text{abs}_t, \text{g'}, \text{con}_t\) is strongly strict.

For the second result "if" is immediate by the first result and lemma 4.2:3. "Only if" follows from the following result:

if \(k_i\) are strongly strict and \(Q_i'(g, h) \Rightarrow Q_i'(k_i(g), h)\)
then, writing \(g' = \text{view}_t[\text{induce}_{(\text{abs, con})}](k_1, \ldots, k_N)(g)\),
it follows that

\[ \text{sim}_t[\triangleq \text{con}', \text{con}'](Q_1, \ldots, Q_N)(g, h) \]
implies

\[ \text{sim}_t[\triangleq \text{con}', \text{con}'](Q_1', \ldots, Q_N')(g', h) \]

The result is proved by induction on types \(t\) such that \(\forall t\) and \(\text{CP}(V, t)\) for some set \(V\) with \(\text{card}(V) = N\). We omit the details (but see the proof of lemma 3.3:10 for hints). ///

Because induce\(_{(\text{abs, con})}\) was defined using view an analogue of lemma 3.3:5 holds. From this it easily follows that

\[
\text{induce}_{(\text{id, id}), t} = \text{id}_t \quad \text{I[1]} \\
\text{induce}_{(\text{abs, con})'(\text{abs}', \text{con}'), t} = \text{induce}_{(\text{abs, con}), t} \quad \text{induce}_{(\text{abs}', \text{con}'), t}
\]

182
where the composition of families of pairs of functions is defined by
\[(\text{abs}_{\text{gt}}, \text{con}_{\text{gt}}) \cdot (\text{abs'}_{\text{gt}}, \text{con'}_{\text{gt}}) \cdot \text{gt} = (\text{abs'}_{\text{gt}}, \text{con'}_{\text{gt}}, \text{con}_{\text{gt}}) \cdot \text{gt}\]

It remains to consider how to induce functionals as this is not found in /CoCo79/. So let \(z\) be a functional of type
\[t = (\text{gt}_{\text{gt}} \rightarrow \text{gt}'), \ldots, (\text{gt}_{\text{gt}} \rightarrow \text{gt}'), \rightarrow (\text{gt} \rightarrow \text{gt}')\]
(which is closed but not contravariantly pure). We shall extend the definition of induce to include such types by defining
\[\text{induce}(\text{abs}, \text{con}), t(\mathbb{I}(z)) = \lambda(h_1, \ldots, h_n). \text{abs}_{\text{gt}} \cdot \mathbb{I}(z)(\text{con'}_{\text{gt}} \cdot h_1 \cdot \text{abs}_{\text{gt}} \cdot \ldots) \cdot \text{con}_{\text{gt}}\]

This is best explained by considering the following diagrams. We begin with
\[
\begin{array}{c}
\mathbb{I}(\text{gt}) \\
\text{h}_1 \rightarrow \mathbb{I}(\text{gt}^{'})
\end{array}
\]
which may be thought of as approximating
\[\mathbb{I}(\text{gt}) \text{I}_1 = \text{con}_{\text{gt}} \cdot h_1 \cdot \text{abs}_{\text{gt}} \rightarrow \mathbb{I}(\text{gt}^{'})\]
The desired composition of these is
\[
\begin{array}{c}
\mathbb{I}(\text{gt}) \mathbb{I}(z)(g_1, \ldots, g_n) \\
\rightarrow \mathbb{I}(\text{gt}^{'})
\end{array}
\]
which has
\[
\begin{array}{c}
\mathbb{I}(\text{gt}) \text{abs}_{\text{gt}} \cdot \mathbb{I}(z)(g_1, \ldots, g_n) \cdot \text{con}_{\text{gt}} \rightarrow \mathbb{I}(\text{gt}^{'})
\end{array}
\]
as its best approximation. In other words, the extension of induce amounts to defining
\[\text{induce}(\text{abs}, \text{con}), t \rightarrow t'(h) = \text{induce}(\text{abs}, \text{con}), t \cdot h' \text{induce}(\text{con}, \text{abs}), t\]
and this idea allows to extend the definition of induce to all closed types.
Putting it all together we define the approximating interpretation 
\[ \text{induce}(I, (\text{abs}, \text{con}), J) \] 
where \( I \) is an interpretation, \( J \) is the type part 
of an interpretation and \((\text{abs}, \text{con})\) is a \( \text{gt} \)-indexed family of adjoined abstraction and concretization functions. The type part is given by 
\( J \) and for an entry \( z \) (of type \( t \)) in the expression part:
\[ \text{induce}(I, (\text{abs}, \text{con}), J)(z) = \text{induce}_{\text{abs}, \text{con}}(I(z)) \]
That this gives the best approximation to \( I \) follows from:

Theorem 4.2:7.

\[ I \nleq_{\text{con}} \text{induce}(I, (\text{abs}, \text{con}), J) \]
\[ I \nleq_{\text{con}} J \iff \text{induce}(I, (\text{abs}, \text{con}), J) \cong_{\text{id}} J \]

Proof. Write \( J' = \text{induce}(I, (\text{abs}, \text{con}), J) \). For the first result consider an entry \( z \) of type \( t \). If \( t \) is of contravariantly pure type the result follows from lemma 4.2:6. Otherwise \( t \) is of the form 
\[
(\text{gt}_1 \rightarrow \text{gt}'_1) \times \ldots \times (\text{gt}_n \rightarrow \text{gt}'_n) \rightarrow (\text{gt} \rightarrow \text{gt}')
\] 
Suppose \( g_i \nleq_{\text{con}, \text{gt} \rightarrow \text{gt}'} h_i \) so that \( g_i \nleq_{\text{con}, \text{gt} \rightarrow \text{gt}'} h_i \cdot \text{abs} \cdot \text{gt} \). Since \( I(z) \) is continuous we may calculate

\[
I(z)(g_1, \ldots, g_n) \cdot \text{con} \cdot \text{gt} \\
\cdot \text{con} \cdot \text{abs} \cdot \text{gt} \\
= \text{induce}(I(z))(h_1, \ldots, h_n)
\]
and this shows that \( I(z)(g_1, \ldots, g_n) \nleq_{\text{con}, \text{gt} \rightarrow \text{gt}'} J'(z)(h_1, \ldots, h_n) \).

The "if" part of the second result is by lemma 4.2:4 and the first result. For "only if" it follows by lemma 4.2:6 that it suffices to consider an entry \( z \) of type \( t \) as above. If \( g_i \nleq_{\text{id}, \text{gt} \rightarrow \text{gt}'} h_i \) then the result
\[ J'(z)(g_1, \ldots, g_n) \nleq_{\text{id}, \text{gt} \rightarrow \text{gt}'} J(z)(h_1, \ldots, h_n) \]

184
follows from the calculation

\[ J'(z)(g_1, \ldots, g_n) = \]
\[ \text{abs}_{g_t} \cdot I(z)(\text{con}_{g_t} \cdot g_1 \cdot \text{abs}_{g_t} \cdot \ldots) \cdot \text{con}_{g_t} \cdot E \]

(since \( I \leq \text{con} \cdot J \))

\[ \text{abs}_{g_t} \cdot \text{con}_{g_t} \cdot J(z)(h_1, \ldots, h_n) \cdot E \]

\[ J(z)(h_1, \ldots, h_n) \]

It is straightforward to show the equalities

\[ \text{induce}(I, (\text{id}, \text{id}), I) = I \]
\[ \text{induce}(\text{induce}(I, (\text{abs}, \text{con}), J), (\text{abs'}, \text{con'}), K) = \]
\[ \text{induce}(I, (\text{abs'}, \text{con'}), (\text{abs}, \text{con}), K) \]

The last equality says that data flow analyses may be developed in a stepwise process and the resulting data flow analysis is as if it had been developed using only one step.

4.3 COMPOSITIONAL DEFINITION OF ABS AND CON

The definition of an interpretation specified that the meaning of bottom-level types was defined in a compositional way. It does not appear to be useful to have a notion of "long interpretation" where this is not so. Similarly it does not appear to be useful to consider \( \text{abs}_{g_t} \) and \( \text{con}_{g_t} \) that are not compositionally defined.

The developments in the previous section were independent of how \( \text{abs}_{g_t} \) and \( \text{con}_{g_t} \) are specified so the question of how to specify them compositionally has been deferred to this section.

To be more specific let \( I \) and \( J \) be type parts of approximating interpretations. We shall assume as given pairs

\[ (\text{abs}_{A_1} : I(A_1) \rightarrow J(A_1), \text{con}_{A_1} : J(A_1) \rightarrow I(A_1)) \]
of adjoined (relative to \(\text{ACLs}\)) abstraction and concretization functions. The goal is to obtain a family

\[
(\text{abs}_{gt}: I\{gt\} \rightarrow J\{gt\}, \text{con}_{gt}: J\{gt\} \rightarrow I\{gt\})_{gt}
\]

of adjoined functions (where \(gt\) ranges over the closed types only).

It may help to begin with sketching the simple case where \(I\) and \(J\) specify the same locally continuous semi-functor for the same bottom-level domain constructor. In the terminology of data flow analysis this implies that if \(I\) uses the relational method for \(x\), i.e. \(I(x) = \emptyset\), then so does \(J\). Consider a closed type \(gt\) and let

\(A_{11}, \ldots, A_{1n}\) be all the \(A_i\)'s occurring in \(gt\). Define the term \(gt' = gt[x_j/A_{1j}]\) to be \(gt\) with \(A_{1j}\) replaced by \(X_j\). Then \(I\{gt'\}\) (as a semi-functor over \(\text{ACLs}\)) agrees with \(I\{gt\}\) and we shall denote it by \(F_{gt}\). Further \(I\{gt\} = F_{gt}(I(A_{11}), \ldots, I(A_{1n}))\) and similarly for \(J\{gt\}\). Defining

\[
\begin{align*}
\text{abs}_{gt} &= F_{gt}(\text{abs}_{A_{11}}, \ldots, \text{abs}_{A_{1n}}) \\
\text{con}_{gt} &= F_{gt}(\text{con}_{A_{11}}, \ldots, \text{con}_{A_{1n}})
\end{align*}
\]

then specifies functions of the right functionality and each

\((\text{abs}_{gt}, \text{con}_{gt})\) is an adjoined pair because \(F_{gt}\) is a locally continuous semi-functor. This corresponds roughly to what is done in /Nie81a/ for a somewhat simpler structure of types.

**The Framework for Change of Method**

In the general case the definition of \(\text{abs}_{gt}\) and \(\text{con}_{gt}\) is not so straightforward. As an example suppose that \(I\) uses the relational method for \(x\), i.e. \(I(x) = \emptyset\), whereas \(J\) uses the independent attribute method, i.e. \(J(x) = x\). A change in "method" takes place and some
additional information is needed to specify "how". The usual way of passing from one functor to another is by a natural transformation. This is complicated by \( I(x) \) not being a functor upon all of \( \text{ACLs} \) and we shall therefore consider subcategories of \( \text{ACLs} \). It is possible to give criteria for which subcategories that may be used but for the present purposes it suffices to consider the following two categories.

Define \( \text{ACLsl} \) to be as \( \text{ACLs} \) except that the morphisms included are only the lower adjoints of \( \text{ACLs} \). Similarly \( \text{ACLsu} \) has upper adjoints as morphisms. Both are subcategories of \( \text{ACLs} \) (but not sub cpo-categories). By fact 4.2:1 any locally continuous semi-functor specializes to a functor over both \( \text{ACLsl} \) and \( \text{ACLsu} \).

Let \( \# \) be any one of the bottom-level domain constructors \( x, y, +, \bot \). To specify "how" the method is transformed from \( I(\#) \) to \( J(\#) \) we assume natural transformations

\[
\begin{align*}
\text{ABS}_{\#} & \quad \text{from } I(\#) \text{ to } J(\#) \text{ over } \text{ACLsl} \\
\text{CON}_{\#} & \quad \text{from } J(\#) \text{ to } I(\#) \text{ over } \text{ACLsu}
\end{align*}
\]

satisfying that each

\[
(\text{ABS}_{\#}(L_1, \ldots, L_k), \text{CON}_{\#}(L_1, \ldots, L_k))
\]

is a pair of adjoined functions. The definition of \( \text{abs}_{\#} \) and \( \text{con}_{\#} \) is very similar so a function \( \text{lengthen}_{\#} \) is defined below. It will be used to define

\[
\begin{align*}
\text{abs}_{\#} & = \text{lengthen}_{\#}(I,(\text{abs}_{\#} L_1, \ldots, L_i, \text{ABS}_{\#}), J)() \\
\text{con}_{\#} & = \text{lengthen}_{\#}(J,(\text{con}_{\#} L_1, \ldots, L_i, \text{CON}_{\#}), I)()
\end{align*}
\]

and it will be shown that \( (\text{abs}_{\#}, \text{con}_{\#}) \) is a pair of adjoined functions of the right functionality.
The function \( \text{lengthen}_{\text{gt}} \) is defined by structural induction on types \( \text{gt} \) such that \( V_1 \cdot \text{gt} \) for some \( V \) with cardinality \( N \). Let \( I \) and \( J \) be type parts of approximating interpretations and let

\[
\begin{align*}
\rightarrow \lambda A_1 : I(A_1) 
\end{align*}
\]

be morphisms of \( \text{ACLs} \) and similarly

\[
\begin{align*}
\rightarrow \lambda h_1 \cdot L_1 
\end{align*}
\]

be morphisms of \( \text{ACLs} \) and similarly

\[
\begin{align*}
\rightarrow \lambda L_1 \cdot L_1 
\end{align*}
\]

for all choices of objects \( L_1 \). It is convenient to abbreviate

\[
\begin{align*}
\rightarrow \lambda h_1 \cdot h_1 \cdot h_N 
\end{align*}
\]

The definition then is

\[
\begin{align*}
\rightarrow \lambda \ln_{\text{gt}} h_1 \cdot \cdots \cdot h_N 
\end{align*}
\]

\[
\begin{align*}
\rightarrow \lambda \ln_{\text{gt}} h_1 \cdot \cdots \cdot h_N 
\end{align*}
\]

for any one of \( X \), \( \ast \), \( + \), \( \bot \)

\[
\begin{align*}
\rightarrow \lambda \ln_{\text{recX.gt}^0} h_1 \cdot \cdots \cdot h_N 
\end{align*}
\]

where \( s_n \) and \( r_n \) are the evident embeddings of \( \text{ACLs} \) and

\[
\begin{align*}
\rightarrow \lambda \ln_{\text{gt}^0} h_1 \cdot \cdots \cdot h_N 
\end{align*}
\]

assuming that \( X \) has index \( N+1 \) in \( V_{\mathcal{X}} \).

Clearly this definition makes sense in \( \text{ACLs} \) and defines a morphism of functionality

\[
\begin{align*}
\rightarrow \lambda \ln_{\text{gt}} h_1 \cdot \cdots \cdot h_N 
\end{align*}
\]

That \( \text{abs}_{\text{gt}} \) is actually a lower adjoint and \( \text{con}_{\text{gt}} \) an upper adjoint is a consequence of the following:

**Lemma 4.3.1.** Let \( \text{ACLsz} \) be one of \( \text{ACLs1} \) and \( \text{ACLs} \). Let the \( \rightarrow \lambda A_1 
\)
morphisms of $\text{ACL}_{sz}$ and $nt$ is a natural transformation over $\text{ACL}_{sz}$. Then

$$\text{lengthen}(I,(f_{\Lambda_1}^*(\Lambda_1), (nt)^*J()))$$
is a morphism of $\text{ACL}_{sz}$.  

\textit{Proof}  
The result follows from (using the abbreviation $\text{ln}_{gt}$)

if $h_1, \ldots, h_N$ are morphisms of $\text{ACL}_{sz}$ then so is $\text{ln}_{gt}(h_1, \ldots, h_N)$

that is proved by structural induction on $gt$ such that $V \vdash gt$ for
some $V$ with $\text{card}(V) = N$. The cases $gt = \Lambda_i$ and $gt = X$ are straightforward.
The case $gt = gt\ldots \# gt_k$ for $\#$ one of $X$, $\times$, $+$ or $\Lambda$ is because $I(\#)$
is a functor over $\text{ACL}_{sz}$, $nt_\#$ is a natural transformation over $\text{ACL}_{sz}$
and because the composition of $\text{ACL}_{sz}$ morphisms gives such a morphism.
Indeed if $\text{ACL}_{sz}$ is $\text{ACL}_{sl}$ and $g^U$ denotes the upper adjoint of $g$ then

$$\text{ln}_{gt}(h_1, \ldots, h_N)^U = I(\#)(\text{ln}_{gt_1}(h_1, \ldots, h_N)^U, \ldots)$$

and similarly if $\text{ACL}_{sz}$ is $\text{ACL}_{su}$. For the case $gt = \text{rec}_X, gt_0$ it is a straightforward induction to prove that $LN_{gt_0, n}$ is in $\text{ACL}_{sz}$ (because $1: U \to U$ is). To see that $U \ni n \text{ln}_{gt_0, n}^U$ is in $\text{ACL}_{sz}$ consider the case where $\text{ACL}_{sz}$ is $\text{ACL}_{sl}$ as the other case is similar. Let $LN_{gt_0, n}^U$ be the upper adjoint of $LN_{gt_0, n}$ and define $x = \bigcup_n r_n^{LN_{gt_0, n}} U$.
The calculations $\text{ln}_{gt}(h_1, \ldots, h_N)^* x \text{id}$ and $x^* \text{ln}_{gt}(h_1, \ldots, h_N)^* \text{id}$ show

that $\text{ln}_{gt}(h_1, \ldots, h_N)$ is a lower adjoint of $\text{ACL}_{sz}$ with upper adjoint $x$.

\textit{Corollary 4.3:2.}  
Given the conditions of the lemma, and assuming that
$h_i : L_i \to L'_i$ are morphisms of $\text{ACL}_{sz}$ we have that

$$\text{ln}_{gt_1, \ldots, gt_k}(h_1, \ldots, h_N) = I(\#)(\text{ln}_{gt_1}(h_1, \ldots, h_N), \ldots)$$

\textit{Proof}  
Use that $nt_\#$ is a natural transformation.
Given this corollary it is straightforward to adapt the proof of the lemma to show that \((\text{abs}_{gt}, \text{con}_{gt})\) is a pair of adjoined functions.

The lengthening process satisfies certain "reflexive" and "transitive" properties. To state these let \(I, J\) and \(K\) be type parts of approximating interpretations and let \(\text{id}\) be the natural transformation from \(I(\#)\) to \(I(\#)\) given by \(\text{id}_{\#}(L_1, \ldots) = \text{id}_{\#}(L_1, \ldots)\).

For a term \(gt\) such that \(\forall \theta \cdot \text{gt and card}(V) = N\) let \(i_1, \ldots, i_n\) be the increasing list of indices \(i\) such that \(A_i\) occurs in \(gt\). Further let \(\text{gt}(x_{N+j}/A_i)\) denote the term where each \(A_i\) is replaced by \(x_{N+j}\) (and if necessary renaming domain variables bound by a \(\text{rec}\)).

**Lemma 4.3.3:** Let \(\text{ACL}_{sz}\) be one of \(\text{ACL}_{sl}\) or \(\text{ACL}_{su}\). Let \(nt = (nt_1, \ldots, nt_n)\) and \(nt'\) be lists of natural transformations over \(\text{ACL}_{sz}\) and let \(f = (f_1, \ldots, f_n)\), \(f'\) and \(f''\) be sequences of morphisms of \(\text{ACL}_{sz}\) and let \(h_1, h'_1, h''_1\) be morphisms of \(\text{ACL}_{sz}\). Then ("reflexivity")

\[
\text{lengthen}_{gt}(I, f, \text{id}, I)(h_1, \ldots, h_n) = I[I[\text{gt}(x_{N+j}/A_i)] I](h_1, \ldots, h_n, f_1, \ldots, f_n)
\]

and ("transitivity")

\[
\text{lengthen}_{gt}(I, f'', nt', K)(h_1, \ldots, h'', h''_1) = \text{lengthen}_{gt}(I, f', nt', J)(h_1, \ldots, h''_1)
\]

The proof is by structural induction on \(gt\) and uses that \(nt\) is a natural transformation from \(I(\#)\) to \(J(\#)\) and that \(I(\#)\) is a functor.

The definition of \(\text{lengthen}_{\text{recX}, \text{gt}}_0\) was given in a form using embeddings. Using the previous lemma the following fixed point definition can be given much in the same way as was done for \(\text{REC}\) in section 2.2. This result will be used in the next section.
Lemma 4.3.4. Under the same conditions as in lemma 4.3.1 and its corollary we have
\[
\ln_{\text{recX} \cdot \gt_0} (h_1, \ldots, h_n) = \text{LFP}(\forall h_{N+1}, i' \ln_{\gt_0} (h_1, \ldots, h_{N+1}) \cdot j^{-1})
\]
Here it is assumed that \( \chi \) has index \( N+1 \) in \( V_\nu \{X\} \), where \( V_\nu \text{recX} \cdot \gt_0 \), and that \( i \) and \( j \) are the evident isomorphisms from \( \gt [\text{recX} \cdot \gt/X] \) to \( \text{recX} \cdot \gt \).

Proof. The definition of the lefthand side is \( \bigcup_n s_n \cdot \text{Ln} \cdot \gt_0, n \cdot r_n^U \) and the righthand side is of the form \( \bigcup_n G^n(\Lambda) \). To show the result it follows from \( \bigcup_n r_n^U = \text{id} \) and \( \bigcup_n s_n \cdot s_n^U = \text{id} \) that it suffices to show either
\[
s_n \cdot \text{Ln} \cdot \gt_0, n = G^n(\Lambda) \cdot r_n \quad \text{or} \quad \text{Ln} \cdot \gt_0, n \cdot r_n^U = s_n \cdot G^n(\Lambda)
\]
If the category in question (ACL\textsubscript{ex}) is ACL\textsubscript{ex1} then the first result will be shown and when the category is ACL\textsubscript{su} the second result will be shown.

The proofs are similar so suppose the category is ACL\textsubscript{ex1}. The proof is by induction on \( n \) and the base case is trivial so consider \( n+1 \). We may calculate
\[
G^{n+1}(\Lambda) \cdot r_{n+1} =
\]
(by section 2.2 \( r_{n+1} = j' \text{Ig}_0 \{ (\text{id}, \ldots, r_n) \} \))
\[
i' \ln_{\gt_0} (h_1, \ldots, h_N, G^n(\Lambda)) \cdot \text{Ig}_0 \{ (\text{id}, \ldots, r_n) \} =
\]
(by the previous lemma because \( G^n(\Lambda), h_i \) and \( r_n \) are lower adjoints)
\[
i' \ln_{\gt_0} (h_1, \ldots, h_N, G^n(\Lambda)) \cdot \text{lengthen}_{\gt_0} (\ldots) (\text{id}, \ldots, r_n) =
\]
(by the previous lemma)
\[
i' \ln_{\gt_0} (h_1, \ldots, h_N, G^n(\Lambda) \cdot r_n) =
\]
(by the inductive hypothesis and calculation as above)
\[
s_{n+1} \cdot \text{Ln}_{\gt_0, n+1}
\]

///

191
EXAMPLE CHANGES OF METHOD

The compositional definition of \( \text{abs}_{gt} \) and \( \text{con}_{gt} \) depends on the natural transformations \( \text{ABS} \) and \( \text{CON} \). When \( I(\#) \) and \( J(\#) \) are the same locally continuous semi-functor it is natural to use the identity natural transformation \( \text{id}_\# \). When they differ we shall consider three examples below. They do all have the flavour of changing from a "relational method" to an "independent attribute method". The latter methods may be desirable from a practical point of view because they are likely to result in faster although less precise data flow analyses (see /Jon81a/ for a formal result).

Further it is mostly such methods that have previously been considered (in e.g. /Don78/). There is no clear definition of when a method is relational but we shall take it to mean "is as in the collecting semantics".

Example: The domain constructor \( X \).

Here we investigate changing from the relational method \( I(X) = \emptyset \) to the independent attribute method \( J(X) = X \). The proposed transformations are (recalling section 3.2):

\[
\text{CON}_X(L_1, \ldots, L_k) = \text{cross}_X
\]

\[
= \lambda(l_1, \ldots, l_k). \{ t \mid \exists b_1 \in \text{B}_{L_1}, \ldots, (\forall b_1 : b_1 \in L_1). t \not\in \emptyset(b_1, \ldots, b_k) \}
\]

\[
\text{ABS}_X(L_1, \ldots, L_k) = \text{id}_X
\]

\[
= \lambda J. \cup \{ (b_1, \ldots, b_k) \mid \emptyset(b_1, \ldots, b_k) = J \}
\]

Clearly this defines morphisms in \( \text{ACLs} \).

To see that the pairs of morphisms constitute pairs of (exactly) adjoined functions it suffices to perform the following two
calculations. First

\[ \text{ABS}_{X}(L_1, \ldots, L_k) \times \text{CON}_{X}(L_1, \ldots, L_k) = \text{id}^X \times \text{cross} = \text{id} \times \text{id} \]

where \( \text{id}^X \times \text{cross} = \text{id} \) is because \( L_1 \otimes \ldots \otimes L_k \) is a tensor product with inclusion \( \times \) and extension of functions given by \( \ldots \). Secondly

\[ \text{CON}_{X}(L_1, \ldots, L_k) \times \text{ABS}_{X}(L_1, \ldots, L_k) = \]

\[ \text{cross} \times \text{id}^X = \]

\[ \lambda_{J, \{t \otimes b_1 \in B_L, \ldots, t \otimes (b_1, \ldots, b_k) \}} \times \left( (b_1, \ldots, b_k) \in \left( \bigcup_{(b_1', \ldots, b_k') \in J} \bigotimes (b_1', \ldots, b_k') \right) \right) = \]

\[ \lambda_{J, J} = \text{id} \]

showing that the desired result holds.

That \( \text{CON}_{X} \) is a natural transformation over \( \text{ACL}_{su} \) follows because the diagram

\[ \begin{array}{ccc}
L_1 \otimes \ldots \otimes L_k & \xleftarrow{\text{cross}} & L_1^X \times \ldots \times L_k^X \\
\downarrow h_1 \otimes \ldots \otimes h_k & & \downarrow h_1^X \times \ldots \times h_k \\
L'_1 \otimes \ldots \otimes L'_k & \xleftarrow{\text{cross}} & L'_1^X \times \ldots \times L'_k
\end{array} \]

commutes for all \( \text{ACL} \) morphisms \( h_i \). To see this recall that \( h_1 \otimes \ldots \otimes h_k \)

was defined as \( (\text{cross} \times h_1 \times \ldots \times h_k)^X \) and it was stated in section 3.2 that \( f^X \times \text{cross} = f \) for all \( \text{ACL} \) morphisms \( f \).

To see that \( \text{ABS}_{X} \) is a natural transformation over \( \text{ACL}_{su} \) it suffices to show that

\[ \begin{array}{ccc}
L_1 \otimes \ldots \otimes L_k & \xrightarrow{\text{id}^X} & L_1^X \times \ldots \times L_k^X \\
\downarrow h_1 \otimes \ldots \otimes h_k & & \downarrow h_1^X \times \ldots \times h_k \\
L'_1 \otimes \ldots \otimes L'_k & \xrightarrow{\text{id}^X} & L'_1^X \times \ldots \times L'_k
\end{array} \]
commutes for all $ACL\alpha$ morphisms $h$. Commutativity of the diagram is verified by the calculation

$$id^x \cdot h_1 \otimes \ldots \otimes h_k = \ldots$$

$$id^x \cdot (\text{cross}^* h_1 \times \ldots \times h_k) = \ldots$$

(because $id^x$ is completely additive and it is straightforward to verify that $g \cdot f^x = (g \cdot f)^x$ whenever $g$ is completely additive)

$$(id^x \cdot \text{cross}^* h_1 \times \ldots \times h_k)^x = \ldots$$

(because $id^x \cdot \text{cross}^* = id$)

$$(h_1 \times \ldots \times h_k \cdot id)^x = \ldots$$

(as before because $h_1$ is completely additive and therefore so is $h_1 \times \ldots \times h_k$)

$$h_1 \times \ldots \times h_k \cdot id^x$$

**Note** The additivity of the $h_i$ cannot simply be dispensed with. As an example take $L_1 = L_2 = L'_1 = L'_2 = V$ where

![Diagram](image)

was defined in section 3.1. Next define $h_1 = h_2 = \lambda x. x^2$ where $x^2 = -$ etc. For $J = \{ t \mid t \in \varnothing(-,-) \varnothing(+,-) \}$ we have $((h_1 \times h_2) \cdot id^x)(J) = (T, T)$ whereas $(id^x \cdot (h_1 \varnothing h_2))(J) = (+, +)$.

One could also consider investigating a change from the independent attribute method $\times$ back to the relational method $\varnothing$. It is hard to do so, however, because it is hard to find a pair

$$(\text{abs}: L_1 \times \ldots \times L_k \rightarrow L_1 \varnothing \ldots \varnothing L_k, \text{con}: L_1 \varnothing \ldots \varnothing L_k \rightarrow L_1 \times \ldots \times L_k)$$

of adjoined functions (relative to $ACL\alpha$). This does not give cause for concern, however, as the proposed change does not seem to be of
interest for data flow analysis purposes.

**Example:** The domain constructor $\mathcal{X}$.

We first investigate changing from $I(\mathcal{X})=\emptyset$ to $J(\mathcal{X})=\mathcal{X}$. The proposed transformations are:

$$CON_{\mathcal{X}}(L_1, \ldots, L_k) = \text{cross}_{\mathcal{X}}$$

$$ABS_{\mathcal{X}}(L_1, \ldots, L_k) = \text{id}_{\mathcal{X}}$$

Clearly this defines morphisms in $\mathcal{ACLs}_{\mathcal{X}}$.

By calculations similar to those given for $\mathcal{X}$ it can be shown that the pairs $(ABS_{\mathcal{X}}(L_1, \ldots, L_k), CON_{\mathcal{X}}(L_1, \ldots, L_k))$ are (exactly) adjoined. Further, $CON_{\mathcal{X}}$ is a natural transformation from $\mathcal{X}$ to $\emptyset$ over $\mathcal{ACLs}_{\mathcal{X}}$ because (similarly to $\mathcal{X}$)

$$\text{cross} \cdot h_1 \cdot \ldots \cdot h_k = h_1 \emptyset \ldots \emptyset h_k \cdot \text{cross}$$

holds for all $\mathcal{ACLs}_{\mathcal{X}}$ morphisms $h_i$. Finally, $ABS_{\mathcal{X}}$ is a natural transformation from $\emptyset$ to $\mathcal{X}$ over $\mathcal{ACLs}_{\mathcal{X}}$ because (similarly to $\mathcal{X}$)

$$h_1 \mathcal{X} \ldots h_k \cdot \text{id}_{\mathcal{X}} = \text{id}_{\mathcal{X}} \cdot h_1 \emptyset \ldots \emptyset h_k$$

holds for all strongly strict $\mathcal{ACLs}_{\mathcal{X}}$ morphisms $h_i$ (as then $h_1 \mathcal{X} \ldots h_k$ is completely additive).

Next we investigate changing from $I(\mathcal{X})=\emptyset$ to $J(\mathcal{X})=\mathcal{X}$. For this define $ABS_{\mathcal{X}}$ and $CON_{\mathcal{X}}$ by the formulae explicitly exhibited above. It may be verified that $(ABS_{\mathcal{X}}(L_1, \ldots, L_k), CON_{\mathcal{X}}(L_1, \ldots, L_k))$ is a pair of adjoined functions (with $ABS_{\mathcal{X}}(L_1, \ldots, L_k) \cdot CON_{\mathcal{X}}(L_1, \ldots, L_k) = \text{smash}$). To show that $CON_{\mathcal{X}}$ is a natural transformation over $\mathcal{ACLs}_{\mathcal{X}}$ it suffices to
show that
\[ \text{CON}(L_1, \ldots, L_k) \cdot h \times \cdots \times h_k = h \circ \cdots \circ h_k \cdot \text{CON}(L_1, \ldots, L_k) \]
for all ACLs morphisms \( h_i \). To see this let \( \text{cross} : L_1 \times \cdots \times L_k \rightarrow L_1 \otimes \cdots \otimes L_k \) be as above and note that
\[ \text{cross}^* h \times \cdots \times h_k \cdot \text{smash} = h \circ \cdots \circ h_k \cdot \text{cross} \cdot \text{smash} \]
follows from the previous paragraph. The result then follows because \( h \times \cdots \times h_k \cdot \text{smash} \) equals \( \text{smash}^* h \times \cdots \times h_k \) and \( \text{cross} \cdot \text{smash} \) equals \( h \cdot \text{abs} \).

Finally, to show that \( \text{ABS} \) is a natural transformation over ACLs it suffices to show that
\[ h \times \cdots \times h_k \cdot \text{ABS}(\cdots) = \text{ABS}(\cdots) \cdot h \circ \cdots \circ h_k \]
holds for all strongly strict ACLs morphisms \( h_i \). This follows from the previous paragraph in much the same way as for \( \text{CON} \).

**Example:** The domain constructor \(+\).

We first define a locally continuous semi-functor \(\Theta\). On objects
\[ L_1 \otimes \cdots \otimes L_k = \{ i, \tau \} \cup \bigcup_i \{ (i, d) \mid d \in L_i \} \]
which is partially ordered by
\[ x \leq y \text{ iff } x = \bot \text{ or } y = \tau \text{ or for some } i, d, d' \text{ that } x = (i, d) \text{ and } y = (i, d') \text{ and } d \leq d' \]
If all \( L_i \) are algebraic complete lattices then so is \( L_1 \otimes \cdots \otimes L_k \) and
\[ B_1 \otimes \cdots \otimes B_k = \{ i \} \cup \bigcup_i B_{L_i}. \]
(If no new greatest element had been added and instead the \( (i, \tau) \) had been identified then algebraicity might be violated.) On morphisms
\[ h_1 \circ \cdots \circ h_k = \begin{cases} \tau & \text{if } x = \tau \\ \bot & \text{if } x = \bot \\ (i, h_1(d)) & \text{if } x = (i, d) \end{cases} \]
This defines a strict and continuous function when all \( h_i \) are continuous. Further the definition depends continuously on the \( h_i \). If all \( h_i \) are (completely) additive then so is \( h_1 \otimes \ldots \otimes h_k \). (This would not have been the case if no new least element had been added and instead the \((i, \bot)\) had been identified.) It is then straightforward to verify that \( \Theta \) is a locally continuous functor and semi-functor over \( \text{ACLS} \).

Next we consider changing from the relational method \( I(\pi) = \chi \) to the independent attribute method \( J(\pi) = \Theta \). The proposed transformations are

\[
\begin{align*}
\text{CON}_+(L_1, \ldots, L_k) &= \chi, \quad \text{if } x = \bot \\
\text{CON}_+(L_1, \ldots, L_k) &= \chi, \quad \text{if } x = \top \\
\text{CON}_+(L_1, \ldots, L_k) &= \chi(L_1, \ldots, L_k), \quad \text{if } x = (i, l)
\end{align*}
\]

\[
\text{ABS}_+(L_1, \ldots, L_k) = \chi(l_1, \ldots, l_k) = \begin{cases} 
1 & \text{if } \forall i: l_i = \bot \\
(i, l_i) & \text{if } \exists i: l_i = \bot \\
\top & \text{otherwise}
\end{cases}
\]

Clearly this specifies morphisms of \( \text{ACLS} \) and it is straightforward to verify that the definitions specify pairs of adjoined functions. That \( \text{CON}_+ \) is a natural transformation over \( \text{ACLS} \) is because the diagram

\[
\begin{array}{c}
L_1 \ldots X L_k \xleftarrow{\text{CON}_+(L_1, \ldots, L_k)} L_1 \Theta \ldots \Theta L_k \\
\downarrow h_1 \ldots X h_k \quad \downarrow h_1 \Theta \ldots \Theta h_k \\
L_1' \ldots X L_k' \xleftarrow{\text{CON}_+(L_1', \ldots, L_k')} L_1' \Theta \ldots \Theta L_k'
\end{array}
\]

commutes for all \( \text{ACLS} \) morphisms \( h_i \) such that \( h_i(\top) = \top \). That \( \text{ABS}_+ \) is a natural transformation over \( \text{ACLS} \) is because the diagram

197
The only bottom-level domain constructor that has not been considered so far is \( \bot \). One idea might be to change from \( I(\bot) = \bot \) to \( J(\bot) = \text{Id} \). It is unclear how this should be done because it is unclear how to find a pair

\[
(\text{abs:} \bot \rightarrow L, \text{con:} L \rightarrow \bot)
\]

of adjoined functions (relative to ACLs). For this reason no "change of method" will be considered for lifting.

### 4.4 Expected Definitions

The notion of inducing an interpretation allows for an automatic specification of the expression part of an approximating interpretation. It was shown in theorem 4.2:7 that this results in a very "precise" approximation of the original interpretation. There are circumstances, however, where it may be desirable to use more approximate versions of the functionals and some of the constants. These versions will be called expected definitions and in this section such definitions will be proposed and some of their properties studied.

Perhaps the most convincing example is composition \( \Box \) in the bottom-level metalanguage. Consider an approximating interpretation \( I \) with \( I(\Box)(g_1, g_2) = g_1 \circ g_2 \) and let \( J \) be induced from \( I \). Then
\( J(\mathcal{D}) (h_1, h_2) \) is of the form

\[
(\text{abs'con})' h_1' (\text{abs'con})' h_2' (\text{abs'con})
\]

which may differ from \( h_1' h_2 \) when \( \text{(abs, con)} \) is not a pair of exactly adjoined functions. One may contemplate using an approximating interpretation \( J' \) that is mostly as \( J \) but has \( J'(\mathcal{D}) (h_1, h_2) = h_1' h_2' \).

Motivations for doing so may be conceptual, that "composition" should be composition, or based on implementation considerations such as whether it is easier to implement or faster to execute. Indeed previous formulations of abstract interpretation have always made this assumption. Having defined functional composition as the expected definition of \( \mathcal{D} \) the question arises whether it is guaranteed to be safe to let both \( I \) and \( J' \) use the expected definitions. Formally the question is whether \( I(\mathcal{D}) = \lambda(g_1, g_2). g_1' g_2 \) and \( J'(\mathcal{D}) = \lambda(h_1, h_2). h_1' h_2 \) satisfy that

\[
I(\mathcal{D}) \cong_{\text{ft}, \text{ft}} J'(\mathcal{D})
\]

By lemma 4.2:6 this is equivalent to asking whether

\[
J(\mathcal{D}) \cong_{\text{id}, \text{ft}} J'(\mathcal{D}) \text{. It is straightforward to verify that this is the case.}
\]

Additional examples

In the remainder of this section a similar treatment is given for all entries in the first and second components of the expression part of an interpretation. For some (e.g. \( \text{in}_i \)) the expected definition depends on which semi-functor is used to interpret some bottom-level domain constructor (e.g. \( \text{+} \)). It will emerge that the collecting interpretation uses the expected definitions.
To facilitate the development the following assumptions are made. Let $I$ and $J$ be approximating interpretations such that $J$ is induced from $I$, i.e.

$$J = \text{induce}(I, (\text{abs, con}), J)$$

for a family $(\text{abs, con}) = (\text{abs}_t, \text{con}_t)$ of pairs of adjoined functions. For each of the bottom-level domain constructors (written $\#$) we shall assume that the "change in method" from $I(\#)$ to $J(\#)$ is one of those considered in section 4.3. This means that the index for $(I(\#), J(\#))$ in the following table for $\#$ must be a $\checkmark$.

<table>
<thead>
<tr>
<th>$\downarrow$</th>
<th>$#$</th>
<th>$\downarrow$</th>
<th>$#$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\downarrow$</td>
<td>$\checkmark$</td>
<td>$\downarrow$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
</tr>
</tbody>
</table>

Further we shall assume that $\text{abs}_t$ and $\text{con}_t$ are specified compositionally as proposed in section 4.3. Taking $\text{abs}_t$ as an example this means that

$$\text{abs}_t = \text{lengthen}_t(I, (\text{abs}_{A_1} A_1, (\text{ABS }_\#), J)$$

where $\text{ABS }_\#$ was specified in section 4.3. (In particular $\text{ABS }_\#$ is the identity transformation if $I(\#)$ equals $J(\#)$.) These assumptions suffice for most of the development but additional assumptions will be introduced to handle the expected definition of filter (used in cond) or when $J$ uses tensor products ($J(x) = \emptyset$ or $J(\#) = \emptyset$).

The development is by means of a list of examples. The proposal of expected definitions may be viewed as defining

$$J' = \text{expected-induce}(I, (\text{abs, con}), J)$$

that is $\text{induce}(I, (\text{abs, con}), J)$ modified so as to use the expected definitions. The proposals are guided by what holds in the collecting
interpretation. To show that it is safe to use the expected definitions amounts to proving

\[ I \leq_{\text{con}} \text{expected-induce}(I, (\text{abs, con}), J) \]

and the relevant cases for that proof is given in each example.

**Example:** cond.

Consider the following expected definition

\[ J'(\text{cond})(h_1, h_2, h_3) = \text{strict}(h_2) \cdot J'(\text{filter}_{\text{tt}})(h_1) \]
\[ \cup \text{strict}(h_3) \cdot J'(\text{filter}_{\text{ff}})(h_2) \]

where \( J'(\text{filter}_x) \) is not specified. There is no entry for filter in the expression part of an interpretation but this is not a serious problem because when \( J'(\text{cond}) \) is of the above form \( J'(\text{filter}_{\text{tt}})(h_1) \) equals \( J'(\text{cond})(h_1, \text{id, L}) \) and similarly for \( J'(\text{filter}_{\text{ff}})(h_1) \). If also \( I \) uses this expected definition it is straightforward to prove that

\[ I(\text{filter}_x) \leq_{\text{con}} J'(\text{filter}_x) \quad \text{when } x \text{ is } \text{tt} \text{ and when } x \text{ is } \text{ff} \]

implies \( I(\text{cond}) \leq_{\text{con}} J'(\text{cond}) \)

Here types have been omitted from \( \leq_{\text{con}} \) and this will often be the case in this section.

**Example:** fold and unfold.

It follows from section 2.2 that there is an isomorphism \( \theta \) from \( J'[\text{recX.gt}] \) to \( J'[\text{recX.gt}] \). It is natural to expect the definitions

\[ J'(\text{fold}) = \theta \quad \text{and } J'(\text{unfold}) = \theta^{-1} \]

If also \( I \) uses the expected definitions then

\[ \text{con}_{\text{recX.gt}} J'(\text{fold}) = I(\text{fold}) \cdot \text{ln}_{\text{gt}} (\text{con}_{\text{recX.gt}}) \]
is an easy consequence of lemma 4.3:4. Further it can be shown by structural induction (see e.g. the proof of theorem 3.3:4) that

\[ \text{con}_{gt[\text{recX}.gt/X]} = \text{ln}_{gt}(\text{con}_{\text{recX}.gt}) \]

This shows that \( \text{I}(\text{fold}) \leq_{\text{con}} \text{J'}(\text{fold}) \) and \( \text{I}(\text{unfold}) \leq_{\text{con}} \text{J'}(\text{unfold}) \).

///

Example: lift and up.

The only functor considered is \( \text{J'}(\_)=\_ \). This gives rise to the expected definitions

\[ \text{J'}(\text{lift})(h) = \lambda v. \text{def}(v) \rightarrow h(\text{down}(v)), \_ \]
\[ \text{J'}(\text{up}) = \text{up} = \lambda v.(0,v) \]

If also \( \text{I} \) uses these expected definitions it is easy to show that

\[ \text{I}(\text{up}) \leq_{\text{con}} \text{J'}(\text{up}) \]

from which \( \text{I}(\text{up}) \leq_{\text{con},gt \rightarrow gt'} \text{J'}(\text{up}) \) follows. It is equally straightforward to show that

\[ \text{I}(\text{lift}) \leq_{\text{con}} \text{J'}(\text{lift}) \]

///

Example: case and in _.

For + we have considered two semi-functors: X and Θ. Naturally enough the expected definitions depend on the semi-functor. When \( \text{J'}(+)=X \) they are

\[ \text{J'}(\text{case})(h_1,\ldots,h_k)(l_1,\ldots,l_k) = \lambda (l_1,\ldots,l_k) . h_1(l_1)\ldots h_k(l_k) \]
\[ \text{J'}(\text{in}_{\_}) = \lambda i. (\_,\ldots,\_i\ldots\_) \]

and when \( \text{J'}(+)=\Theta \) they are given by

\[ \text{J'}(\text{case})(h_1,\ldots,h_k)(\_) = \bigcup_{i=1}^{k} h_i(\_) \]
\[ \text{J'}(\text{case})(h_1,\ldots,h_k)(j,l_j) = (\bigcup_{i=1}^{k} h_i(\_)) \cup h_j(l_j) \]
\[ \text{J'}(\text{case})(h_1,\ldots,h_k)(\tau) = \tau \]
\[ \text{J'}(\text{in}_{\_}) = \lambda i. \_l_\rightarrow \_, (i,\_) \]

202
The need for U^k i=1 h_i(\lambda) only arises because it has not been assumed that morphisms are strict (see the remark in section 2.3). The definition given for J'(in_i) could be replaced by the non-strict \lambda 1(i, 1).

Suppose now that also I uses these expected definitions. The proof that I(in_i) \leq_{\text{con}} J'(in_i) and I(case) \leq_{\text{con}} J'(case) is by cases of whether (I(t), J'(t)) is (\lambda, \lambda) or (\lambda, \theta) or (\theta, \theta). For the first case I(in_i) \leq_{\text{con}} J'(in_i) amounts to showing

I(in_i) \leq_{\text{con}} (\text{con} \ldots \text{con} \text{gt}_i^{*}) J'(in_i)

which is immediate. To show I(case) \leq_{\text{con}} J'(case) assume that g_i \leq_{\text{gt}_1 \ldots \text{gt}_k} h_i and calculate

I(case)(g_1, \ldots, g_k) \leq_{\text{con} \text{gt}_1 \ldots \text{con} \text{gt}_k} \lambda(1, \ldots, l_k). g_i(\text{con} \text{gt}_1^{*} \ldots \text{con} \text{gt}_k^{*}) \leq_{\text{con}} \lambda(1, \ldots, l_k). \text{con} \text{gt}_1^{*} \ldots \text{con} \text{gt}_k^{*} \text{gt}(h_i) \leq_{\text{con} \text{gt}_1^{*} \ldots \text{con} \text{gt}_k^{*} \text{gt}_1^{*} \ldots \text{con} \text{gt}_k^{*}}

For the second case I(in_i) \leq_{\text{con}} J'(in_i) is by a straightforward calculation. To show I(case) \leq_{\text{con}} J'(case) assume that g_i \leq_{\text{gt}_1^{*} \ldots \text{gt}_k^{*}} h_i and calculate (using the corollary to lemma 4.3:1)

I(case)(g_1, \ldots, g_k) \leq_{\text{con} \text{gt}_1^{*} \ldots \text{con} \text{gt}_k^{*}} \lambda(1, \ldots, l_k). g_i(\text{con} \text{gt}_1^{*} \ldots \text{con} \text{gt}_k^{*}) \leq_{\text{con} \text{gt}_1^{*} \ldots \text{con} \text{gt}_k^{*} \text{gt}(h_i) \leq_{\text{con} \text{gt}_1^{*} \ldots \text{con} \text{gt}_k^{*} \text{gt}_1^{*} \ldots \text{con} \text{gt}_k^{*}}

For the third case I(in_i) \leq_{\text{con}} J'(in_i) is again straightforward and for I(case) \leq_{\text{con}} J'(case) assume that g_i \leq_{\text{gt}_1^{*} \ldots \text{gt}_k^{*}} h_i and calculate

I(case)(g_1, \ldots, g_k) \leq_{\text{con} \text{gt}_1^{*} \ldots \text{con} \text{gt}_k^{*}}
where case is defined by the same formula that defined \( \mathcal{I}(\text{case}) \) and 
\( \mathcal{J}'(\text{case}) \) (but the functionalities differ).

Example: tuple and \( \text{take}_1 \).

For \( x \) the two semi-functors \( \mathcal{O} \) and \( \mathcal{X} \) have been considered. When
\( \mathcal{J}'(x) = \varnothing \) the expected definitions will be

\[
\mathcal{J}'(\text{tuple})(h_1, \ldots, h_k) = \text{lin}(\underline{\text{cross}}_x \lambda l. (h_1(l), \ldots, h_k(l)))
\]

\[
\mathcal{J}'(\text{take}_1) = \lambda l. \text{id}^x(l) \downarrow l
\]
as was used in the collecting interpretation. The choice for \( \mathcal{J}'(\text{take}_1) \)
is hardly unnatural and the definition of \( \mathcal{J}'(\text{tuple}) \) will be further
motivated below. When \( \mathcal{J}'(x) = x \) it is natural to take

\[
\mathcal{J}'(\text{tuple})(h_1, \ldots, h_k) = \lambda l. (h_1(l), \ldots, h_k(l))
\]

\[
\mathcal{J}'(\text{take}_1) = \lambda (l_1, \ldots, l_k). l_1
\]
as the expected definitions.

To motivate the "expected definition" for \( \mathcal{J}'(\text{tuple}) \) when
\( \mathcal{J}'(x) = \varnothing \) we shall consider the following example. Recall the cpo

\[
\mathcal{V} = \begin{array}{ccc}
\vdots & 0 & + \\
\downarrow & \downarrow & \downarrow \\
\end{array}
\]

where the intuitive meaning of the elements is given by the
concretization function \( \text{con}: \mathcal{V} \rightarrow 0(\ldots, -1, 0, 1, \ldots) \) defined by
\( \text{con}(\downarrow) = \downarrow \), \( \text{con}(\downarrow) = \{ \ldots, -1, 1 \} \) etc. Next consider the "diagonal"
function \( \mathcal{J}'(\text{tuple})(\text{id}, \text{id}): \mathcal{V} \rightarrow \mathcal{V} \). When applied to \( \top \) it will be
claimed that the desired result is $J_1 = \{ t | t \mathcal{E} \langle \overline{\mathcal{O}}(-, -) \mathcal{O}(0, 0) \mathcal{O}(+, +) \} \),

rather than e.g. $J_2 = \{ t | t \mathcal{E} \mathcal{O}(\tau, \tau) \} \). This is because $J_1$ expresses as precisely as possible that "the two components are really the same" quite unlike $J_2$. As an example of this $J_2$ contains $\mathcal{O}(-, +)$ but $J_1$ does not. If $J'(\text{tuple})(h_1, \ldots, h_k)$ had simply been cross $\lambda l. (h_1(l), \ldots, h_k(l))$ then $J_2$ would be obtained whereas the use of lin ensures that $J_1$ is obtained. The definition of lin given in section 3.3 was

$$\text{lin}(g) = \lambda l. \bigcup \{ g(l) \mid l \in A \land l \in \text{IB} \}$$

but for the purposes of section 3.3 PB could have been used instead of IB. This is not the case here because $\text{IB}_V = V - \{ \tau \}$ whereas $\text{PB}_V = \{ l \}$. Also the fact that IB is the set of finite and essential elements (see section 4.1) may be viewed as suggesting that IB is the right choice (or at least better than PB).

In the above discussion it was tacitly assumed that con is irreducibly generated. For had con been specified as above except that it was of functionality

$$\text{con}: V \rightarrow \mathcal{F}(\{ \ldots, -1, 0, 1, \ldots, tt, ff \} \cup)$$

then $J_1$ would intuitively be incorrect and $J_2$ should be used instead. It is therefore not surprising that the condition that abs specializes to abs:IB$\rightarrow$IB (see lemma 4.1:4) will be used when showing $I(\text{tuple}) \leq_{\text{con}} J'(\text{tuple})$ in the case where $J'(X) = \emptyset$. To make it clear which parts of the developments can be performed without such assumptions this case will be postponed.

So suppose that $I$ and $J'$ both use the expected definitions and that $I(X) = \emptyset$ and $J'(X) = X$. That $I(\text{take}_I) \leq_{\text{con}} J'(\text{take}_I)$ follows from calculating
\[
I(\text{take}_i) \cdot \text{cross} \cdot \text{con}_{gt_1} \cdot x \cdots \text{con}_{gt_k} = \\
(\lambda l.1\downarrow i) \cdot \text{id} \cdot \text{cross} \cdot \text{con}_{gt_1} \cdot x \cdots \text{con}_{gt_k} = \\
\text{con}_{gt_i} \cdot (\lambda l.1\downarrow i) = \text{con}_{gt_i} \cdot J'(\text{take}_i)
\]

For \(I(\text{tuple}) \triangleq \text{con}_{gt_i} \cdot J'(\text{tuple})\) assume that \(g_1 \cdot \text{con}_{gt_i} \cdot \text{con}_{gt_i} \cdot h_i\) and calculate

\[
I(\text{tuple})(g_1, \ldots, g_k) = \text{con}_{gt_i} \cdot (\lambda l.1\downarrow i) = \text{con}_{gt_i} \cdot J'(\text{tuple})(h_1, \ldots, h_k)
\]

When both \(I(\mathcal{X})\) and \(J'(\mathcal{X})\) are \(\mathcal{X}\), the calculations for \(I(\text{take}_i) \triangleq J'(\text{take}_i)\) and \(I(\text{tuple}) \triangleq J'(\text{tuple})\) are even more straightforward. ///

Remark Taking \(\mathcal{X}\) as an example one may consider whether

\[
\text{con}_{gt_1} \cdot x \cdots \text{con}_{gt_k} = I(\text{tuple})(\text{con}_{gt_i} \cdot J'(\text{take}_1), \ldots)
\]

and similarly for abs \(\text{con}_{gt_1} \cdot x \cdots \text{con}_{gt_k}\). If true this would give a pleasant connection between the semi-functors and natural transformations and the functionals and constants. The equality holds in some cases but fails when \(I(\mathcal{X}) = \emptyset\) and \(J(\mathcal{X}) = \mathcal{X}\): if \(\text{con}_{gt_i} = \text{id}\) it reduces to

\[
\text{cross} = \text{lin} (\text{cross})
\]

which does not hold in general. ///

Example: \(\text{smashtuple}\) and \(\text{smashtake}_i\).

For \(\mathcal{X}\) the three semi-functors \(\emptyset, \cdot \) and \(\mathcal{X}\) have been considered. When \(J'(\mathcal{X}) = \emptyset\) the expected definitions are

\[
J'(\text{smashtuple})(h_1, \ldots, h_k) = \\
\text{lin} (\text{cross} \cdot \text{smashtake}_i \cdot \lambda l. (h_1(l), \ldots, h_k(l)))
\]

\[
J'(\text{smashtake}_i) = \lambda l. \text{id} (l) \downarrow i
\]

206
as in the collecting interpretation. When \( J'(X) = Y \) they are

\[
J'(\text{smashtuple})(h_1, \ldots, h_k) = \text{smash}^* \lambda l. (h_1(1), \ldots, h_k(1)) \\
J'(\text{smashtake}_i) = \lambda l. 1^i
\]

and when \( J'(X) = X \) they are

\[
J'(\text{smashtuple})(h_1, \ldots, h_k) = \lambda l. (h_1(1), \ldots, h_k(1)) \\
J'(\text{smashtake}_i) = \lambda l. 1^i
\]

If both \( I \) and \( J' \) use expected definitions then there are five cases of \( (I(X), J'(X)) \) to consider in the proof of

\[ I(\text{smashtake}_i) \leq \text{con} \ J'(\text{smashtake}_i) \text{ and } I(\text{smashtuple}) \leq \text{con} \ J'(\text{smashtuple}). \]

As in the previous example the case \((\emptyset \emptyset)\) will be postponed. The remaining cases are tedious and are omitted as they are reasonably straightforward.

Making additional assumptions

It has already been said that the examples given so far constitute a definition of

\[ J' = \text{expected-induce}(I, (\text{abs}, \text{con}), J) \]

An exception is that no expected definitions for filter \( X \) have been proposed. Furthermore, the examples constitute a proof of

\[ I \leq \text{con} \ J' \text{ except when } J'(X) = \emptyset \text{ or } J'(X) = \emptyset. \]

These omissions will now be rectified but additional assumptions about the abstraction functions \( \text{abs} \) are needed for this.

In all these cases the same problem comes up. The desired result is of the form

\[ \text{lin}(g) \leq \text{con,ft} \text{ lin}(h) \]
where \( g \leq_{\text{con, ft}} h \) is fairly straightforward to achieve. This is not, in general, sufficient for the desired result. As an example let

\[
M = \begin{array}{c}
1 \\
\downarrow \\
2
\end{array}
\]

and \( \text{con}: \mathcal{M} \to \mathcal{P}(\mathcal{L}) \) be defined by \( \text{con}(\tau) = \mathcal{L} \), \( \text{con}(1) = \{1, \perp\} \) etc. Further let \( g(Y) = \{1 + (y \div 3) \mid y \in \mathcal{L}, y \neq \perp\} \) and \( h = \text{abs' } g \cdot \text{con} \) for \( \text{abs} \) the lower adjoint of \( \text{con} \). Then \( g \leq_{\text{con, ft}} h \) holds but \( \text{lin}(g) \leq_{\text{con, ft}} \text{lin}(h) \) fails because \( (\text{lin}(g) \cdot \text{con})(\tau) = \mathcal{L} \) whereas \( (\text{con' } \text{lin}(h))(\tau) = \{1, \perp\} \). It is immediate that the concretization function \( \text{con} \) is not irreducibly generated. By lemma 4.1:4 this means that the abstraction function does not specialize to \( \text{abs': IB} \to \text{IB} \). The importance of this is expressed in:

**Lemma 4.4.1.** Let \((\text{abs}_{\text{gt}}, \text{con}_{\text{gt}})_{\text{gt}}\) be a family of pairs of adjoined functions such that \( g \leq_{\text{con}, \text{gt} \to \text{gt}} h \). Then \( \text{lin}(g) \leq_{\text{con}, \text{gt} \to \text{gt}} \text{lin}(h) \) holds provided that \( \text{abs}_{\text{gt}} \) specializes to \( \text{abs}_{\text{gt}}: \text{IB} \to \text{IB} \).

**Proof** Note that \( g \leq_{\text{con}, \text{gt} \to \text{gt}} h \) amounts to \( \text{abs}_{\text{gt}}' \cdot \text{lin}(g) \leq \text{abs}_{\text{gt}}' \cdot \text{lin}(h) \). The result then follows from the calculation:

\[
\text{abs}_{\text{gt}}' \cdot \text{lin}(g) = \lambda \, \bigcup \{ (\text{abs}_{\text{gt}}', g)(i) \mid i \in \text{IB} \} \subseteq \lambda \, \bigcup \{ h(\text{abs}_{\text{gt}}(i)) \mid i \in \text{IB} \} \subseteq \lambda \, \bigcup \{ h(\text{abs}_{\text{gt}}(i)) \mid i \in \text{IB} \} \subseteq \text{abs}_{\text{gt}}(\text{abs}_{\text{gt}}(1)) \subseteq \text{IB}
\]

where it has been used that \( \text{abs}_{\text{gt}}' \) is completely additive and that \( i \in \text{IB} \) implies \( \text{abs}_{\text{gt}}(i) \in \text{IB} \).

//
Example: tuple and \( \text{take}_i \) (continued).

It remains to consider the case where \( I(x) \) and \( J'(x) \) are both \( \circlearrowleft \).

For \( I(\text{take}_i) \leq \text{con} \ J'(\text{take}_i) \) it suffices to calculate:

\[
I(\text{take}_i) \cdot \text{con} \gt x \ldots \gt k = (\lambda 1.1 \cdot \text{id}^x \cdot \text{con} \gt 1 \ldots \gt k) =
\]

\[
(\lambda 1.1) \cdot \text{id}^x \cdot \text{con} \gt 1 \ldots \gt k
\]

(as in the example for \( x \) in section 4.3)

\[
(\lambda 1.1) \cdot \text{con} \gt x \ldots \gt k
\]

\[
(\lambda 1.1) \cdot \text{id}^x \cdot \text{con} \gt 1 \ldots \gt k
\]

For \( I(\text{tuple}) \leq \text{con} \ J'(\text{tuple}) \) first assume \( g_i \cdot \text{con} \gt \text{con} \gt \text{h}_i \) and calculate

\[
\text{cross}^* (\lambda_1.g_1(1), \ldots, g_k(1)) \cdot \text{con} \gt \text{con} \gt \text{x} \cdot \text{con} \gt 1 \ldots \gt k \cdot \text{con} \gt \text{x} \cdot \text{con} \gt \text{con} \gt \text{con} \gt 1 \ldots \gt k
\]

(by definition of \( \circlearrowleft \) upon morphisms and by \( \text{cross}=f \))

\[
\text{con} \gt 1 \ldots \gt \text{con} \gt 1 \ldots \gt \text{con} \gt \text{con} \gt \text{con} \gt 1 \ldots \gt k \cdot \text{con} \gt 1 \cdot \text{con} \gt 1 \ldots \gt \text{con} \gt k
\]

If it is assumed that all abs \( \text{gt} \) specialize to abs \( \text{gt}:IB \rightarrow IB \) then the result follows by the previous lemma.

Example: smashtuple and smashtake\( _i \) (continued).

The calculations and additional assumptions needed are much as above so the details will be omitted.

Example: filter.

In the expected definition for \( J'(\text{cond}) \) the functional \( J'(\text{filter}_x) \) was used and the proof of \( I(\text{cond}) \leq \text{con} \ J'(\text{cond}) \) was reduced to that of \( I(\text{filter}_x) \leq \text{con} \ J'(\text{filter}_x) \) for \( x=tt \) and \( x=ff \). As the expected definition for \( J'(\text{filter}_x) \) one might take
\[ J'(\text{filter}_x)(h) = \text{lin}(\lambda l. \ h(l) \not\exists j_x \rightarrow 1, 1) \]

where \( j_x \in \mathcal{T} \) is the representation in \( J' \) of the truthvalue \( x \).

(The collecting interpretation used \{x\}_R.) Rewriting this as

\[ J'(\text{filter}_x)(h) = \lambda l. \ \bigcup [i \in I_B \mid i \not\in 1 \ \lambda j_x \ \mathcal{E} h(i)] \]

shows that the effect of \( J'(\text{filter}_x)(h)(1) \) is to take the part of \( 1 \) upon which \( h \) "evaluates to" \( x \).

Suppose further that \( I(\text{filter}_x) \) is also of this form but with

\( i \in \mathcal{T} \) representing the truthvalue \( x \). The proof of

\( I(\text{filter}_x) \not\in \mathcal{T} \) \[ J'(\text{filter}_x) \]

will next be reduced to that of

\( \text{abs}_T (i, j_x) \). So assume that \( g \not\in \mathcal{T} \) and that \( \text{abs}_T (i, j_x) \)

and calculate:

\[
(\lambda l. \ g(l) \not\exists i_x \rightarrow 1, 1)' \not\in \mathcal{T} = \\
\lambda m. (g \not\in \mathcal{T} (m) \not\exists i_x \rightarrow \not\in \mathcal{T} (m), 1 \not\in \mathcal{T} = \\
\not\in \mathcal{T} (h(m) \not\exists i_x \rightarrow m, 1 \not\in \mathcal{T} = \\
\not\in \mathcal{T} (h(m) \not\exists j_x \rightarrow m, 1 \not\in \mathcal{T}
\]

If it is further assumed that \( \text{abs}_g \) specializes to \( \text{abs}_g : IB \rightarrow IB \)

then the result follows by the previous lemma.

It remains to investigate when all abstraction functions \( \text{abs}_g \)

specialize to \( \text{abs}_g : IB \rightarrow IB \). The lemma below plays a key role in this.

Recall that \( \text{abs}_g \) was defined in section 4.3 as

\[ \text{abs}_g = \text{lengthen}_g (I, (\text{abs}_A)^A, (\text{ABS}_A)'(I'))() \]

Let \( \text{ACLs}_{\mathcal{L}_I} \) be the subcategory of \( \text{ACLs}_{\mathcal{L}_I} \) of those morphisms \( \text{abs} \) that specialize to \( \text{abs}: IB \rightarrow IB \).

Lemma 4.4.2. Let \( \text{abs}_A \) be morphisms of \( \text{ACLs}_{\mathcal{L}_I} \) and suppose that each

210
ABS \((L_1, \ldots, L_k)\) is in ACLsli. Further suppose that for each bottom-level domain constructor \# that \(I(\#)\) or \(J'(\#)\) specializes to a functor over ACLsli. Then all \(\text{abs}^{\#} \) are morphisms of ACLsli. \\

**Proof** The result follows from the stronger result:

- if additionally \(h_i: L_i \to L_i'\) are morphisms of ACLsli
- then so is \(\text{id}(I, \text{abs}, \text{ABS}, J')((h_1, \ldots, h_n))\)

that is proved by structural induction on \(\text{id}\) such that \(V\text{id}\) for some \(V\) with \(\text{card}(V)=N\). The cases \(\text{id}=\Lambda_i\) and \(\text{id}=X\) are immediate. The case \(\text{id}=\text{id} \# \ldots \# \text{id}_k\) (where \(\#\) is one of \(X, \, \star, \, +\) or \(\Lambda\)) is straightforward if \(I(\#)\) is a functor over ACLsli and by corollary 4.3:2 also if it is \(J'(\#)\) that is a functor over ACLsli.

For the case \(\text{id}=\text{rec}X, \text{id}_0\) we use the notation from section 4.3. It is a straightforward numerical induction to show that all \(\text{LN}_{\text{id}_0, n}\) are in ACLsli. It follows from lemma 4.3:1 that \(h = \bigcup_n s^\text{LT}_{\text{id}_0, n}^\text{id}^\text{U}\) is in ACLsli so it suffices to show that \(h(i)\) is irreducible when \(i \in I\). First note that \(h(i) = (s^\text{LT}_{\text{id}_0, n}^\text{id}^\text{U})(i)\) for sufficiently large \(n\) because \(h(i)\) is finite. Secondly we show that \(r_n^\text{id}(i)\) is finite and irreducible for sufficiently large \(n\). Note that

\[
r_n^\text{id}(r_n^\text{id}(i)) = i\;
\]

when \(n\) is sufficiently large because \(i\) is finite and \(\bigcup_m r_n^\text{id} \equiv \text{id}\). To see that \(r_n^\text{id}(i)\) is finite suppose that

\[
r_n^\text{id}(i) \equiv \bigcup_m d_m.
\]

Then \(i = r_n^\text{id}(r_n^\text{id}(i)) \equiv \bigcup_m r_n^\text{id}(d_m)\) so there exists an \(m\) such that \(i \equiv r_n^\text{id}(d_m)\) and then \(r_n^\text{id}(i) \equiv r_n^\text{id}(r_n^\text{id}(d_m)) \equiv d_m\). To see that \(r_n^\text{id}(i)\) is irreducible let \(r_n^\text{id}(i) = d_1 \ldots d_2\). Because \(r_n\) is additive \(i = r_n^\text{id}(d_1) \ldots r_n^\text{id}(d_2)\) so \(i=r_n^\text{id}(d_k)\) for some \(k\) and it follows that \(r_n^\text{id}(i) = d_k\).

Suppose now that \(n\) is large enough that \(r_n^\text{id}(i)\) is finite and irreducible and \(h(i)\) equals \(s_n^\text{LT}_{\text{id}_0, n}(r_n^\text{id}(i))\). Because \(\text{LN}^\#_{\text{id}_0, n}\) is in ACLsli it follows that \(j = \text{LN}^\#_{\text{id}_0, n}(r_n^\text{id}(i))\) is finite and
irreducible. Now \( s_n \) is an embedding of \( ACLsa \) and therefore \( s_n(j) \) is irreducible. For if \( s_n(j) = d_1 \cup d_2 \) then \( j = s_n^U(s_n(j)) = s_n^U(d_1) \cup s_n^U(d_2) \) because \( s_n^U \) is additive. So there exists a \( k \) such that \( j = s_n^U(d_k) \) and it follows that \( s_n(j) = s_n(s_n^U(d_k)) \subseteq d_k^U \). But \( s_n(j) \not\subseteq d_k \) is trivial so \( s_n(j) = d_k \) follows. Hence \( h(i) \) is irreducible.  

The conditions of the above lemma may be made more concrete by using the assumptions about the connections between \( I(#) \) and \( J'(#) \) (which is \( J(#) \)) that were listed in the beginning of this section.

Lemma 4.4.3. The conditions of lemma 4.4.2 hold iff

- all abs \( \Lambda_i \) specialize to abs \( \Lambda_i : IB \rightarrow IB \), and
- \( I(x) = J'(x) \), and
- \( I(x) = J'(x) \neq x \)

Proof The proof relies heavily upon information about the finite and irreducible elements so we begin with stating this information. It is straightforward to show that

\[
IB_L \times \ldots \times L_k = \bigcup_{i=1}^{k} \{ (i) \times L_i \times \ldots \times L_k \}
\]

\[
IB_L \times \ldots \times L_k = \bigcup_{i=1}^{k} \{ (i, b_i) \mid b_i \in IB_L \}
\]

\[
IB_L \times \ldots \times L_k = \{ (i) \cup \{ (i, \text{up}(b)) \mid b \in IB_L \} \}
\]

It is a consequence of lemma 3.2:19 that

\[
IB_L \times \ldots \times L_k = \{ \text{cross}(b_1, \ldots, b_k) \mid \forall i: b_i \in IB_L \}
\]

\[
IB_L \times \ldots \times L_k = \{ \text{cross(smash}(b_1, \ldots, b_k) \mid \forall i: b_i \in IB_L \}
\]

We sketch the proof for \( \times \). From theorem 3.1:3 it follows that

\[
IB_L \times \ldots \times L_k = \{ \bigcup_{i=1}^{k} \text{cross}(b_i, \ldots, b_k) \mid b_i \in IB_L \}
\]

and it is then easy to see that \( \times \) holds above. To see that \( \times \) holds note that if \( b_i \in IB_L \) then \( \times(b_1, \ldots, b_k) \) is irreducible by lemma 3.2:19 and if \( \text{cross}(b_1, \ldots, b_k) \) was not irreducible this would give a contradiction.
Finally we state without proof that

\[ \text{IB}_{L_1 \ldots L_k} = \{ (l_1, \ldots, l_k) \cup \{ (b_1, \ldots, b_k) | \exists i: \forall j: b_i \in \text{IB}_{L_i} \land b_i \neq \perp \land (\forall j: j \neq 1 \Rightarrow b_j \text{ is an atom}) \} \]

where \( b_j \) is said to be an atom iff \( b_j \not\in \perp \) implies that \( d = b_j \) or \( d = \perp \).

Turning to the proof for "if" it is straightforward to show that all of \( \emptyset, \&, x, \Theta, \perp \) give functors over \( \text{ACLsLI} \) (they give functors over \( \text{ACLsL} \) because they are locally continuous semi-functors). For the natural transformations it is evident that \( \text{ABS}_\emptyset (L_1, \ldots, L_k) \) is in \( \text{ACLsLI} \) when \( I(\emptyset) = J(\emptyset) \) (because it is the identity). The only situation where this is not so is when \( I(\emptyset) = X \) and \( J(\emptyset) = \emptyset \) and here it is straightforward to verify that \( \text{ABS}_{\emptyset} (L_1, \ldots, L_k) \) is in \( \text{ACLsLI} \).

Concerning the proof for "only if" it is straightforward to see that \( I(X) = J'(X) \) because otherwise

\[ \text{ABS}_X (L_1, \ldots, L_k) : L_1 \emptyset \ldots \emptyset L_k \rightarrow L_1 X \ldots X L_k \]

is not in \( \text{ACLsLI} \). Similarly \( I(X) = J'(X) \) is required. That \( J'(X) \neq X \) is because \( X \) does not give a functor over \( \text{ACLsLI} \). As an example define

\[
L_1 = (\{0,1\}, \geq) \\
L_2 = (\{0,1,2,\ldots\}, \geq)
\]

and \( \text{abs}: L_1 \rightarrow L_2 \) by \( \text{abs}(0) = 0 \) and \( \text{abs}(1) = \omega \). Then \( \text{abs} \) is in \( \text{ACLsLI} \) but \( \text{abs}: L_1 \times L_1 \rightarrow L_2 \times L_2 \) is not.

In discussing the implications of these results let us assume that \( I \) is the collecting interpretation and let us forget filter for a moment. The motivation for assuming that \( \text{abs}_{gt} \) specializes to \( \text{abs}_{gt}: \text{IB} \rightarrow \text{IB} \) was to handle the case where \( J'(X) = \emptyset \) or \( J'(X) = \emptyset \). If
this property of abs \( \gt \) is to be guaranteed by lemma 4.4:2 it follows
from lemma 4.4:3 that once a change from relational method to
independent attribute method takes place for one of \( X \) or \( X' \) then it
should take place for the other as well. This is a bit restrictive
but probably not a problem for practical applications.

Concerning filter the state of affairs is less ideal. Once a
change from relational method to independent attribute method has
taken place (for \( X \) or \( X' \)) then the expected definition no longer can
be relied upon. It is, of course, possible to remove the use of \( \text{lin} \)
but this means that a substantial amount of approximation takes
place. It is not clear what other choice would be better.

Remark It might be worthwhile to investigate the following idea but
to do so some changes in the metalanguage and the definition of
interpretations will be needed. The idea is to work in the subcategory
of ACLs where the irreducible finite elements of an object are the
least element and the atoms (they were defined in the previous proof).
It is probably not overly restrictive to assume this for the bottom-
level constant types \( \Lambda_i \). (In fact the "type structures" of /WaSh77/
are still included.) This does not work well with \( 1, X \) and \( \Theta \). So
one might remove \( 1 \) and \( X \) from the metalanguage. (Luckily they were
not used much in the examples in section 2.4.) One should disallow
\( J'(X) = X \) and redefine \( \Theta \) to identify the \((i,1)\) with \( 1 \). (It then does
not give a semi-functor but it is still a functor that preserves
lower adjoints.) This should be sufficient for a modified development
where the second and third (1) conditions in lemma 4.4:3 vanish. Then
one can still use \( \text{lin} \) in the expected definition of filter. //
4.5 PRAGMATICS OF THE TENSOR PRODUCT

The distinction between relational data flow analysis methods and independent attribute methods is an intuitive one. It is relevant for all bottom-level domain constructors (except possibly \( \perp \)) and we shall begin with \( \perp \). If \( L_i \) are "useful" approximations to \( \rho(D_i) \) then which of \( L_1 \times \ldots \times L_k \) or \( L_1 \Theta \ldots \Theta L_k \) should be used to approximate \( \rho(D_1 + \ldots + D_k) \)? First, it is intuitively clear that use of \( L_1 \times \ldots \times L_k \) is likely to give more precise results than use of \( L_1 \Theta \ldots \Theta L_k \). (Note that if \( \Theta \) had identified the \((i, \perp)\) with \( \perp \) then there would be an abstraction function from \( L_1 \times \ldots \times L_k \) to \( L_1 \Theta \ldots \Theta L_k \) such that it and the corresponding concretization function is a pair of exactly adjoined functions.) Secondly, if \( L_i = \phi(E_i) \) then \( L_1 \times \ldots \times L_k \) is isomorphic to \( \phi(E_1 \ldots + E_k) \) and it is intuitively clear that this would give as precise results as possible (at least if the abstraction functions from \( \rho(D_i) \) to \( \rho(E_i) \) were defined using representation functions). For these reasons we shall say that use of \( \times \) (for \( \perp \)) gives a relational method and use of \( \Theta \) (for \( \perp \)) gives an independent attribute method. For \( \times \) it is \( \Theta \) and \( \times \) respectively and for \( \perp \) it is \( \Theta \) and \( \ast \).

Intuitively, there is one catch in the above. There might be many constructors that agree on powerdomains but disagree elsewhere. To be a bit more precise the above considerations only "determine" the relational method as a functor over \( PD \) that is defined as the image of \( ACCs \) under \( \rho \). Given that the relational method for \( \perp \) is \( \times \) over \( PD \) it seems intuitively clear that it should be \( \times \) also over \( ACCs \). This is not necessarily as clear for the tensor products so in the remainder of this section we shall increase the confidence in the use of the tensor product \( \Theta \). Such considerations have not
previously been made when formulating data flow analysis denotionally. For example /Don78/ only considers $\pi$ for $x$ and $+$ for $t$.

The key idea is to define the notion of a "weak product". It is closely related to a tensor product with respect to "irreducibly generated" rather than "additive". It is convenient to perform the development in the following subcategory $\mathbf{ACL}_I$ of $\mathbf{ACL}$. An object $L$ is included iff

$$B_L = \{ib_1 \ldots ib_n | n \geq 0, j: ib_j \in B_L\}$$

and a morphism $f : L \to L'$ iff both $L$ and $L'$ are in $\mathbf{ACL}_I$. It is not overly restrictive to consider this category for it contains all powerdomains and also all objects $L$ (of $\mathbf{ACL}$) such that there is a pair

$$(\text{abs}: \rho(D) \to L, \text{con}: L \to \rho(D))$$

of exactly adjoined functions with con irreducibly generated. Any additive morphism in this category is irreducibly generated. The weak product (in $\mathbf{ACL}_I$) of the objects $L_i$ is an object $L_1 \times \ldots \times L_k$ and a morphism $\text{unit}: L_1 \times \ldots \times L_k \to L_1 \times \ldots \times L_k$ such that

- unit is separately additive
- for each separately irreducibly generated $f: L_1 \times \ldots \times L_k \to L$

there exists precisely one irreducibly generated $f : L_1 \times \ldots \times L_k \to L$ such that $f \circ \text{unit} = f$.

Why is the property of a weak product more "intuitive" than that of the tensor product? That unit (or cross in the tensor product) is separately additive can be paraphrased as follows. (Think of $L_i$ as the complete lattice $V$ with elements $\bot, \top, 0, +, 1$.) It is possible to distinguish between combinations of arguments (e.g. $\{(+, -), (-, +)\}$ versus $\{(+, -), (+, +), (-, +)\}$) but this must not be used
to avoid the identifications in some argument that is prescribed by
the least upper bound structure upon $L_i$ (e.g. $\{(+,-),(-,-),(-,+)\}$ cannot
be distinguished from $\{(+,-),(-,-),(0,-)\}$). The difference between
the weak product and the tensor product is the condition assumed of
$f$ and guaranteed for $f^W$. The problem with the tensor product is
that separate additivity is unlikely to hold for $f$. (As an example
consider $f: V \to V$ defined informally by $f(x) = x^2$. This function is not
separately additive so the tensor product property seems useless.)
The condition that $f$ is separately irreducibly generated, i.e.
\[
f(l_1, \ldots, l_k) = \bigcup \{ f(i_1, \ldots, i_k) \mid \forall j: i_j \in L_j \} \ni i_{ib_j} e_{IB_j}
\]
is likely to hold much more often. (But one should be careful for
the property is not preserved by composition.) It says that $f$ is
given as the best (i.e. $\bigcup$) description of the results of all
combinations of essential arguments (i.e. $(i_1, \ldots, i_k)$) that is
permitted by the actual argument.

The purpose of $f^W$ and $L_1 \otimes \cdots \otimes L_k$ then is to allow more precision
in which combinations to consider. This is made clearer by

**Lemma 4.5.1.** If $L_1 \otimes \cdots \otimes L_k$ and unit is a weak product then

- $f^W(x) = \bigcup \{ f(i_1, \ldots, i_k) \mid \text{unit}(i_1, \ldots, i_k) \in X \forall j: i_j \in L_j \}$

**Proof** We begin with showing $\subseteq$ in the first result. For the sake of
contradiction suppose that $i_b$ is an element of the lefthand side
but not of the righthand side. Define $L = \{ \{ \} \}$ and $f: L_1 \otimes \cdots \otimes L_k \to L$ by

\[
f(l_1, \ldots, l_k) = 1 \iff \forall (i_1, \ldots, i_k) \in \text{IB}_L \times \text{IB}_L : \left( (i_1, \ldots, i_k) \in (l_1, \ldots, l_k) \Rightarrow \text{unit}(i_1, \ldots, i_k) \in \text{ib} \right)
\]

This gives a continuous and separately irreducibly generated function
(but it is not separately additive). Next define $g_1: L_1 \times \cdots \times L_k \to L$ by
\[
g_1(x) = \bot \quad \text{iff} \quad x \in \text{ib}
\]
\[
g_2(x) = \bot \quad \text{iff} \quad x \in \text{ib} \land x \notin \text{ib}
\]
Both $g_1$ and $g_2$ are continuous and additive functions. (For $g_2$ because ib is finite and irreducible.) So $g_1$ and $g_2$ are irreducibly generated because $L_1 \times \cdots \times L_k$ is an object of $\text{ACL}_I$. To show that $g_1\. \text{unit} = f$ it suffices by continuity to show that $g_1(\text{unit}(b_1, \ldots, b_k)) = f(b_1, \ldots, b_k)$ holds when $b_j \in L_j$. For this note that
\[
\text{unit}(b_1, \ldots, b_k) \in \text{ib}
\]
holds iff
\[
\forall (ib_1, \ldots) \in \text{IB}_I \times \cdots: (ib_1, \ldots) \in (b_1, \ldots) \Rightarrow \text{unit}(ib_1, \ldots) \in \text{ib}
\]
because unit is separately additive and $L_i$ are objects of $\text{ACL}_I$ (and in the case of $g_2$ because of the way ib was chosen). By the weak product property $g_1 = f^\vartheta = g_2$ but this contradicts $g_1 \neq g_2$.

For 2 let $ib_j \in \text{IB}_{L_j}$ and define $L = \{1\}_L$ and $f: L_1 \times \cdots \times L_k \to L$ by
\[
f(l_1, \ldots, l_k) = \top \quad \text{iff} \quad \forall j: l_j \notin \text{ib}_j.
\]
This defines a continuous and separately irreducibly generated function. Further
\[
\tau = f(ib_1, \ldots, ib_k) = \text{(by } f^\vartheta \text{ irreducibly generated and } f^\vartheta \\text{unit} = f\text{)}
\]
\[
\bigcup \{f^\vartheta(ib) \mid ib \in \text{unit}(ib_1, \ldots, ib_k) \land ib \in \text{IB}\} = \text{(by } \vartheta \text{ above and } f^\vartheta \\text{unit} = f\text{)}
\]
\[
\bigcup \{f(ib_1', \ldots, ib_k') \mid \text{unit}(ib_1', \ldots, ib_k') \in \text{unit}(ib_1, \ldots, ib_k) \land (\forall j: ib_j' \in \text{IB}_{L_j}) \land \text{unit}(ib_1', \ldots, ib_k') \in \text{IB}\}
\]
Let $ib_j' \in \text{IB}_{L_j}$ be chosen such that $\text{unit}(ib_1', \ldots, ib_k') \in \text{IB}$ and $f(ib_1', \ldots, ib_k') = \tau$ and $\text{unit}(ib_1', \ldots, ib_k') \in \text{unit}(ib_1, \ldots, ib_k)$. Then
(ib_1', \ldots, ib_k') \equiv (ib_1, \ldots, ib_k) \) so by monotonicity of unit it follows that

\[
\text{unit}(ib_1, \ldots, ib_k) = \text{unit}(ib_1', \ldots, ib_k') \in IB
\]

The second result is an immediate consequence of the first. ///

The study of the tensor product has not been in vain, however.

\textbf{Lemma 4.5: 2.} The tensor product is a weak product (with unit = \text{cross} and \( f^\otimes = f^\times \)). ///

\textbf{Proof} It follows from lemma 3.2:19 that \( L_1 \otimes \ldots \otimes L_k \) is an object of ACLI with \( IB_{L_1} \otimes \ldots \otimes IB_{L_k} \) being the set of elements \( \text{cross}(ib_1, \ldots, ib_k) \) for \( ib_j \in IB_{L_j} \). It is then straightforward that \( \text{cross} \) is a separately additive morphism of ACLI. Next let \( f: L_1 \times \ldots \times L_k \to L \) be separately irreducibly generated. Uniqueness of \( f^\otimes \) is by

\[
f^\otimes(x) =
(f^\otimes \text{irreducibly generated}, f^\otimes \text{cross} = f \text{ and information about IB})
\]

\[
\bigcup \{ f(ib_1, \ldots, ib_k) \mid \forall j: ib_j \in IB_{L_j} \wedge \text{cross}(ib_1, \ldots, ib_k) \in x \} =
\]

(\( \xi \) obvious and \( f \) by \( f \) separately irreducibly generated)

\[
\bigcup \{ f(b_1, \ldots, b_k) \mid \forall j: b_j \in IB_{L_j} \wedge \text{cross}(b_1, \ldots, b_k) \in x \} = f^\times(x)
\]

For existence define \( f^\otimes \) to be \( f^\times \). Clearly \( f^\times \) is a continuous function such that \( f^\times \text{cross} = f \). It is immediate by the above calculation that \( f^\times \) is irreducibly generated. ///

The tensor product is not the only weak product.

\textbf{Example} Recall that the elements of \( \mathbb{W}_V \) are directed ideals \( J \) of terms \( t \) (e.g. \( t = \mathcal{O}(+, +) \cup \mathcal{O}(-, -) \)). Define the term \( \text{add}_i(t) \) as

\[
tw \bigcup_{b_1 \neq b_1'} b_1, b_1' \in \mathcal{O}(b_1, b_1') \text{ in } t \mathcal{O}(b_1' b_2, b_1' b_2')
\]
Define the directed ideal \( \text{add}_2(J) \) as
\[
\{ t \mid \exists t' \in J : t \in \text{add}_1(t') \}
\]
and note that \( \text{add}_2 \) is monotonic. By /Tar55/ the set of fixed points of \( \text{add}_2 : V \rightarrow V \) is a complete lattice (when ordered by \( \leq \) as in \( V \)) which will be denoted \( V \). Then \( \text{add}_3(J) = \bigcup_{n=0}^{\infty} \text{add}_2^n(J) \) is the least fixed point of \( \text{add}_2 \) that contains \( J \). Defining \( \text{unit} = \text{add}_3 \) cross the claim is that this gives a weak product with \( f^\oplus \) being the restriction of \( f \) to \( V \). Clearly \( V \) is a proper subset of \( V \) and therefore is not (isomorphic to) the tensor product.

It is slightly complicated to verify the claim. Since the result is not of profound interest only a very brief sketch will be given. First \( IB_{V \rightarrow V} \supseteq IB_{V \rightarrow V} \) is shown and this is used to show that \( V \) is an ACLI object. This again implies that \( IB_{V \rightarrow V} = IB_{V \rightarrow V} \). Since \( \text{add}_3 \) is additive it is easy to see that unit satisfies the condition. It is straightforward to verify that the suggested \( f^\oplus \) works and that no other possibilities exist.

The example above gives a weak product that is somehow simpler than the tensor product: it has fewer elements and so allows for less precision in which combinations of arguments that can be distinguished. In a certain sense this holds in general.

**Lemma 4.5.3.** For each weak product \( L_1 \cdots L_k \) there exist functions \( \text{abs} : L_1 \rightarrow L_1 \cdots L_k \) and \( \text{con} : L_1 \cdots L_k \rightarrow L_1 \cdots L_k \) such that \( (\text{abs}, \text{con}) \) is a pair of exactly adjoined functions (relative to ACLIs).
Proof Let unit and $f^\gamma$ refer to $L_1 \ldots \otimes L_k$ and cross and $f^\chi$ to $L_1 \otimes \ldots \otimes L_k$. Recall that in ACLI every additive function is irreducibly generated. Consider the diagram

\[
\begin{array}{ccc}
L_1 \times \ldots \times L_k & \overset{\text{unit}}{\longrightarrow} & L_1 \otimes \ldots \otimes L_k \\
\downarrow & & \downarrow \text{cross} \\
L_1 \otimes \ldots \otimes L_k & \overset{\text{cross}}{\longrightarrow} & L_1 \otimes \ldots \otimes L_k \\
\end{array}
\]

We shall verify the conditions for $\text{abs} = \text{cross}$ and $\text{con} = \text{unit}^\chi$.

It is immediate that $\text{id} \cdot \text{unit} = \text{unit}^\chi \cdot \text{cross}^\theta \cdot \text{unit}$. Further $\text{unit}^\chi$ is completely additive: it is additive by the tensor product property and is strict because unit is. Clearly $\text{cross}^\theta$ is irreducibly generated and it follows that so is $\text{unit}^\chi \cdot \text{cross}^\theta$. It follows that both $\text{id}$ and $\text{unit}^\chi \cdot \text{cross}^\theta$ are candidates for $\text{unit}^\theta$ and since $L_1 \otimes \ldots \otimes L_k$ is a weak product they are equal.

Next we show that $\text{id} \cdot \text{cross} = \text{lin}(\text{cross}^\theta \cdot \text{unit}^\chi) \cdot \text{cross}$. For this calculate

\[
\begin{align*}
\text{id} \cdot \text{cross} &= \\
\text{cross}^\theta \cdot \text{unit}^\chi \cdot \text{cross}^\theta &= \\
\text{lin}(\text{cross}^\theta \cdot \text{unit}^\chi) \cdot \text{cross} &= \\
\text{by lemma 4.5:1 for } L_1 \otimes \ldots \otimes L_k \\
\lambda(l_1, \ldots, l_k), \bigcup \{ \text{cross}^\theta \cdot \text{unit}^\chi \cdot \text{cross}(ib_1, \ldots, ib_k) \} \\
\forall j: \text{ib}_j \in \text{IB}_{L_j} \land \text{cross}(ib_1, \ldots, ib_k) \in \text{cross}(l_1, \ldots, l_k) \\
\text{by lemma 3.2:7 and } \text{cross}^\theta \cdot \text{unit}^\chi \cdot \text{cross} = \text{cross} \\
\lambda(l_1, \ldots, l_k), \bigcup \{ \text{cross}(ib_1, \ldots, ib_k) \} \\
\forall j: (\text{ib}_j \in \text{IB}_{L_j} \land \text{ib}_j \in \text{IB}_{L_j}) \\
\text{by cross separately additive hence separately irreducibly generated} \\
\text{id} \cdot \text{cross}
\end{align*}
\]
As in the previous paragraph it follows that \( \text{id} = \text{lin}(\text{cross}^{\mathbf{unit}^\times}) \)
and then \( \text{id} \in \text{cross}^{\mathbf{unit}^\times} \) is immediate.

We shall not consider the corresponding development for \( \emptyset \). One technical problem in doing so is that \( L_1 \times \ldots \times L_k \) needs not be an \( \text{ACLI} \) object even if all \( L_i \) are. (In fact \( L_2 \times L_2 \) in the proof of lemma 4.4:3 is an example.) To overcome these problems one might adopt the suggestion in the final remark in section 4.4 (but this would make \( \emptyset \) uninteresting).
The development of abstract interpretation in the previous chapters was based on the relational powerdomain and approximations to it. This setting is useful for many purposes but there seems to be situations where the relational powerdomain excludes subsets that are of interest. This motivates a development based on a powerdomain that allows more subsets to be considered.

The motivation for why more subsets seem to be needed arises from program transformations whose safeness depend on whether or not the execution of a certain piece of program terminates. As an example consider an imperative programming language and the statements

(1) \texttt{let } p\texttt{(value } x\texttt{) be } b_1 \texttt{ in } b_2

and

(2) \texttt{let } p\texttt{(name } x\texttt{) be } b_1 \texttt{ in } b_2

Both statements declare a procedure \( p \) with one argument \( x \). The parameter mechanism is call-by-value in (1) and call-by-name in (2). The procedure \( p \) may be called in \( b_2 \) and is defined by \( b_1 \). We shall further assume that \( x \) does not occur in \( b_1 \) (or at least that there is a way of executing \( b_1 \) such that \( x \) is not referenced). A similar set-up may be imagined for an applicative programming language where \( b_1 \) and \( b_2 \) will be expressions.
One program transformation might be to replace (2) by (1). For this to be safe every argument e to a call of p in b₂ must terminate upon execution. For otherwise (2) may terminate in a situation where (1) does not and therefore the program transformation does not preserve the overall meaning of the program. This transformation was considered in /Myc80, Myc81/ in the context of optimising applicative programs. Another program transformation is to replace (1) by (2). This is essentially what happens in an applicative language when expanding a call in-line. To see this note that (2) is equivalent to

\[(2') \ b₂[ b₁[e/x] / p(e) \]

that is b₂ with all occurrences of a call p(e) replaced by the body b₁ except that x is replaced by e. In an imperative programming language in-line expansion resembles replacing (2) by (1). For (1) is essentially equivalent to

\[(1') \ b₂[ x:=e; b₁ / p(e) \]

We shall not go further into such transformations, but the examples show that a data flow analysis for detecting safe approximations to "will terminate" may be useful. Therefore it ought to be possible to prove it correct within the framework of abstract interpretation.

It is natural to take the point of view that possible non-termination of an expression is just another facet of which value it may evaluate to. This is in line with domain theory where the value ⊥ is returned by a function iff it does not terminate. This means that the denotation \( [e] \) of e in the collecting semantics should not be forced to contain ⊥ as is the case with the relational powerdomain. In section 3.1 it was argued that the Smyth powerdomain is not suitable (only one infinite set) and this also holds for the Plotkin
powerdomain (all infinite sets are forced to contain \( \bot \)). Therefore a powerdomain with fewer restrictions is studied in section 5.2.

Unfortunately continuity properties may fail and we rely on monotonicity instead. The relevant domain theory is covered in section 5.1. (Throughout this chapter recursive domains will be excluded from consideration.)

Section 5.3 gives the standard and collecting semantics for the applicative language of section 2.4. They are shown to be related much as in section 3.3 but the collecting semantics is a more faithful representation of the standard semantics than before (because the powerdomain allows more subsets to be considered). In section 5.4 it is argued that abstract interpretation should use spaces with two partial orders. For the powerdomain they are the Egli-Milner order (used for least fixed points) and subset inclusion (used for safe approximation). These partial orders are the same for the relational powerdomain so this is a more general outlook than in the previous chapters. When the abstraction and concretization functions preserve the relations in a suitable way a development of abstract interpretation can be given. The major application in section 5.5 is to validate two data flow analyses taken from /Myc81/ and there used to detect when call-by-name can be replaced by call-by-value. One of the analyses seems to require this stronger approach.

Section 5.6 defines a small nondeterministic programming language and its nondeterministic semantics. Also a deterministic semantics is given where oracles (strings of 0's and 1's) are used to resolve which choice is to be made. Abstract interpretation is then extended so as to allow some arguments not to be "collected"; for the collecting semantics this means we may have functionalities \( N_\bot \times \Phi(N_\bot) \to \Phi(N_\bot) \).
When applying this to the deterministic semantics it is shown that the nondeterministic semantics results. This gives a clear perspective upon what nondeterminism "is". It is a natural data flow analysis question to investigate what happens if not all oracles are possible and this is done in section 5.7. Under reasonable assumptions it is shown that this gives a semantics \( D_0 \) that specifies convergence more often than the nondeterministic semantics. However, if the nondeterministic semantics is modified to use another fixed point operator it is possible to obtain a semantics that specifies convergence more often than \( D_0 \). In this way \( D_0 \) is bounded by the two nondeterministic semantics.

The main purpose of this chapter is to explore a theory of abstract interpretation where aspects of termination may be dealt with. Much work remains in order to attain the same degree of "universality" that holds for the development of the previous chapters. Even then there seems to be applications where the "more general outlook" of this chapter is too narrow as will be discussed in chapter 6.

5.1 NON-CONTINUOUS DOMAIN THEORY

In this section we overview a theory of domains where assumptions about continuity are replaced by assumptions about monotonicity. Most of the concepts are quite standard and can be found in /ApP182/ or /Mar76/.

Let \( D = (D, \leq) \) be a partially ordered set. A subset \( Y \) of \( D \) is \underline{directed} iff for every finite subset \( Y' \) of \( Y \) there is an element \( y \) of \( Y \) that is an upper bound for \( Y' \), i.e. \( \forall y' \in Y': y' \leq y \). A directed set cannot be empty because \( Y' \) can be chosen as the empty set. The
partially ordered set is directed complete (is a dcpo) iff it has a least element \( \bot \) and for every directed subset \( Y \) of \( D \) there is a least upper bound \( \bigvee Y \) in \( D \). It follows from section 2.2 that a dcpo is a cpo and an algebraic cpo is a dcpo. The definition of dcpo will be further clarified shortly (equivalent definitions and different concepts). For the present it suffices to state that not all cpo's are dcpo's.

... Only a few constructions upon dcpo's will be needed. For a set \( S \) the dcpo \( S_\bot \) has as elements \( S \cup \{ \bot \} \) (assuming \( \bot \) is not an element of \( S \)) and is partially ordered by \( x \leq y \) iff \( x = \bot \) or \( x = y \). If \( k > 1 \) and \( D_i \) are dcpo's then so is the cartesian product \( D_1 \times \cdots \times D_k \). It is partially ordered componentwise and the least upper bound is also componentwise, i.e.

\[
\bigvee Y = \{ \bigcup \{ y \mid y \in Y \} \}.
\]

Also the smash product

\[
D_1 \times \cdots \times D_k = \{ \text{smash}(d_1, \ldots, d_k) \mid \forall i: d_i \in D_i \}
\]

is a dcpo with least upper bounds determined componentwise. For dcpo's \( D \) and \( E \) the function space \( D \rightarrow E \) is redefined to consist of all monotonic functions from \( D \) to \( E \). It is partially ordered pointwise and gives a dcpo where least upper bounds are pointwise, i.e.

\[
(\bigvee Y)(d) = \bigcup \{ f(d) \mid f \in Y \}.
\]

Every monotonic function over a dcpo has a least fixed point. The construction and proof uses ordinal numbers and transfinite induction (see /Hal60/). Let \( f: D \rightarrow D \) be a monotonic function and \( D \) a dcpo. Inductively define, for each ordinal \( \lambda \), the element \( f(\lambda) \) of \( D \) by
\[ f(\lambda) = f( \bigcup \{ f(K) \mid K < \lambda \} ) \]

This obviously makes sense when \( \lambda \) is a natural number and gives \( f(n) = f^{n+1}(\lambda) \). To see it makes sense in general it suffices to show that \( \lambda_1 < \lambda_2 < \lambda \) implies \( f(\lambda_1) \in f(\lambda_2) \) as then \( \{ f(K) \mid K < \lambda \} \) is directed. But this follows from the monotonicity of \( f \) and

\[ \{ f(K) \mid K < \lambda_1 \} \subseteq \{ f(K) \mid K < \lambda_2 \} \]

Next define

\[ \text{LFP}(f) = f(\lambda_D) \]

for \( \lambda_D \) the (ordinal number corresponding to the) cardinality of the powerset of \( D \).

**Lemma 5.1:** \( \text{LFP}(f) \) is the least fixed point of \( f \).

**Proof** We first show that \( \text{LFP}(f) \) is less than or equal to any fixed point. So let \( f(d) \leq d \) and show by transfinite induction on \( \lambda \leq \lambda_D \) that \( f(\lambda) \leq d \). If the result holds for all ordinals \( k < \lambda \) then

\[ \bigcup \{ f(K) \mid K < \lambda \} \leq d \]

and monotonicity of \( f \) and the assumption about \( d \) gives the result for \( \lambda \) too.

We next show that \( \text{LFP}(f) \) is a fixed point. It cannot be that all \( f(\lambda) \) for \( \lambda < \lambda_D \) are distinct as this would contradict Cantor's theorem \cite{hal60}. So let \( \lambda_f \) be the least ordinal \( \lambda \) such that \( f(\lambda) = f(\lambda_f) \) for some ordinal \( \lambda_f \) that is not \( \lambda \). This means that there is an ordinal \( \kappa_f > \lambda_f \) such that \( f(\lambda_f) = f(\kappa_f) \). We have \( \lambda_f < \lambda_f + 1 < \kappa_f < \lambda_D \) and because \( f(\lambda) \) depends monotonically on \( \lambda \), and because \( f(\lambda+1) = f(f(\lambda)) \), this gives

\[ f(\lambda_f) \in f(f(\lambda_f)) \leq f(\kappa_f) \in \text{LFP}(f) \]
But by choice of $K$ this shows that $f^{(A_K)}$ is a fixed point of $f$ that is less than or equal to $\text{LFP}(f)$. By the first half of the proof it must equal $\text{LFP}(f)$. 

Corresponding to the definition of least fixed points there is the following rule of fixed point induction. A predicate $Q$ on $D$ is (directed) admissible iff $Q(\lambda)$ holds and $Q(\bigcup Y)$ holds for every directed set $Y$ such that $\forall y \in Y: Q(y)$. The fixed point induction principle says

$$\forall d \in D: Q(d) \supset Q(f(d))$$

for every directed admissible predicate $Q$ and monotonic function $f$ over a dcpo $D$. The proof of this principle is straightforward by transfinite induction. (For ordinal $\lambda$ one shows $Q(f^{(\lambda)})$ using that $Q(f^{(k)})$ holds when $k < \lambda$.)

If $Q$ is a predicate on $D \rightarrow E$ we write $D \rightarrow Q E$ for the subset of those monotonic functions that satisfy $Q$. This gives a partially ordered set when using the partial order of $D \rightarrow E$. The predicate $Q$ is admissible iff $D \rightarrow Q E$ is a dcpo with the same least element as in $D \rightarrow E$ and with the same least upper bounds of directed sets as in $D \rightarrow E$.

Strictness is an admissible predicate and so is directed continuity (which means that least upper bounds of directed sets are preserved as in $f(\bigcup Y) = \bigcup \{f(y) \mid y \in Y\}$).

**Clarifying the definition of dcpo**

To clarify the definition of dcpo we need the following concepts. A directed chain is a directed set that is totally ordered. Recall that an ordinal $\lambda$ is partially ordered by $\leq_{/\text{Hal60}/}$. A chain
indexed by $\lambda$ is a monotonic function from $\lambda$ (into some partially ordered set $D$). A chain indexed by $\omega$, the smallest infinite ordinal /Hal60/, then is what was called a chain in chapter 1.

A chain $(d_k)_{k<\lambda}$ indexed by $\lambda$ is cofinal in the directed set $Y$ iff each $d_k$ is an element of $Y$ and for each $y \in Y$ there is an ordinal $k<\lambda$ such that $y \subseteq d_k$. If $(d_k)_{k<\lambda}$ is cofinal in $Y$ then $Y$ and $\{d_k | k<\lambda\}$ have the same upper bounds. So $\bigcup Y$ exists iff $\bigcup \{d_k | k<\lambda\}$ exists and they are equal if one exists.

**Lemma 5.1.2.** For every directed chain $Y$ there is a cofinal chain $(d_k)_{k<\lambda}$ indexed by some cardinal $\lambda<\text{card}(Y)$.

**Proof.** We first show the weaker result where the assumption that $\lambda$ is a cardinal is replaced by the assumption that $\lambda$ is merely an ordinal. Since $\text{card}(Y)$ is a cardinal there is a bijection $i:\text{card}(Y) \rightarrow Y$. Let $g$ be a choice function /Hal60/ for the set $Y$. The definition of $d_k$ and $\lambda$ is by transfinite induction on $k<\text{card}(Y)$.

If $\{d_k | k<\lambda\}$ does not have an upper bound in $Y$ we stop the construction and set $\lambda = \lambda'$. Otherwise let $d_{\lambda'}$ be the larger of $i(\lambda')$ and some upper bound of $\{d_k | k<\lambda\}$. (The upper bound can be chosen uniquely by using the choice function $g$.) If $d_k$ are defined for all $k<\text{card}(Y)$ we set $\lambda = \text{card}(Y)$.

Clearly $(d_k)_{k<\lambda}$ is a chain indexed by $\lambda$ and each $d_k$ is an element of $Y$. To see it is cofinal fix $y \in Y$. If $i^{-1}(y)<\lambda$ it is obvious that $y \subseteq d_{i^{-1}(y)}$. If $i^{-1}(y)\geq \lambda$ we must have that $y$ is not an upper bound of $\{d_k | k<\lambda\}$. Hence there is $k<\lambda$ such that $\tau(d_k) = y$ and this implies $y \subseteq d_k$ because $Y$ is totally ordered.
To prove the result stated in the lemma let \( \lambda \) be the smallest ordinal such that there is a cofinal chain \( \{d_k\}_{k<\lambda} \) in \( Y \). Next use the weak result above on \( (Y=) \{d_k\}_{k<\lambda} \) to get a cofinal chain \( \{d'_k\}_{k<\lambda'} \) with \( \lambda' \leq \text{card}(\lambda) \). Since \( \{d'_k\}_{k<\lambda'} \) is cofinal in \( Y \) too this shows that \( \lambda' = \lambda \). Hence \( \lambda = \text{card}(\lambda) \) and therefore \( \lambda \) is a cardinal.

It follows from example 1 in /Mar76/ that the lemma may fail if \( Y \) is only a directed set.

The next theorem shows that it was not important that the definition of dcpo focused on directed sets.

**Theorem 5.1:3.** Let \( \lambda \) be a cardinal and \( D \) a partially ordered set with a least element. Consider the following assumptions about which least upper bounds should exist in \( D \):

1. of all directed sets (of cardinality at most \( \lambda \))
2. of all directed chains (of cardinality at most \( \lambda \))
3. of all chains indexed by an ordinal (at most \( \lambda \))
4. of all chains indexed by a cardinal (at most \( \lambda \))

All these statements are equivalent and this also holds if the parts in parentheses are ignored.

**Proof** Let \( (1'), \ldots, (4') \) be \( (1), \ldots, (4) \) with the parts in parentheses ignored. It is obvious that \( (1) \Rightarrow (2) \Rightarrow (3) \Rightarrow (4) \) and that \( (1') \Rightarrow (2') \Rightarrow (3') \Rightarrow (4') \). That \( (2) \Rightarrow (1) \) and \( (2') \Rightarrow (1') \) is by corollaries 1 and 2 in /Mar76/. (The proof essentially proceeds by transfinite induction on the cardinality of the directed set. For each directed set the assumption \( (2) \) or \( (2') \) is used to construct a directed chain with the same least upper bound.) That \( (4) \Rightarrow (2) \) is
by the previous lemma and (4') \(\Rightarrow\) (2') is similar. 

What was important in the definition of dcpo is that there was no cardinality restrictions upon the directed sets considered. Let a \(\lambda\)-cpo mean a partially ordered set with a least element and with least upper bounds of directed sets of cardinality at most \(\lambda\). Then \(D\) is a dcpo iff it is a \(\text{card}(D)\)-cpo and cpo in the sense of section 2.2 means \(\omega\)-cpo.

**Fact 5.1: 4.** If \(\lambda_1 < \lambda_2\) then every \(\lambda_2\)-cpo is a \(\lambda_1\)-cpo. 

**Fact 5.1: 5.** If \(\lambda_1 < \lambda_2\) are infinite cardinals there is a \(\lambda_1\)-cpo that is not a \(\lambda_2\)-cpo.

**Proof** Let \(\alpha_3\) be the smallest cardinal strictly greater than \(\lambda_1\). Since it is a limit ordinal it has no greatest element and therefore the directed subset \(\alpha_3\) has no least upper bound. This shows that \(\alpha_3\) is not a \(\lambda_2\)-cpo. To see it is a \(\lambda_1\)-cpo let \(y\) be a directed subset of cardinality at most \(\lambda_1\). The elements of \(y\) have cardinality at most \(\lambda_1\) and therefore so has \(\cup y\) (because \(\lambda_1 \cdot \lambda_1 = \lambda_1\) /Hal60 p.97/). So \(\cup y\) is an element of \(\alpha_3\) and is the least upper bound /Hal60 p.79/.

If \(n\) is a natural number then \(n\)-cpo means a partially ordered set with a least element. The partially ordered sets considered in /Plo82/ are the \(\Omega\)-cpo's, where \(\Omega\) is the smallest non-countable ordinal /Hal60/.

An advantage of not placing cardinality restrictions upon the directed sets is that it is not necessary with continuity assumptions in order to guarantee that least fixed points exist. Another consequence is:

**Fact 5.1: 6.** A dcpo is a complete lattice iff it is a semi-lattice.
This is corollary 5 in /Mar76/. The idea in "if" is to consider a nonempty set $Y$. Let $i: \text{card}(Y) \rightarrow Y$ be a bijection and define $d_A$ by $i(\lambda) \cup \{d_k | k < \lambda\}$. Then $\bigcup Y$ exists and equals $\bigcup \{d_k | k < \text{card}(Y)\}$.

The definition of directed continuous can be analysed in much the same way. For a monotonic function between dcpo's one can view (1) to (4) above as assumptions about which least upper bounds it should preserve. One can show that an analogue of theorem 5.1:3 holds. (For (2) $\Rightarrow$ (1) use corollary 3 of /Mar76/.) Finally a monotonic function is completely additive iff it is strict, additive and directed continuous.

### 5.2 POWERDOMAINS

As has been argued in the introduction to this chapter the Plotkin powerdomain /Plo76/ does not allow (to distinguish between) all the sets that are desired. In this section a preliminary theory is given for a powerdomain that is better in this respect.

The powerdomain of a dcpo $D$ has as elements certain subsets of $D$. A subset $Y$ of $D$ is convex iff whenever $d_1 \leq d_2 \leq d_3$ with $d_1$ and $d_3$ elements of $Y$ then so is $d_2$. The Egli-Milner order $\subseteq_{EM}$ is defined between subsets by

$$y_1 \subseteq_{EM} y_2 \iff \forall y_2 \in y_1 : \exists y_2 \in y_2 : y_1 \subseteq y_2$$

and

$$\forall y_2 \in y_2 : \exists y_1 \in y_1 : y_1 \subseteq y_2$$

**Definition** The powerdomain $\mathcal{P}(D)$ of a dcpo $D$ is defined as

$$\{ \{Y \subseteq D | Y \text{ is convex and nonempty} \}, \subseteq_{EM} \}.$$  

It is straightforward to verify that this gives a partially ordered set with $\{1\}$ as least element. It is not always a dcpo and we
now investigate when it is.

**As a dcpo**

It is convenient to define, for every subset $Y$ of $D$, the sets

- $\text{LC}(Y) = \{ d \in D \mid \exists y \in Y: d \leq y \}$ (left closure)
- $\text{RC}(Y) = \{ d \in D \mid \exists y \in Y: d \geq y \}$ (right closure)
- $\text{CON}(Y) = \text{LC}(Y) \cap \text{RC}(Y)$ (convex closure)
- $\text{MAX}(Y) = \{ y \in Y \mid \forall y' \in Y: y' \neq y \}$ (maximal elements)

One may then reformulate $Y_1 \leq_{\text{EM}} Y_2$ as

$$Y_1 \subseteq \text{LC}(Y_2) \text{ and } \text{RC}(Y_1) \supseteq Y_2$$

and a set $Y$ is convex iff $Y = \text{CON}(Y)$. A dcpo is of finite height (is a fdcpo) iff every directed subset contains its own least upper bound.

**Fact 5.2:** 1. If $D$ is of finite height then $Y \subseteq \text{LC}(\text{MAX}(Y))$, i.e. for all $y \in Y$ there exists $y_N \in \text{MAX}(Y)$ such that $y \leq y_N$. //

**Proof** For the sake of contradiction suppose that the fact is violated when $y = y_0$. Construct $y_{n+1}$ as some element of $Y$ that is strictly greater than $y_n$. Then $\{y_n \mid n \geq 0\}$ is a directed set that does not contain its least upper bound. This is the desired contradiction with the finite height of $D$. ///

Two elements $d_1$ and $d_2$ are **incomparable** iff neither $d_1 \leq d_2$ nor $d_1 \geq d_2$. The elements of a set $Y$ are pairwise incomparable iff every element of $Y$ is incomparable with every other element of $Y$. This is the case iff $Y = \text{MAX}(Y)$. //
The following definition is central for investigating when $\mathcal{D}(D)$ is a dcpo.

**Definition** A dcpo $D$ is benign iff for every subset $Y_L$ and $Y_R$ of $D$ and element $z$ of $D$ such that

1. $Y_L$ is infinite
2. $Y_L = \text{MAX}(Y_L)$, i.e. the elements of $Y_L$ are pairwise incomparable.
3. $Y_L \subseteq \text{LC} \{z\}$, i.e. each element of $Y_L$ is "below" $z$
4. $Y_L \subseteq \text{LC}(Y_R)$
5. each element of $Y_R$ is incomparable with $z$

it is also the case that

6. $\exists y \in \text{LC}(Y_R) \cap \text{LC}(\{z\}) \cap (\text{RC}(Y_L) - Y_L)$

(where $-$ denotes set difference).

Conditions (1) to (5) are illustrated by

![Diagram](https://via.placeholder.com/150)

It is hard to give an "intuitive" motivation for this concept but we have:

**Theorem 5.2:2** Let $D$ be a dcpo of finite height.

(i) $\mathcal{D}(D)$ is a dcpo iff $D$ is benign

(ii) $U_Y = \text{LC}(U_Y) \cap \bigcap_{y \in Y} \text{RC}(Y)$ when $D$ is benign
(iii) the class of fdcpo's such that

no infinite set of pairwise incomparable elements has

an upper bound

is the largest class of benign fdcpo's that is closed
under cartesian product.

Proof of "only if" in (i): Suppose by way of contradiction that
\( \mathcal{D}(D) \) is a dcpo but that \( D \) is a fdcpo that is not benign. Let \( Y_L \),
\( Y_R \) and \( z \) fulfill (1) to (5) and violate (6). We shall construct a
directed chain in \( \mathcal{E}(D) \) and obtain the desired contradiction.

The cardinal number \( \beta = \text{card}(Y_L) \) is infinite by (1). Let \( i: \beta \to Y_L \)
be a bijection and define for \( \lambda < \beta \) the set

\[ Y_\lambda = \text{CON}(Y_L \cup \{ i(\kappa) \mid \lambda \leq \kappa < \beta \}) \]

To see that this gives a chain indexed by \( \beta \) let \( \beta < \lambda < \beta \). Since
\( Y_\alpha \supseteq Y_\lambda \) we get \( \text{RC}(Y_\lambda) \supseteq Y_\lambda \). Also \( \text{MAX}(Y_\lambda) = \text{MAX}(Y_R) = \text{MAX}(Y_\lambda) \) and
the previous fact shows that \( Y_\lambda \subseteq \text{LC}(\text{MAX}(Y_\lambda)) \subseteq \text{LC}(Y_\lambda) \). It follows
that \( Y_\lambda \in \mathcal{E}(D) \).

Both \( M_1 = \text{MAX}(Y_R) \) and \( M_2 = \text{MAX}(Y_R \cup \{ z \}) \) are convex, nonempty sets
and we now show that they are upper bounds of \( (Y_\lambda)_{\lambda < \beta} \). That :
\( Y_\lambda \subseteq \text{LC}(M_1) \) is immediate for both \( i = 1 \) and \( i = 2 \). Next that \( \text{RC}(Y_\lambda) \supseteq M_1 \)
is immediate and that \( \text{RC}(Y_\lambda) \supseteq M_2 \) is because each \( Y_\lambda \) contains an
element of \( Y_L \) and \( z \) will be greater than or equal to that element.

Because \( \mathcal{E}(D) \) is a dcpo there is a least upper bound \( M \) of \( (Y_\lambda)_{\lambda < \beta} \)
and we have \( M \in \mathcal{E}(M_1) \) and \( M \in \mathcal{E}(M_2) \). This gives \( m \in M \) such that \( m \in z \)
and \( y \in Y_R \) such that \( m \in y \). It follows that \( m \in \text{LC}(Y_R) \cap \text{LC}(\{ z \}) \).
Also for each \( \lambda \) there is \( y_\lambda \in Y_L \) such that \( y_\lambda \in M \). This shows that
\( m \in \text{RC}(Y_L) \) and because (6) is violated we must have \( m \in Y_L \). But for
Proof of "if" in (i) and of (ii): Suppose that $D$ is a benign fdcpo with $\mathcal{Y}$ as a directed subset. Let $\mathcal{U}B$ be defined by the formula in (ii).

We first show that $\mathcal{U}B$ is a convex, nonempty set that is an upper bound of $\mathcal{Y}$. To see it is convex let $d_1 \leq d_2 \leq d_3$ with $d_1 \in \mathcal{U}B$ and $d_3 \in \mathcal{U}B$. Because $d_3 \in \mathcal{L}C(\mathcal{U}Y)$ we get $d_2 \in \mathcal{L}C(\mathcal{U}Y)$ and because $d_1 \in \cap_{y \in \mathcal{Y}} \mathcal{R}C(Y)$ we get $d_2 \in \cap_{y \in \mathcal{Y}} \mathcal{R}C(Y)$. This shows that $d_2 \in \mathcal{U}B$. To see that it is not empty we note that $\text{MAX}(\mathcal{U}Y)$ is not empty and show that $\text{MAX}(\mathcal{U}Y)$ is a subset of $\mathcal{U}B$. So let $y \in \text{MAX}(\mathcal{U}Y)$ and note that this gives $y \in \mathcal{Y}$ such that $y \in \mathcal{Y}$. To see that $y \in \mathcal{U}B$ it suffices to consider $y' \in \mathcal{Y}$ and show $y' \in \mathcal{R}C(Y')$. Let $Y'$ be an upper bound in $\mathcal{Y}$ of $\mathcal{Y}$ and $\mathcal{Y}'$. By the assumption about $\mathcal{Y}$ we have $Y' \subseteq Y$ and $\mathcal{Y}' \subseteq Y'$. This shows that $y \subseteq Y'$ and $y \in \mathcal{R}C(Y)$. Finally, to see that it is an upper bound consider $y \in \mathcal{Y}$ and show $y \in \mathcal{U}B$. For $y \in \mathcal{Y}$ we use the previous fact to find $y' \in \text{MAX}(\mathcal{U}Y)$ such that $y' \in \mathcal{Y}$ and by the above $y' \in \mathcal{U}B$. For $y' \in \mathcal{U}B$ we have $y' \in \mathcal{R}C(Y)$ and therefore there is $y \in \mathcal{Y}$ such that $y \in \mathcal{Y}'$.

Next let $M$ be another upper bound of $\mathcal{Y}$ and show $\mathcal{U}B \subseteq M$. Note that $\mathcal{U}B$ is a subset of $\mathcal{U}Y$. For $y \in \mathcal{U}B$ we therefore have $y \in \mathcal{Y}$ such that $y \in \mathcal{U}B$. For the sake of contradiction assume that no such $y$ exists. Then also there is no $y \in \mathcal{U}B$ such that $z \in \mathcal{Y}$ for otherwise $z$ would be an element of $\mathcal{U}B$ thereby contradicting the assumption.
Define \( Y_R = \text{MAX}(\text{UB}) \) and \( Y_{VL} = \text{MAX}( (U_y)_\cap \text{LC}(\{z\}) ) \) and note that both sets are convex and not empty (for \( Y_{VL} \) use that \( M \) is an upper bound of \( Y \)). Define \( Y_L = \text{MAX}( \text{LC}(Y_R) \cap \text{LC}(\{z\}) \cap \text{RC}(Y_{VL}) \) and note that this gives a convex and nonempty set (the latter is because \( Y_{VL} \) is a subset of the set of which \( \text{MAX} \) is taken).

Suppose that \( Y_L \) is infinite. Clearly conditions (1) to (5) of benign hold. This gives by (6) a certain element \( y \). Since \( y \in \text{RC}(Y_L) - Y_L \) we get \( y \in \text{RC}(Y_{VL}) \) and because \( y \in \text{LC}(\{z\}) \cap \text{LC}(Y_R) \) this gives \( y \in \text{LC}(Y_L) \). But this gives a contradiction with \( y \in \text{RC}(Y_L) - Y_L \). Next suppose that \( Y_L \) is finite and that \( y_1, \ldots, y_n \) are all the elements. Each \( y_i \) is in \( \text{LC}(Y_R) \) and hence in \( \text{LC}(U_y) \). Since \( y_i \notin \text{L} \) the assumption about "no such \( y \)" gives \( y_i \in Y \) such that \( y_i \in \text{RC}(Y_i) \). Let \( Y \) be an upper bound in \( Y \) of \( y_1, \ldots, y_n \). Then \( y_i \in \text{RC}(Y) \) must be the case for all \( i \). But \( y \in Y \) gives \( y_i \in Y \) such that \( y \in Y \). Therefore there is \( y'' \in Y_{VL} \) such that \( y'' \in Y \) and \( y_i \in Y_L \) such that \( y_i \in Y_L \). This shows that \( y_i \in \text{RC}(Y) \) and is the desired contradiction.

Proof of (iii): We first show that the class described is a subclass of the benign fdcpo's and is closed under cartesian product. Clearly each fdcpo in the class is benign. If \( D_1, \ldots, D_k (k > 1) \) are fdcpo's in the class then \( D_1 \times \ldots \times D_k \) is a fdcpo. To see it is in the class it suffices to consider \( k=2 \) as \( D_1 \times \ldots \times D_k \) is isomorphic to \( (D_1 \times D_2) \times \ldots \times D_k \).

Suppose by way of contradiction that \( Y \) is an infinite subset of \( D_1 \times D_2 \) that has an upper bound \( (u_1, u_2) \) and whose elements are pairwise incomparable. Both \( \{y \mid y \in Y\} \) and \( \{y \mid y \in Y\} \) have an upper bound and at least one is infinite. Suppose (without loss of generality) it is \( \{y \mid y \in Y\} \) and call this set \( L_0 \). We construct a strictly
decreasing sequence \((d_n)_{n \geq 1}\) of elements of \(L_0\) and a strictly decreasing sequence \((L_n)_n\) of infinite subsets of \(L_0\). Since \(L_{n-1}\) is infinite and \(\text{MAX}(L_{n-1})\) is finite there is an element of \(\text{MAX}(L_{n-1})\) such that infinitely many elements of \(L_{n-1}\) is less than it. Let \(d_n\) be such an element of \(\text{MAX}(L_{n-1})\) and put \(L_n = \{d \in L_{n-1} \mid d \leq d_n \land d \neq d_m\}\). Clearly \(n \neq m\) implies that \(d_n \notin d_m\) and \(d_n \neq d_m\).

It is immediate that there exists \(e_n\) such that \((d_n, e_n) \in Y\) for all \(n\). The sequence \((e_n)_n\) must be non-decreasing, i.e. \(m > n\) implies \(e_m \leq e_n\), because the elements of \(Y\) are pairwise incomparable. Therefore, if \(m > n\) then \(e_m \neq e_n\) and if \(e_m\) and \(e_n\) are comparable we must have \(e_m \leq e_n\). Next fix \(n_0\). It cannot be that for each \(n > n_0\) there is \(m > n\) such that \(e_m\) and \(e_n\) are comparable. For if this is the case we can construct an infinite strictly increasing sequence and this contradicts the finite height of \(D_2\). So define \(f(n_0)\) as the least \(n_1 > n_0\) such that \(m > n_1\) implies that \(e_m\) and \(e_{n_1}\) are incomparable. Then \(\{e_j \mid j = f(i)(0)\}\) is an infinite set of pairwise incomparable elements. Since it has \(u_2\) as an upper bound this contradicts the assumption about \(D_2\) being in the class.

To see that the described class is largest suppose by way of contradiction that it is not. So there is another class containing a benign fdcpo \(D\) that is not in the described class. Hence \(D \times D\) is a benign fdcpo and there is an infinite subset \(\{d_n \mid n \geq 1\}\) of pairwise incomparable elements of \(D\) with an upper bound \(u\). (It is no restriction to assume that the set is countable.) Because \(D\) is of finite height there exists a maximal lower bound \(l\) for \(d_1\) and \(d_2\). In \(D \times D\) define \(Y_L = \{(d_1, 1) \mid n \geq 1\}\), \(Y_R = \{(d_n, d_1) \mid n \geq 1\}\) and \(z = (u, d_2)\). Conditions (1) to (5) of benign hold so by (6) there is an element \(y\) and numbers \(n\) and \(m\) such that \(y \notin (u, d_2)\) and \(y \notin (d_n, d_1)\) and \(y \notin (d_m, 1)\).
and \( y \neq (d, m, l) \). The first three statements show that \( y = (d, m, l) \) and this contradicts the fourth. Hence the described class is largest.

///

**AS AN AUGMENTED DCPPO**

So far the powerdomain has been studied with respect to the Egli-Milner order \( E_{EM} \). Another partial order that may be studied is subset inclusion \( \subseteq \). We shall see in section 5.4 that both are of importance for abstract interpretation. (The Egli-Milner order will be used to define least fixed points and subset inclusion to express safe approximation.) Analogously to /Egl75/ and /HeAs80/ this motivates a study of structures with two partial orders.

An augmented dcpo \( B = (B, \mathcal{E}, \mathcal{S}) \) is a dcpo \( (B, \mathcal{S}) \) such that \( (B, \mathcal{E}) \) is a partially ordered set. The augmentation is admissible iff \( \mathcal{E} \) is an admissible predicate upon \( B \times B \) (with respect to \( \mathcal{S} \) defined componentwise). The augmentation is complete iff \( (B, \mathcal{S}) \) has least upper bounds of all nonempty subsets and the least upper bound operator \( \lor \) satisfies the following monotonicity property:

\[
\text{if } y, y' \text{ are nonempty subsets of } B \text{ and } y \subseteq y' \text{ then } \lor y \subseteq \lor y'.
\]

(Here \( \subseteq_{EM} \) is defined with respect to \( \subseteq \), i.e. \( y \subseteq_{EM} y' \) iff for all \( y' \subseteq y' \) there is \( y \subseteq y' \) such that \( y \subseteq y' \) and \( \ldots \) ) The following fact is used in the proof of the theorem below and gives a connection between an admissible augmentation and what is sometimes called a continuous semi-lattice /PloLN/.

**Fact 5.2:** If \( B = (B, \mathcal{E}, \mathcal{S}) \) is a completely augmented dcpo and

\[
\forall = \lambda(y_1, y_2). \lor \{y_1, y_2\}\text{ is directed continuous then the augmentation}
\]
Proof Clearly \(Y\) holds and let \(Y\) be a directed subset of \(B\) such that \(Y_1 \leq Y_2\) for every \((Y_1, Y_2) \in Y\). Then \(w(Y_1, Y_2) = Y_2\) holds for \((Y_1, Y_2) \in Y\) and the calculation

\[
(u_1 \cup u_2) (u_1 v_1 (u_1 \cup u_2) = u_1 v_1 (u_1 \cup u_2) = v(u_1 \cup u_2)
\]

shows that \((u_1 \cup u_2) \leq (u_1 \cup u_2)\) as was to be shown.

The definition of augmented dcpo pays off because the powerdomain is such a structure.

Theorem 5.2.4. If \(D\) is a benign fdcpo then \(\mathcal{E}(D) = (\mathcal{E}(D), \mathcal{E}_{EM}, \mathcal{E})\) is an admissibly and completely augmented dcpo with \(Uy = \text{CON}(Uy)\).

Proof Clearly \(\mathcal{E}\) is a partial order upon \(\mathcal{F}(D)\). If \(Y\) is a set of elements of \(\mathcal{F}(D)\) then an element \(Y\) of \(\mathcal{F}(D)\) is an upper bound iff \(Uy \leq Y\). Since \(Y\) is convex this holds iff \(\text{CON}(Uy) \leq Y\). This shows that \((\mathcal{F}(D), \mathcal{E})\) has least upper bounds of nonempty subsets and that the least upper bound is given by \(U\).

If \(Y, Y' \in \mathcal{F}(D)\) we must show \(Uy \leq \mathcal{E}_{EM} Uy'\). Since \(\text{CON}(Y), Y, Y' \in \mathcal{F}(D)\) holds for all subsets \(Y\) of \(D\) it suffices to show that \(Uy \leq \mathcal{E}_{EM} Uy'\). When \(y \in Uy\) there is \(y \in Y\) such that \(y \leq Y\). Hence there is \(y' \in Y'\) with \(y \in Y\) and this gives \(y' \in Y\) such that \(y \leq Y\). Clearly \(y'\) is an element of \(Uy\). When \(y' \in Uy\) one finds \(y \leq Uy\) such that \(y \leq Uy\) in a similar way.
To see that \( S \) is admissible we use the previous fact and show that \( \psi = \lambda(y_1, y_2). \psi[y_1, y_2] \) is directed continuous. It suffices to show that \( \psi \) is continuous in each argument and because \( \psi \) is commutative it suffices to consider the left argument. Let \( Z \) be an element of \( \mathcal{E}(D) \) and \( Y \) a directed subset. By monotonicity of \( \psi \) it suffices to show

\[
(\bigcup y)_{y \in Z} \uparrow \uparrow_{EM} \bigcup \{y_z \mid y \in Y\}
\]

which will be abbreviated to \( LHS \uparrow \uparrow_{EM} RHS \). When \( y \) is an element of \( Z \) it is immediate that \( y \) is an element of \( RHS \) because \( y \in Y \) for each \( y \in Y \). When \( y \in Y \) we have \( \psi(y) \) such that \( y \in Y \) and hence \( y \in \psi(Y) \). Also if \( y \in Y \) then \( y \in \mathcal{C}(Y) \) and therefore \( y \in \mathcal{C}(Y \psi Z) \). This shows that \( y \) is an element of \( RHS \). Conversely let \( y' \) be an element of \( RHS \). If \( y \in \mathcal{C}(Z) \) it is immediate to get \( y \) in \( LHS \) such that \( y \in y' \). Otherwise \( y \in \mathcal{C}(Y) \) for every \( y \in Y \). Since \( U \) is an upper bound of \( Y \) this shows that \( (U \psi \psi \{y\}) \) also is. By theorem 5.2:2 \( U \psi \psi \{y\} \uparrow \uparrow_{EM} (U \psi \psi \{y\}) \)

and this gives \( y \) in \( LHS \) such that \( y \in y' \).

When studying the relational powerdomain it was shown how to extend a strict and continuous function \( f: D \rightarrow B \) to a completely additive function \( f^*: \mathcal{E}(D) \rightarrow B \). (This was used to define \( \mathcal{C} \) as a functor and later to define the collecting semantics.) In the present setting \( \mathcal{E}(D) \) and \( B \) may be viewed as having two partial orders. It will be useful to make use of the additional partial order in stating the properties of \( f^* \).

Let \( B = (B, \leq, S) \) and \( C = (C, \leq, \mathcal{C}) \) be two augmented dcpo's and \( g: B \rightarrow C \) a function. It is said to be \( \mathcal{E} \)-monotonic iff \( Y_1 \leq Y_2 \) implies \( g(Y_1) \leq \mathcal{C}(Y_2) \) for all elements \( Y_1 \) and \( Y_2 \) of \( B \). If the augmentations of \( B \) and \( C \) are complete it is said to be completely linear if
g(∪y) = ∪{g(y) ∣ y ∈ Y} holds for every nonempty subset Y of B. We shall use S and X to name these properties of functions and write g: B → C and g: B → C respectively. The property of being ≤-monotonic is admissible if the augmentation of C is admissible. Complete linearity needs not be an admissible property even if the augmentation of C is admissible. To see this define \( g_i : 2^\mathbb{N}_i \rightarrow 2^\mathbb{N}_i \) by

\[
g_i(y) = \{ n \mid \exists y \in \mathbb{N}_i \} \cup \{ 0 \mid \exists n \in \mathbb{N}_i : n < i \}
\]

and \( Y = \{ g_i \mid i \geq 1 \} \). Then \( Y \) is a directed set of completely linear functions but

\[
\bigcup Y = \mathcal{V}_y \{ 1 \mid \exists y \text{ or } y \text{ infinite} \} \cup \{ 0 \mid \exists n \in \mathbb{N}_i : n \neq 1 \}
\]
is not completely linear.

For the extension of functions define the singleton function \{d\}: D → 2(D) by \{d\}(d) = \{d\}. It is clearly strict and monotonic.

**Theorem 5.2.5.** Let D be a benign fdcpo and B a completely augmented dcpo. For each monotonic \( f: D \rightarrow B \) there exists precisely one monotonic and completely linear \( f^\uparrow: 2(D) \rightarrow B \) such that \( f^\uparrow \{d\} = f \). It is given by \( f^\uparrow(y) = \bigcup \{ f(y) \mid y \in Y \} \) and \( \lambda f. f^\uparrow \) is a monotonic and ≤-monotonic bijection from \( D \rightarrow B \) to \( 2(D) \rightarrow B \) and its inverse \( \lambda g. g^\uparrow \) is also monotonic and ≤-monotonic.

**Proof** Suppose that \( g \) is a monotonic and completely linear function such that \( g^\uparrow \{d\} = f \). Then

\[
g(Y) = g(∪\{y\} \mid y \in Y) = \bigcup \{ g(y) \mid y \in Y \} = \bigcup \{ f(y) \mid y \in Y \}
\]
shows that \( g \) must equal \( f^\dagger \). To see that \( f^\dagger \) is as claimed note that

\[ \{ f(y) \mid y \in Y_1 \} \subseteq f_{EM} \{ f(y) \mid y \in Y_2 \} \]

and by monotonicity of \( \cup \) it follows that \( f^\dagger(Y_1) \subseteq f^\dagger(Y_2) \). For complete

linearity of \( f^\dagger \) calculate

\[
\begin{align*}
\left( f^\dagger(\cup y) \right) &= \ldots \\
&= f^\dagger(\cup y) = \\
&= \cup \{ f(y) \mid y \in y \} = \\
&= \cup \{ f^\dagger(y) \mid y \in y \}
\end{align*}
\]

(by \( f^\dagger \) monotonic and \( \cup y \subseteq f_{EM} \cup y \subseteq f_{EM} f_y \))

Clearly \( \lambda f. f^\dagger \) is a bijection with inverse \( \lambda g. g^\dagger \). The inverse

is obviously monotonic and \( \varepsilon \)-monotonic. To see that \( \lambda f. f^\dagger \) is

\( \varepsilon \)-monotonic let \( f_1 \subseteq f_2 \), i.e. \( \forall d \colon f_1(d) \subseteq f_2(d) \). Then \( f_1^\dagger(Y) = \\
\cup \{ f_1(d) \mid d \in Y \} \subseteq \cup \{ f_2(d) \mid d \in Y \} = f_2^\dagger(Y) \). To see that \( \lambda f. f^\dagger \) is

monotonic let \( f_1 \subseteq f_2 \). Then \( f_1^\dagger(Y) = \cup \{ f_1(d) \mid d \in Y \} \subseteq \cup \{ f_2(d) \mid d \in Y \} = \\
f_2^\dagger(Y) \) because \( \cup \) is monotonic.

This theorem makes it possible to extend \( \mathcal{E} \) to a covariant functor

from the category \( BFDCPO \) of benign dcpo's and monotonic functions

to \( AUG \) of augmented dcpo's and monotonic and \( \varepsilon \)-monotonic functions.

The effect upon objects has been defined and for a morphism \( f \colon D \to E \)

define \( \mathcal{E}(f) \colon \mathcal{E}(D) \to \mathcal{E}(E) \) by

\[ \mathcal{E}(f) = (\lambda y. \cup \{ f(y) \mid y \in y \})^\dagger = \lambda y. \cup \{ f(y) \mid y \in y \} \].

Clearly \( \mathcal{E}(f) \) is in \( AUG \) and \( \mathcal{E}(id) = id \). To see that

\[ \mathcal{E}(f_1^\dagger f_2^\dagger) = \mathcal{E}(f_1^\dagger) \circ \mathcal{E}(f_2^\dagger) \]

it suffices by complete linearity to show that
$$\mathcal{B}(f_1^{*} f_2^{*})^{*} = \mathcal{B}(f_1^{*}) \cdot \mathcal{B}(f_2^{*})^{*}$$

and this is straightforward.

For \( g : \mathcal{E}(D) \rightarrow B \) a morphism of \( \mathcal{AUG} \) define \( \text{LIN}(g) : \mathcal{E}(D) \rightarrow B \) by

\[
\text{LIN}(g) = (g^{*}\{\})^{\dagger}.
\]

Clearly this gives a completely linear morphism of \( \mathcal{AUG} \) such that \( \text{LIN}(g)^{*}\{\} = g^{*}\{\} \). As a function \( \text{LIN} \) is monotonic and \( \leq \)-monotonic by the theorem. It further follows from the theorem that \( \mathcal{E}(D) \rightarrow B \) is a dcpo with

\[
\bigcup Y = \\
(\bigcup\{g^{*}\{\} | g \in Y\})^{\dagger} = \\
(\lambda d. \bigcup\{g(d) | g \in Y\})^{\dagger} = \\
\text{LIN}(\lambda Y. \bigcup\{g(Y) | g \in Y\})
\]

The use of \( \text{LIN} \) cannot be avoided in general because complete linearity needs not be an admissible property upon \( \mathcal{E}(D) \rightarrow B \) (and \( \lambda Y. \bigcup\{g(Y) | g \in Y\} \) is the least upper bound in \( \mathcal{E}(D) \rightarrow B \)).

It is because of the above theorem that no continuity assumptions are made. Clearly a function \( f : D \rightarrow B \) from a benign fdcpo to an augmented dcpo is directed continuous because it is monotonic and \( D \) is of finite height. It is not the case, however, that \( f^{\dagger} \) must be directed continuous. To see this define \( f : N_x \times \{\tau\} \rightarrow \mathcal{E}(D) \) by \( f(x, y) = \{y\} \). Further define the directed subset \( Y \) of \( \mathcal{E}(N_x \times \{\tau\}) \) by

\[
Y = \{ \{(1,\tau), \ldots, (j,\tau), (j+1,\tau), \ldots \} | j \geq 1 \}
\]

Then \( f^{\dagger}(Y) = \{L, \tau\} \) for \( \gamma \in Y \) so \( \bigcup\{f^{\dagger}(Y) | \gamma \in Y\} \) equals \( \{L, \tau\} \). Since

\[
\bigcup Y = \{(1,\tau), \ldots \}
\]

this differs from \( f^{\dagger}(\bigcup Y) = \{\tau\} \).
Remark For data flow analysis purposes it is natural to include the empty set as an element of the powerdomain. Then \(\mathcal{E}_{EM}\) needs to be extended to a partial order \(\mathcal{E}\) such that the previous development can still be carried through. It is not clear how to do so. For example if \(\mathcal{E}\) is \(\mathcal{E}_{EM}\) augmented with \(\{\emptyset\}\) and \(\emptyset\) it seems necessary to assume strictness in theorem 5.2:5 and this excludes modelling call-by-name as done in the next section.

Remark Clearly \(\mathcal{E}(N)\) differs from the Smyth, Plotkin and relational powerdomains described in section 3.1. The definition of \(\mathcal{E}(D)\) extends /ApP182/ where it was assumed that \(D\) was flat and countable. It is more difficult to compare \(\mathcal{E}(D)\) with the powerdomains of /Plo82/ as these are not described as (a representation of) a collection of sets. (It is conjectured that this cannot be done, i.e. that contrary to \(\mathcal{E}(D)\) there may be an element \(Y\) such that

\[
\bigcup \{\{y\} | y \in Y\}
\]

does not equal \(Y\).) However, a certain difference between the powerdomains is shown below.

In /Plo82/ the powerdomain \(\mathcal{P}(D)\) and singleton function \(\{\}_1 : D \to \mathcal{P}(D)\) defined there is shown to satisfy an analogue of theorem 5.2:5. This analogue is not fulfilled by \(\mathcal{E}(D)\) and \(\{\}_1 : D \to \mathcal{E}(D)\). The crucial observation is that the extension \(f^+\) of \(f\) in /Plo82/ is always continuous (preserves least upper bounds of chains indexed by \(\omega\)) but it has just been shown that this is not necessarily the case for the \(f^+\) guaranteed by 5.2:5. (We omit the detailed argument.) Similar considerations apply to the other powerdomain defined in /Plo82/.
5.3 STANDARD AND COLLECTING SEMANTICS

In this section we consider the applicative language of recursion equation schemes that was already defined in section 2.4. Its syntax is given by

\[
\text{exp} ::= x_1 \quad \text{(if } i \leq k\text{)}
\]

\[
| F_1(\text{exp}_1, \ldots, \text{exp}_k) \quad \text{(if } i < n\text{)}
\]

\[
| A_i(\text{exp}_1, \ldots, \text{exp}_k) \quad \text{(if } i\text{)}
\]

\[
\text{pro} ::= \text{let } F_1(x_1, \ldots, x_k) = \text{exp}_1 \& \ldots \\
& \quad \& F_n(x_1, \ldots, x_k) = \text{exp}_n \text{ in } \text{exp}
\]

and the parameter mechanism is intended to be call-by-name.

We begin with defining the semantics of this language. This is done relative to an interpretation as in the previous chapters but the notion of an interpretation will be simpler than before. Then the standard and collecting interpretations are defined. The standard semantics agrees with the one given in chapter 2 but the collecting semantics differs from the one of chapter 3 due to the richness of the present powerdomain. The latter is a straightforward consequence of a theorem (a simple analogue of 3.3:14) that shows the relation between the standard and collecting semantics.

An interpretation \( I \) consists of specifications of the components \( L_k, Q, L_1, \text{take}_i, \text{tuple}, a_i \text{ and } \sigma \). Actually \( \text{take}_i \) and \( a_i \) are families \( (\text{take}_i)_{i=1, \ldots, k} \) and \( (a_i)_{i=1, \ldots, k} \) but we shall use the more informal notation. The conditions that they must satisfy are given below together with explanations of the analogous components in the previous chapters:
- $L_k$ and $L_1$ are dcpo's; they correspond to the bottom-level constant domains and the intention is that $L_k = L_1 \times \ldots \times L_1$.

- $Q$ is a predicate upon $L_k \rightarrow L_1$ such that $L_k \rightarrow L_1$ is a dcpo; it corresponds to the bottom-level category.

- $\text{take}_i : L_k \rightarrow L_1$ is much as before.

- $\text{tuple} : (L_k \rightarrow L_1)^k \rightarrow L_k \rightarrow L_k$ is much as before.

- $\text{a}_i : L_k \rightarrow L_1$ corresponds to the constants of contravariantly pure type.

- $\text{a} : (L_k \rightarrow L_1) \times (L_k \rightarrow L_1) \rightarrow L_k \rightarrow L_1$ corresponds to composition in the bottom-level.

As before we shall write $I(L_k)$ etc. when it is not evident which interpretation is considered. The dcpo $L_k$ has been explicitly defined so there is no analogue of the bottom-level domain constructors and therefore it will not be necessary to study a tensor product. It is straightforward to generalise this to allow dcpo's $L_1$ for a class of types $\tau$ but this will not be necessary to illustrate the strong approach. It should be clear that $\ldots \times \ldots$ corresponds to the use of top-level domain constructors.

The semantic functions

$$I(\text{exp}) : (L_k \rightarrow L_1)^n \rightarrow (L_k \rightarrow L_1)$$

$$I(\text{pro}) : L_k \rightarrow L_1$$

is defined (relative to $I$) much as before. The equations are:

$$I(\text{take}_i)\text{env} = \text{take}_i$$

$$I(\text{tuple}(\text{exp}_1, \ldots, \text{exp}_k))\text{env} =$$

$$(\text{env} \downarrow i) \circ \text{tuple}(I(\text{exp}_1)\text{env}, \ldots, I(\text{exp}_k)\text{env})$$

$$I(a_i \circ \text{tuple}(I(\text{exp}_1)\text{env}, \ldots, I(\text{exp}_k)\text{env})$$
The main difference from before is that continuity is not assumed but this causes no problems.

**Fact 5.3.1.** If $I$ is an interpretation then $\overline{\mathcal{I} [\text{pro}]}$ and $\overline{\mathcal{I} [\text{exp}]}$ are defined with functionalities as stated.  

The proof is straightforward by structural induction.

To define the standard interpretation $\mathcal{S}$ let $S$ be some given set. Then put $L_1 = S_\bot$ and $L_k = S_\bot^k$ and let $a_i: S_\bot \to S_\bot$ be unspecified functions. Further let

$\text{take}_i = \lambda(v_1, \ldots, v_k) \cdot v_i$

$\text{tuple}(f_1, \ldots, f_k) = \lambda(v_1, \ldots, v_k)$.

$(f_1(v_1, \ldots, v_k), \ldots, f_k(v_1, \ldots, v_k))$

$\quad f_1 \circ f_2 = \lambda(v_1, \ldots, v_k) \cdot f_1(f_2(v_1, \ldots, v_k))$

Finally, let $Q$ be the constantly true predicate on $L_k \to L_1$. It is straightforward to verify that this gives an interpretation. One could as well have chosen $Q$ to specify continuity because $S_\bot^k$ is of finite height and therefore every function in $L_k \to L_1$ is already continuous. Also tuple and $\circ$ are directed continuous in their arguments so it is straightforward to prove by structural induction that

$\overline{\mathcal{I} [\text{exp}]} : (L_k \to L_1)^n \to L_k \to L_1$

i.e. that $\overline{\mathcal{I} [\text{exp}]}(\text{env})$ depends continuously on $\text{env}$. This implies that, using abbr $1$ above,

$LFP(\text{abbr}_1) = \bigcup_{m=0}^\infty \text{abbr}_1^m(\bot)$
and it is then immediate that this standard semantics agrees with the one of chapter 2.

The aim with the collecting interpretation $C$ is to achieve that $C[\text{pro}] = \mathcal{E}(\text{pro})$. To obtain this we define $C$ by:

- $L_k = \mathcal{E}(S_{k}) = \mathcal{E}(S_{\perp}^k)$
- $L_1 = \mathcal{E}(S_{1}) = \mathcal{E}(S_{\perp})$
- $Q(g) \equiv g$ is $\mathcal{E}$-monotonic
- $\text{take}_1 = \mathcal{E}(\text{take}_1) = \lambda Y. \{v_1 \mid (v_1, \ldots, v_k) \in Y\}$
- $\text{tuple}(g_1, \ldots, g_k) = \text{LIN}(A, g_1(Y) \times \ldots \times g_k(Y))$
- $a_1 = \mathcal{E}(a_1)$
- $g_1 \cdot g_2 = \text{LIN}(g_1, g_2)$

It follows from theorem 5.2:2 that $\mathcal{E}(S_{\perp})$ and $\mathcal{E}(S_{\perp}^k)$ both are dcpo's. Also $\mathcal{E}$-monotonicity is an admissible predicate on $\mathcal{E}(S_{\perp}^k) \rightarrow \mathcal{E}(S_{\perp})$ because the augmentation of $\mathcal{E}(S_{\perp})$ is admissible. It is then straightforward to verify that $C$ is an interpretation.

Apart from the definition of $D$ the definitions are much as might be expected from chapter 3. No tensor products are considered but the main purpose of the tensor product was to obtain $\mathcal{E}(S_{\perp}^k)$ from $k$ copies of $\mathcal{E}(S_{\perp})$ so direct use of $\mathcal{E}(S_{\perp}^k)$ is satisfactory for the collecting semantics. Since $\mathcal{E}$ represents "safe approximation" it is natural that $Q$ expresses some property with respect to this partial order. Because $C[\text{pro}]$ is to be $\mathcal{E}(\text{pro})$, which is completely linear, it might be natural to let $Q$ specify complete linearity. This is analogous to complete additivity in the previous chapters but it was decided against doing so because complete additivity does not hold in general for abstract interpretation. Similar considerations apply here and $\mathcal{E}$-monotonicity is used because it can be enforced in general.
The explanation of the use of LIN in the definition of $\mathcal{C}$ will be postponed because the following result is not affected by the removal of LIN. The result expresses a weaker connection between $S$ and $C$ than the desired $\mathcal{C}(\text{pro}) = \mathcal{E}(S_{\text{pro}})$, namely that $\mathcal{C}(\text{pro})$ is pointwise correct with respect to $S_{\text{pro}}$:

**Lemma 5.32.** \[ \mathcal{C}(\text{pro}) = \{\} \]

**Proof** Let the predicates $R$ and $R_n$ be defined by

\[ R(f, g) \equiv g^* \{f \} = \{g^*f \}
\]

\[ R_n(s_\text{env}, c_\text{env}) \equiv \forall j: R(s_\text{env}^j, c_\text{env}^j) \]

It is a straightforward structural induction to prove that

\[ R_n(i_\text{env}, c_\text{env}) \Rightarrow R(I\exp U(i_\text{env}), C\exp U(c_\text{env})) \]

(whether or not LIN is used in the definition of $\mathcal{C}(\text{pro})$). To show the result it therefore suffices to show

\[ R_n(LFP(F), LFP(G)) \]

for the obvious functionals $F$ and $G$. This result follows from the induction principle of section 5.1 if $R_n$ is an admissible predicate. Clearly $R_n$ is admissible iff $R$ is. It is immediate that $R(\bot, \bot)$ holds so let $Y$ be a directed subset of $(S^k \rightarrow S^k) \times (\mathcal{E}(S^k) \rightarrow \mathcal{E}(S^k))$ such that $R(f, g)$ holds for each $(f, g) \in Y$. To see $R(Y)$, i.e., $R((\bigcup Y)^1, (\bigcup Y)^2)$, we calculate:

\[
(\bigcup Y)^1 = (\bigcup \{g \mid \exists f: (f, g) \in Y\})^1 = (\text{pointwise least upper bounds in } \mathcal{E}(S^k) \rightarrow \mathcal{E}(S^k))
\]

\[
\bigcup \{g^* \mid \exists f: (f, g) \in Y\} = \bigcup \{f^* \mid \exists g: (f, g) \in Y\}
\]
The use of LIN in the definition of $C(Q)$ is one way of achieving that $C[proj]$ is completely linear and the desired result then follows from the previous lemma:

**Theorem 5.3.3.** $C[proj] = E(S[proj])$  

**Proof.** It is immediate that the placement of $\mathfrak{L}$ ensures that $C[exp](\text{cenv})$ always is completely linear. Therefore $C[proj]$ is completely linear as well and equals $\left(\mathfrak{C}[proj]\right)^\dagger$.

Using the previous lemma this equals $E(S[proj])$.  

There are other ways that can be used to obtain that $C[proj]$ is completely linear. The present choice was inspired by /Hen82/ that considers

$$g_1 \cdot g_2 = \text{LIN}(g_1) \cdot g_2$$

(for some notion of LIN). In the present setting $g_2$ will always be tuple(...), hence completely linear, so $\text{LIN}(g_1) \cdot g_2$ agrees with $\text{LIN}(g_1 \cdot g_2)$. Another possibility is to omit LIN from the definition of $\mathfrak{L}$ and let $Q$ be complete linearity (as was decided against above). A third and closely related possibility is to omit LIN from the definition of $\mathfrak{Q}$ and replace $\text{LFP}(G)$ by $\text{LIN}_n(\text{LFP}(G))$ where $\text{LIN}_n(g_1, \ldots, g_n) = (\text{LIN}(g_1), \ldots, \text{LIN}(g_n))$. It can be shown that all possibilities give the same results and that for the present choice

$\text{LFP}(G) = \text{LIN}_n(\bigcup_{m=0}^\infty G^m(\lambda)) = G(\bigcup_{m=0}^\infty G^m(\lambda))$
We omit the proof: it relies on the continuity of $S^e$, the proof of the induction principle of section 5.1 and the characterisation in section 5.2 of least upper bounds in $\mathcal{L}_k \to \mathcal{L}$.

5.4 ABSTRACT INTERPRETATION

As in the previous chapters data flow analyses are specified by certain interpretations (called approximating interpretations). In this section it is studied how to prove the correctness of an approximating semantics with respect to the collecting semantics or another approximating semantics. We also study how to induce an approximating interpretation from the collecting interpretation or another approximating interpretation. The framework differs from that of the previous chapters because $\mathcal{L}_k$ and $\mathcal{L}$ are naturally equipped with two partial orders and we shall argue that this should be the case for all approximation spaces.

Approximating interpretation

So let $\mathcal{I} = (L_k, Q, \mathcal{L}, \text{take}, \text{tuple}, a, \mathcal{V})$ be an interpretation that expresses some data flow analysis. The connection between $L_k$ and $\mathcal{L}_k$ and between $\mathcal{L}$ and $\mathcal{L}$ will be given by concretization functions:

$$\text{con}_k: L_k \to \mathcal{L}_k$$
$$\text{con}_1: \mathcal{L} \to \mathcal{L}$$

The desired relation between $\mathcal{I}$ and the collecting interpretation $\mathcal{C}$ is that $\mathcal{I}[\text{pro}]$ should be a safe description of $\mathcal{C}[\text{pro}]$. We shall formalise this by

$$\mathcal{C}[\text{pro}] \circ \text{con}_k \subseteq \text{con}_1 \circ \mathcal{I}[\text{pro}]$$
rather than
\[ \mathcal{F}(\text{proj} \cdot \text{con}^k \mathcal{E}_\text{EM} \text{ con}^i \text{if proj}) \]
as was used in the previous chapters. (Here \( \mathcal{E} \) and \( \mathcal{E}_\text{EM} \) are extended pointwise to functions.) This is because it is \( \mathcal{E} \) and not \( \mathcal{E}_\text{EM} \) that expresses "safe approximation": it is evident that \( \mathcal{E} \) expresses that every result that is produced by the collecting semantics is included among those produced by the abstracting semantics. This is not the case with \( \mathcal{E}_\text{EM} \): it is problematic that \( \{1, 2\} \mathcal{E}_\text{EM} \{2\} \) holds because then non-termination may be ignored, and it is problematic that \( \{2\} \mathcal{E}_\text{EM} \{2, 3\} \) fails because this makes it almost impossible to approximate.

Not all interpretations specify data flow analyses and we now redefine the notion of approximating interpretation to include those that do. The collecting interpretation is to be one and we have seen that both \( \mathcal{E}_\text{EM} \) and \( \mathcal{E} \) were relevant for \( \mathcal{E}(S^k_\bot) \) and \( \mathcal{E}(S_\bot) \). If \( \mathcal{I} \) is to be an approximating interpretation it is natural that \( L_k \) and \( L_1 \) also have two partial orders. One is written \( \mathcal{E} \) and is used to guarantee the existence of least fixed points in the same way that \( \mathcal{E}_\text{EM} \) has been used. The other is written \( \mathcal{E} \) and expresses safe approximation in much the same way that subset inclusion does. The intuitive difference between \( \mathcal{E} \) and \( \mathcal{E} \) was already mentioned in /Nie81a/ and /Nie82/.

Since \( L_k \) and \( L_1 \) are equipped with the partial order \( \mathcal{E} \) it is natural to let \( Q \) express some property with respect to \( \mathcal{E} \). We shall use \( \mathcal{E} \)-monotonicity because it seems obvious that performing a computation upon a safe description of some argument must give a result that safely describes the result of computing upon the original
argument. Stronger preservation properties such as complete linearity may fail in general and for much the same reasons that additivity could not be assumed in general in chapter 4. The partial order can be extended pointwise so as to relate functions in $L_k \rightarrow L_1$. It is natural to assume that tuple and $\sigma$ are $\preceq$-monotonic in each argument. When $\preceq$ is extended componentwise to cartesian products this just means that tuple and $\sigma$ are $\preceq$-monotonic.

This discussion motivates:

**Definition** An approximating interpretation is a specification of $L_k$, $Q$, $L_1$, $\text{take}_i$, tuple, $a_i$, $\sigma$ such that:

- $L_k$ and $L_1$ are augmented dcpo's with the augmentation of $L_1$ admissible,
- $Q$ specifies $\preceq$-monotonicity,
- $\text{take}_i : L_k \rightarrow L_i$ and $a_i : L_i \rightarrow L_1$,
- tuple: $(L_k \rightarrow L_1)^k \preceq (L_k \rightarrow L_i)$ and $\sigma : (L_k \rightarrow L_1)^k \times (L_k \rightarrow L_i) \rightarrow (L_k \rightarrow L_i)$

The predicate $Q$ is admissible upon $L_k \rightarrow L_1$ because the augmentation of $L_1$ is admissible. It is therefore straightforward to prove that an approximating interpretation is an interpretation and that least upper bounds of directed subsets of $L_k \rightarrow L_1$ are pointwise (as in $L_k \rightarrow L_1$).

If `if...then...else...` had been an explicit construct in the language it would be convenient to assume that the augmentation was complete.

Finally, note that the collecting interpretation is an approximating interpretation.
Relating two interpretations

Next we show how to relate approximating interpretations \( I \) (as above) and \( I' = (L_k', Q', L_1', \text{take}', \text{tuple}', a_i', \sigma') \) in such a way that \( I'_{\text{proj}} \) is a safe approximation to \( I_{\text{proj}} \). To express the relation we need concretization functions

\[
\text{con}_k : L_k' \rightarrow L_k
\]
\[
\text{con}_i : L_i' \rightarrow L_i
\]

and we shall refer to them as \( \text{con} = \langle \text{con}_i \rangle_{i=1,k'} \). The desired relation between \( I_{\text{proj}} \) and \( I'_{\text{proj}} \) is \( \leq_{\text{con}} \) defined on \( (L_k \quad \text{exp}_{L_i \rightarrow L_1'}) \) by

\[
g \leq_{\text{con}} g' \iff \text{con}_k g \leq_{\text{con}} \text{con}_1 g'
\]

We shall also need \( \leq_{\text{con}} \) to be defined on \( (L_k \quad \text{exp}_{L_i \rightarrow L_1'}) \) by a similar formula. The relation may be extended to approximating interpretations by defining \( I \leq_{\text{con}} I' \) to hold whenever

- \( \text{take}_i \leq_{\text{con}} \text{take}_i' \) and \( \text{a}_i \leq_{\text{con}} \text{a}_i' \) (for all \( i \))
- \( g_i \leq_{\text{con}} g_i' \) (for all \( i \)) implies \( \text{tuple}(g_1, \ldots, g_k) \leq_{\text{con}} \text{tuple}'(g_1', \ldots, g_k') \) and
- \( g_i \leq_{\text{con}} g_i' \) (for \( i=1,2 \)) implies \( g_1 \sigma g_2 \leq_{\text{con}} g_1' \sigma' g_2' \)

This definition closely follows the pattern of chapter 4.

To prove that \( I_{\text{proj}} \leq_{\text{con}} I'_{\text{proj}} \) we shall need more assumptions than merely \( I \leq_{\text{con}} I' \). These express conditions about how \( \text{con}_1 \) preserves aspects of the partial order \( \sqsubseteq \) with respect to the partial order \( \leq \). We say \( \text{con}_1 \) is pseudo-strict iff \( \sqsubseteq_{\text{con}_1}(\sqsubseteq) \). If \( L_1 = \sqsubseteq(S_1) \) it is evident that this means that \( \sqsubseteq \) of \( L_1' \) must represent the possibility of a non-terminating outcome. The function \( \text{con}_1 \) is pseudo-continuous iff
Ij Icon 1 Yj Scon, (Uy)
holds for every directed subset Y of L;'. (This is essentially the
dual notion of the quasi-continuity considered in /Plo82/.) It is
immediate that strictness and directed continuity imply pseudo-
strictness and pseudo-continuity, respectively.

Theorem 5.5: If I ≤ con I' and con_ is pseudo-strict and pseudo-
continuous then I (pro) ≤ con I' (pro) holds for all programs pro. //

Proof First extend the relation ≤ con to tuples by defining
env ≤ env' to mean that env_i ≤ con env'_i holds for all i. It is
then a straightforward structural induction to show that

env ≤ env' implies I (exp) (env) ≤ con I' (exp) (env')

To show the result for pro = let ... in exp_0 it therefore suffices
to show that LFP(G) ≤ LFP(G') where

G(env) = (I (exp_1) (env), ..., I (exp_n) (env))

and G' is defined similarly. Since env ≤ env' implies that
G(env) ≤ G'(env') it suffices to show that ≤ is admissible as the
result then follows from the induction principle of section 5.1.

The relation ≤ is admissible iff ≤ con is upon (L_k → L_1) x (L_k → L'_1).
It is immediate that ≤ con because con_ is pseudo-strict. Next
let Y be a directed set such that g ≤ con g' holds for all (g, g') ∈ Y.
For (U_Y) L1 ≤ con (U_Y) L2 calculate as follows:

∀(g, g') ∈ Y: g ≤ con g' ⇒

∀(g, g') ∈ Y: g' con_k ≤ con_1 g' ⇒

(≤ is admissible on L_1)

∪{g' con_k | (g, g') ∈ Y} ≤ ∪{con_1 g', (g, g') ∈ Y} ⇒
(pointwise least upper bounds)
\[(U^Y)_1, con_k \leq U^y, con_1 | (g, g') \epsilon Y \Rightarrow\]
(con_1 is pseudo-continuous)
\[(U^Y)_1, con_k \leq con_1 | (U^y)_2 \Rightarrow\]
\[(U^Y)_1 \leq con | (U^y)_2\]

Inducing an interpretation

Also the notion of inducing an interpretation may be defined in the strong approach. To do so requires abstraction functions

\[
\begin{align*}
\text{abs}_k & : L_k \rightarrow L'_k \\
\text{abs}_1 & : L_1 \rightarrow L'_1 
\end{align*}
\]
in addition to the concretization functions con_k and con_1. In the previous chapter "adjointness" was used to express a desirable connection between abstraction and concretization functions. Here "adjointness" should be defined with respect to \(\preceq\) because it is this partial order that expresses "safe approximation". It is reasonable to assume that abstraction and concretization functions are \(\preceq\)-monotonic because this means that they preserve "safe approximation".

We then redefine:

**Definition** A pair \((\text{abs}: L \rightarrow L', \text{con}: L' \rightarrow L)\) of functions between augmented dcpo's is a pair of **adjointed** functions iff \(\text{abs}\) and \(\text{con}\) are \(\preceq\)-monotonic and monotonic and

\[
\text{abs'} \preceq \text{al'}, \text{and} \quad \text{con'} \preceq \text{al}
\]

Function \(\text{abs}\) is said to be the lower adjoint of \(\text{con}\) and \(\text{con}\) is the upper adjoint of \(\text{abs}\).

As before, lower and upper adjoints are unique if they exist. It will
be used below that they must be monotonic.

Let $I$ be an approximating interpretation with components as above and let $L_k'$ and $L'_1$ be augmented dcpo's such that the augmentation of $L'_1$ is admissible. Further let $(\text{abs}_k : L_k \rightarrow L'_k, \text{con}_k)$ and $(\text{abs}_1 : L_1 \rightarrow L'_1, \text{con}_1)$ be pairs of adjoined functions. The induced interpretation

$$I' = \text{induce}(I, (\text{abs}_1, \text{con}_1), (L'_1))$$

has $I'(L_k) = L_k'$, $I'(L_1) = L_1'$ and $I'(Q)$ to be $\preceq$-monotonicity. The remaining components are given by:

$$\begin{align*}
\text{take}_i' &= \text{abs}_i \cdot \text{take}_i \cdot \text{con}_k \\
tuple'(g_1, \ldots, g_k) &= \text{abs}_k \cdot \tuple(\text{con}_1, g_1 \cdot \text{abs}_k, \ldots) \cdot \text{con}_k \\
a'_i &= \text{abs}_1 \cdot a_i \cdot \text{con}_k \\
g_1 \cdot g_2 &= \text{abs}_1 \cdot (\text{con}_1, g_1 \cdot \text{abs}_k) \cdot (\text{con}_k, g_2 \cdot \text{abs}_k) \cdot \text{con}_k
\end{align*}$$

It is straightforward to verify that this gives an approximating interpretation. The following lemma shows that $I'$ is correct and as precise as possible:

**Lemma 5.4:2.** Let $I$, $L'_1$, $\text{abs}_i$ and $\text{con}_i$ be as above. Then

1. $I \preceq \text{induce}(I, (\text{abs}_1, \text{con}_1), (L'_1))$
2. $I \preceq I''$ iff $\text{induce}(I, (\text{abs}_1, \text{con}_1), (L'_1)) \preceq \text{id } I''$

where id is the family of identity functions.

**Proof** For (1) let $I'$ be defined as above. It is immediate that $\text{take}_i \preceq \text{take}_i'$ and $a_i \preceq a_i'$. Next let $g_i \preceq g_i'$ so that

$$g_i \preceq \text{abs}_1 \cdot g_i' \cdot \text{con}_k$$

By $\preceq$-monotonicity of tuple

$$\tuple(g_1, \ldots, g_k) \preceq \tuple(\text{abs}_1 \cdot g_1' \cdot \text{con}_k, \ldots)$$
from which it follows that
\[ \text{tuple}(g_1, \ldots, g_k) \subseteq \text{tuple}'(g'_1, \ldots, g'_k) \]

That \( g_i \not\subseteq \text{con} g'_i \) implies \( g_i g_2 \not\subseteq \text{con} g'_i g'_2 \) follows similarly.

For (2) let \( I'' = (I_k', Q', L_i', \text{take}_i', \text{tuple'}, a''', \sigma') \) and consider "if".

When \( I' \not\subseteq \text{id} I'' \) it is immediate that \( a_i \not\subseteq \text{con} a''_i \) and \( \text{take}_i \not\subseteq \text{con} \text{take}'_i \).

Next let \( g_i \not\subseteq \text{con} g''_i \) so that
\[ \text{abs}_i g_i \text{con}_k \subseteq g''_i \]

Then by \( I' \not\subseteq \text{id} I'' \)
\[ \text{tuple}'(\text{abs}_i g_i \text{con}_k, \ldots) \subseteq \text{tuple}''(g''_1, \ldots) \]

and using the definition of tuple' this gives
\[ \text{tuple}(\text{con}_1 \text{abs}_1 g_1 \text{con}_k \text{abs}_k, \ldots) \text{con}_k \subseteq \text{con}_1 \text{tuple}''(g''_1, \ldots) \]

By \( \subseteq \)-monotonicity of tuple this gives
\[ \text{tuple}(g_1, \ldots, g_k) \not\subseteq \text{con} \text{tuple}''(g''_1, \ldots, g''_k) \]

In a similar way it is shown that \( g_i \not\subseteq \text{con} g''_i \) implies that
\[ g_i g_2 \not\subseteq \text{con} g''_i \sigma''_i g''_2. \]

It follows that \( I \not\subseteq \text{con} I'' \).

For "only if" in (2) assume that \( I \not\subseteq \text{con} I'' \). It is immediate that
\[ a_i \not\subseteq \text{id} a''_i \] and \( \text{take}_i \not\subseteq \text{id} \text{take}'_i \).

Next let \( g_i \not\subseteq \text{id} g''_i \) so that
\[ \text{con}_1 g_i \text{abs}_k \not\subseteq \text{con} g''_i \]

This gives
\[ \text{tuple}(\text{con}_1 g_i \text{abs}_k, \ldots) \not\subseteq \text{con} \text{tuple}''(g''_1, \ldots) \]

and
\[ \text{tuple}'(g_1', \ldots, g_k') \not\subseteq \text{id} \text{tuple}''(g''_1, \ldots, g''_k) \]

follows much as in the proof of "if". The result for \( a' \) and \( a'' \) is
similar and $I' \leq_{Id} I''$ follows. ///

If additionally $\text{con}_1$ is pseudo-strict and pseudo-continuous it follows by theorem 5.4:1 that $I''[\text{proj}]$ is correct with respect to $\text{proj}$.

As in the previous chapter it is helpful to be able to construct data flow analyses in stages. It is straightforward to prove that if

$I' = \text{induce}(I, (\text{abs}_{1}, \text{con}_{1})_{1}, (L'_{1})_{1})$

$I'' = \text{induce}(I', (\text{abs}_{1}', \text{con}_{1}')_{1}, (L''_{1})_{1})$

then

$I'' = \text{induce}(I', (\text{abs}_{1}', \text{con}_{1}')_{1}, (\text{abs}_{1}', \text{con}_{1}')_{1}, (L''_{1})_{1})$

where $(\text{abs}_{1}, \text{con}_{1})_{1} = (\text{abs}_{1}', \text{con}_{1})_{1}$. The proof is analogous to the proof of "if" for the second statement of lemma 5.4:2. That theorem 5.4:1 will be applicable for $I$ and $I'$ if it is for $I$ and $I''$ and for $I'$ and $I''$, follows from:

Fact 5.4:3. If $\text{con}_{1}$ and $\text{con}_{1}'$ are pseudo-strict and pseudo-continuous upper adjoints then so is $\text{con}_{1}' \text{con}_{1}'$. ///

This shows that one obtains the same result whether or not the construction is "in stages".

Similar results hold for proving approximating interpretations correct. Here

$I \leq_{\text{con}} I'$ and $I \leq_{\text{con}} I''$ and $\text{con}$ and $\text{con}'$ families of upper adjoints imply that $I \leq_{\text{con}' \text{con}} I''$

where $(\text{con}_{1})_{1}'(\text{con}_{1}')_{1} = (\text{con}_{1}' \text{con}_{1})_{1}$. The proof is analogous to the proof of "if" for the second statement of lemma 5.4:2. That theorem 5.4:1 will be applicable for $I$ and $I'$ if it is for $I$ and $I''$ and for $I'$ and $I''$, follows from:

Fact 5.4:3. If $\text{con}_{1}$ and $\text{con}_{1}'$ are pseudo-strict and pseudo-continuous upper adjoints then so is $\text{con}_{1}' \text{con}_{1}'$. ///

261
5.5 APPLICATIONS

The major application in this section is to show the correctness of two data flow analyses used in /Myc81/ for detecting situations where call-by-name can safely be replaced by call-by-value. First, however, we show the connection between the "strong" approach of this chapter and the "weak" approach of the previous chapters. This also gives a connection with the traditional theory of abstract interpretation (/CoCo77b/,...).

"Strong" versus "weak"

Let \( C \) be the collecting interpretation as defined in section 5.3. We define an approximating interpretation \( W \) that is like the collecting interpretation of chapter 3 (for the applicative language considered in section 2.4). It is obvious that \( W(L_1) \) should be \( \varphi(S_{\perp}) \).

For this to make sense \( S_{\perp} \) must be an algebraic cpo so we must assume that \( S \) is countable. Clearly \( \varphi(S_{\perp}) \) is a dcpo but it is not an augmented dcpo because \( \varphi(S_{\perp}) = (\varphi(S_{\perp}), \subseteq) \) is only equipped with one partial order (which happens to be subset-inclusion). Since \( \subseteq \) is used both to obtain least fixed points and to express safeness it should be turned into an augmented dcpo by setting

\[
\varphi(S_{\perp}) = (\varphi(S_{\perp}), \subseteq, \subseteq)
\]

and clearly the augmentation is admissible. According to chapter 3 \( W(L_1) \) should be \( \varphi(S_{\perp}) \circ \ldots \circ \varphi(S_{\perp}) \) but since this is isomorphic to \( \varphi(S_{\perp}^k) \) we shall use the latter. It is made into an augmented dcpo as above. The predicate \( W(Q) \) is \( \subseteq \)-monotonicity and this reduces to monotonicity.
It is then straightforward to define the remaining components (building on the definition in section 3.3):

\[ W(\text{take}_i) = \lambda Y. \{ y_i \mid (y_1, \ldots, y_k) \in Y \} \]

\[ W(\text{tuple})(g_1, \ldots, g_k) = \lambda Y. \lambda \gamma. \prod_{i=1}^{k} \gamma_i(g_i(\{ \gamma \}) \mid \gamma \in Y) \]

\[ W(a_i) = \lambda Y. \lambda \gamma. \prod_{i=1}^{k} \gamma_i(a_i(\{ \gamma \}) \mid (y_1, \ldots, y_k) \in Y) \]

\[ W(\sigma)(g_1, g_2) = g_1 \cdot g_2 \]

It is straightforward to verify that \( W \) is an approximating interpretation. It should also be clear that it corresponds to the collecting interpretation of chapter 3. In particular it makes no harm that \( W(\varnothing) \) does not specify continuity as in chapter 3 because continuity is an admissible property so the least fixed points will be the same.

To show that the present approach subsumes that of the previous chapters we first show that \( W \) can be proved correct with respect to \( C \). For this define the concretization functions

\[ \text{con}_i : \wp(S_i^1) \to \wp(S_i^1) \]

( \( S_i^1 \) is \( S_1 \) ) by \( \text{con}_i(Y) = Y \). The abstraction functions are defined by \( \text{abs}_i(Y) = \lambda Y. \prod_{i=1}^{k} \gamma_i(Y) \). Then \( (\text{abs}_i, \text{con}_i) \) and \( (\text{abs}_k, \text{con}_k) \) are pairs of adjoined functions in the sense of section 5.4. For monotonicity note that \( Y_1 \subseteq Y_2 \) implies \( Y_1 \subseteq Y_2 \) when \( Y_1 \) contains \( \bot \) and that \( Y_1 \subseteq Y_2 \) implies \( \lambda Y. \prod_{i=1}^{k} \gamma_i(Y) \subseteq \lambda Y. \prod_{i=1}^{k} \gamma_i(Y) \) when \( Y_1 \) and \( Y_2 \) are elements of \( \wp(S_i^1) \). Further \( \text{con}_i \) is pseudo-strict and pseudo-continuous. For pseudo-continuity it suffices by admissibility of \( \subseteq \) upon \( \wp(S_i^1) \) to show that

\[ \text{con}_i(Y) \subseteq \text{con}_i(\bigcup Y) \]
whenever $Y$ is an element of a directed set $Y$ of $\mathcal{P}(S_{\perp})$ and this is immediate by $\bigcup Y = UY$. It is straightforward to show that $C \subseteq \text{con} W$ so theorem 5.4:1 shows that $\text{C proj} \subseteq \text{w proj}$ holds for all programs $\text{pro}$.

To fully show that the present approach subsumes the previous approach it must be shown how pairs of adjoined functions and $\subseteq \text{con}$ carry over. So let $I$ and $I'$ be approximating interpretations in the sense of chapter 4. They can be turned into approximating interpretations $I$ and $I'$ in the sense of section 5.4 in the same way $W$ was obtained. Next let $(\text{abs }_C, \text{con }_C)$ be a family of pairs of adjoined functions (in the sense of chapter 4) from $I$ to $I'$. They will still be pairs of adjoined functions in the sense of section 5.4 and the assumed strictness and continuity of $\text{con }_C$ gives pseudo-strictness and pseudo-continuity. Further if $I \subseteq \text{con } I'$ in the sense of chapter 4 then this relation still holds in the sense of section 5.4.

Remark To compare this approach with the traditional theory of e.g. /CoCo77b/ it might be best to restrict $k$ to 1 and augment the language with an explicit conditional. This is because the applicative language then closely corresponds to a flowchart. Alternatively $\mathcal{P}(S_{\perp}X \ldots Z_{\perp})$ can be replaced by $\mathcal{F}(S_{\perp}X \ldots Z_{\perp})$ as the latter is isomorphic to the powerset of $S^k$. Then one should use

\[
\text{con }_k(Y) = Y \cup \bigcup_{i=1}^{k} S_{\perp}X \ldots Z_{\perp} \]

\[
\text{abs }_k(Y) = \{ \text{smash}(y) \mid y \notin Y \}
\]

instead of the definitions above.
The data flow analyses from /Myc81/

We now turn towards the major application of this section. We shall formulate and prove correct two analyses used in /Myc81/. The first can be handled in the weak approach of the previous chapters whereas this seems impossible for the second. Both analyses were "proved correct" in /Myc81/ but the proof of the second fails (as will be pointed out) because it is based on the Plotkin powerdomain.

It is convenient to define the two analyses by inducing them from the following interpretation $\mathcal{I}$ that is essentially an independent attribute method formulation of the collecting interpretation.

$$L_k = \mathcal{E}(S_k)^k$$

$$L_1 = \mathcal{E}(S_1)$$

$Q$ is constantly true

take $i = \lambda(y_1, \ldots, y_k). y_i$

tuple$(g_1, \ldots, g_k) = \lambda(y_1, \ldots, y_k). (g_1(y_1, \ldots, y_k), \ldots, g_k(y_1, \ldots, y_k))$

$$a_i = \lambda(y_1, \ldots, y_k). \{s(a_i)(y_1, \ldots, y_k) | \forall i: y_i \in \{\text{lin}\} \}

$$g_1^*g_2 = g_1^*g_2$$

It should not be surprising that LIN has been removed from tuple because $\mathcal{I}$ is an independent attribute method and it is convenient to remove it from $\mathcal{I}$ as well. It is straightforward to verify that this gives an approximating interpretation. The connection with the collecting interpretation is given by

$$\text{con}_k = \lambda(y_1, \ldots, y_k). y_1 \times \ldots \times y_k$$

$$\text{con}_1 = \lambda y. y$$

It is straightforward to prove that $\text{con}_1$ is pseudo-strict and pseudo-continuous and that $\mathcal{C} \subseteq \text{con}_1$. 

265
We shall formalise the first data flow analysis (§ of /Myc81/) as an approximating interpretation $L$. Let $\emptyset$ be the set $\{0,1\}$. The elements of $L(L_1)$ and $L(L_k)$ are to be $0$ and $0^k$ respectively. The intention is that $0$ represents guaranteed non-termination whereas $1$ represents possible termination. So if

$$L\text{proj}(0,1,\ldots,1) = 0$$

this means that the program cannot terminate unless its first argument represents a terminating computation. This again means that the first parameter (which is call-by-name) can safely be passed by value. Consult /Myc81/ for further information.

To define $L$ as induced from $I$ we must turn $0$ and $0^k$ into augmented dcpo's and define concretization and abstraction functions. On $0$ there is the partial order $\leq$ (less than or equal to) and it may be extended componentwise to $0^k$. Then $L(L_1)$ and $L(L_k)$ will be $(0,\leq,\varepsilon)$ and $(0^k,\leq,\varepsilon)$ respectively and clearly they are admissibly augmented dcpo's. The concretization functions

$$\text{con}_i : (0^0,\leq,\varepsilon) \to I(L_i) \quad i = 1,k$$

(for $0^1 = 0$) are defined by

$$\text{con}_i(0) = \{1\} , \text{con}_i(1) = S_1$$

$$\text{con}_k(v_1,\ldots,v_k) = (\text{con}_1(v_1),\ldots,\text{con}_k(v_k))$$

It is immediate that $\text{con}_i$ is pseudo-strict and pseudo-continuous. Pairs of adjoined functions $(\text{abs}_i,\text{con}_i)$ and $(\text{abs}_k,\text{con}_k)$ may be obtained by defining $\text{abs}_i(Y) = 0$ if $Y = \{1\}$ and $\text{abs}_i(Y) = 1$ otherwise and $\text{abs}_k(Y_1,\ldots,Y_k) = (\text{abs}_1(Y_1),\ldots,\text{abs}_k(Y_k))$. Then $L$ is defined by

$$L = \text{induce}(I,(\text{abs}_i,\text{con}_i)i,((0^0,\leq,\varepsilon))_i)$$

Clearly $L$ is correct with respect to $C$, i.e. $C \varepsilon \text{con} \cdot \text{con} \cdot L$. Further
the conditions of theorem 5.4:1 are fulfilled so that

\[ \xi(S[pro]) \cdot con_k \cdot con_k' \leq con_1 \cdot con_1' \cdot L[pro] \]

holds. This implies that the informal description of \( L \) that was given above is correct.

**Example** Let pro be the following program:

```plaintext
let \( F_1(x_1,x_2,x_3,x_4,x_5) = \)

\( \text{if } x_1=1 \text{ then } x_2+x_3 \text{ else } F_1(x_1-1,x_3,x_2,x_5,x_4) \)

in \( F_1(x_1,x_2,x_3,x_4,x_5) \)
```

Formally this is not a program because all of \( x_1=1, x_2+x_3, x_1-1 \) and \( \text{if...then...else...} \) ought to be written \( \lambda_i(x_1,x_2,x_3,x_4,x_5) \) for some \( i \).

We shall, however, use the shorthands \( x_1=1 \) etc. Further we assume that the standard interpretation \( S \) has the set \( S \) to be the integers and that the functions "\( x_1=1 \)" etc. are interpreted as intended. Since there is only one type, truthvalues should be coded by integers, e.g. 1 means true and all other numbers mean false.

From the definition of \( L \) (and \( I \)) it is straightforward to calculate that

\[ L(\emptyset)(g_1,g_2) = g_1 \cdot g_2 \]

\[ L(\text{tuple})(g_1,\ldots,g_5) = \lambda(v_1,\ldots,v_5). \]

\[ \text{(}g_1(v_1,\ldots,v_5), \ldots, g_5(v_1,\ldots,v_5)\text{)} \]

\[ L(\text{take}_i) = \lambda(v_1,\ldots,v_5). v_i \]

and for examples of \( L(a_i) \)

\[ L("x_2+x_3") = \lambda(v_1,\ldots,v_5). \min\{v_2,v_3\} \]

\[ L("\text{if } x_1 \text{ then } x_2 \text{ else } x_3") = \lambda(v_1,\ldots,v_5). \min\{v_1,\max\{v_2,v_3\}\} \]

One can then calculate
\[ L \text{proj} = \lambda(v_1,\ldots,v_3).\min\{v_1,v_2,v_3\} \]

using that the function \( G \) of which the least fixed point is required has \( \text{LFP}(G) = G(\bot) \). The conclusion is that the first three parameters to the program may safely be passed by value.

The second analysis we shall consider is \( \Phi \) of /Myc81/. It will be formulated as an approximating interpretation \( \mathfrak{M} \) and again the elements of \( \mathfrak{M}(L_1) \) and \( \mathfrak{M}(L_k) \) are \( \emptyset \) and \( \emptyset^k \) respectively. The intention with \( 0 \) now is to denote possible non-termination and with \( 1 \) to denote guaranteed termination. So if

\[ \mathfrak{M}(\text{exp}_i(\text{env})(v_1,\ldots,v_k) = 1 \]

for some environment \( \text{env} \) and values \( v_1 \) then it is safe to use call-by-value for the \( i \)'th parameter in the call \( F(\text{exp}_1,\ldots,\text{exp}_{i-1},\text{exp}_{i+1},\ldots,\text{exp}_k) \).

We refer to /Myc81/ for further information.

The augmented dcpo's \( \mathfrak{M}(L_i) \) will be \((\emptyset^i,\emptyset,\emptyset)\). The use of \( \emptyset \) is essential in order to achieve \( \preceq \)-monotonicity of the concretization functions

\[ \text{con}_i : (\emptyset^i,\emptyset,\emptyset) \to \mathfrak{I}(L_1) \]

defined by

\[ \text{con}_i(0) = S_\bot \quad \text{con}_i(1) = S \]
\[ \text{con}_k(v_1,\ldots,v_k) = (\text{con}_i(v_1),\ldots,\text{con}_i(v_k)) \]

Also note that \( S \) is an element of \( \mathfrak{P}(S_\bot) \) but not (unless \( S \) is finite) of the Plotkin powerdomain which was used in /Myc81/. Clearly \( \text{con}_i \) is pseudo-strict and pseudo-continuous and pairs of adjoined functions \( (\text{abs}_i,\text{con}_i) \) and \( (\text{abs}_k,\text{con}_k) \) are obtained by defining

\[ \text{abs}_i(Y) = 1 \text{ if } Y \subseteq X \] and \( \text{abs}_i(Y) = 0 \) otherwise
\[ \text{abs}_k(Y_1,\ldots,Y_k) = (\text{abs}_i(Y_1),\ldots,\text{abs}_i(Y_k)) \]
Then $M$ is defined by

$$M = \text{induce}(I_i, (\text{abs}^i, \text{con}^i))_{i=1, \ldots, 5, 2}$$

and as before it is correct with respect to $C$.

**Example** Returning to the example program we may calculate:

$$M(n)(g_1, g_2) = g_1 \cdot g_2$$

$$M(\text{tuple})(g_1, \ldots, g_5) = \lambda(v_1, \ldots, v_5).$$

$$(g_1(v_1, \ldots, v_5), \ldots, g_5(v_1, \ldots, v_5))$$

$$M(\text{take}_i) = \lambda(v_1, \ldots, v_5).v_i$$

and for examples of the $M(a_i)$:

$$M("x_2x_3") = \lambda(v_1, \ldots, v_5).\min\{v_2, v_3\}$$

$$M("\text{if } x_1 \text{ then } x_2 \text{ else } x_3") = \lambda(v_1, \ldots, v_5).\min\{v_1, v_2, v_3\}$$

Much as before this allows to calculate that

$$M[\text{prof}] = \lambda(v_1, \ldots, v_5).0$$

which is clearly correct and in fact as precise as possible.  ///

To gain perspective upon this data flow analysis consider the program $\text{prof}'$ that is $\text{prof}$ with $x_1 = 1$ replaced by $x_1 \leq 1$. Then

$$M[\text{prof}'] = M[\text{prof}]$$

which is still correct but not as precise as possible because

$$(\text{abs}^i[\text{prof}'], \text{con}^i) = \lambda(v_1, \ldots, v_5).\min\{v_1, v_2, v_3\}$$

That is the program terminates if the arguments in the first three positions do but $M$ does not detect this. This is not because $M$ is badly chosen but more fundamentally because abstract interpretation has only been considered in a "first-order" manner: functions are viewed as transforming (descriptions of) sets of arguments to sets of results and there is no provision for expressing any relationship.
between the arguments and the result (e.g. the result is less than
the first argument). We shall return to this in the conclusion.

5.6 NONDETERMINISM AS ABSTRACT INTERPRETATION

Nondeterminism means different things in different contexts. In
automata theory the nondeterminism chooses between computation paths
and preference is given to a path that terminates (in an accept
state). This view is closely connected with the amb operator of
/McC67/. In programming languages nondeterminism also chooses between
computation paths (or values in applicative languages) but all paths
are treated equal. This is the notion to be considered here. It is
explained in /Egl75/ as: "We do not want to think of trying all
possible paths in a program. We are computing along a single path
but we might not know which one. This happens, for instance, if
the choice of the path is done according to certain unknown parameters
in the system which we consider as being nondeterministic choices."
We shall use the term oracles for these parameters.

In this section we will show that nondeterminism can be described
as a certain abstract interpretation, i.e. as a certain data flow
analysis. Such a development has not been performed before and it
seems to be essential to use the "strong" approach of this chapter.
Furthermore, it will be necessary to extend the framework with the
notions of partly collecting and partly induced semantics.

THE EXAMPLE LANGUAGE AND ITS SEMANTICS

The development will be performed for the following language of
commands (cmd\&cmd):
Here act stands for unspecified atomic actions and exp stands for unspecified (boolean) expressions; we shall assume these represent deterministic and total recursive computations. Nondeterminism is introduced by the use of cmd₁ or cmd₂.

The main idea is to view cmd₁ or cmd₂ as a shorthand for the following deterministic program

\[
\text{if } \text{hd}(o) \text{ then } o := \text{tl}(o) \; ; \; \text{cmd} \quad \text{else } o := \text{tl}(o) \; ; \; \text{cmd}_2
\]

where o is an oracle. One might argue that hd should also have program variables as parameters but we shall not consider this more complicated setting. When the oracle is not known this gives rise to nondeterministic behaviour as in the above quotation from /Egl75/.

Therefore denotational semantics (as P below) use powerdomains to collect the sets of possible outcomes. The result of cmd₁ or cmd₂ is then the union of the results along either branch. This is quite analogous to what happens in the collecting (or another approximating) semantics if too little is known about the set of states so that the conditional may evaluate to both true and false. In the remainder of this section this idea will be precisely formulated using abstract interpretation.

We first define a denotational semantics P for the nondeterministic language. Let S be an unspecified nonempty set of states. The meaning of atomic actions and (boolean) expressions are given by unspecified total functions

\[
\mathcal{L}(\text{act}) : S \to S \\
\mathcal{B}(\text{exp}) : S \to \{\text{true, false}\}
\]
We shall extend these strictly to \( S_\downarrow \rightarrow S_\downarrow \) and \( S_\downarrow \rightarrow \{\text{true}, \text{false}\}_\downarrow \) respectively. The nondeterministic semantics

\[
P[\text{cmd}] : S_\downarrow \rightarrow P(S_\downarrow)
\]

is then defined as follows (using notation from section 5.2):

\[
P[\text{act}] = \{\forall \text{act}\}
\]

\[
P[\text{cmd}_1; \text{cmd}_2] = P[\text{cmd}_2] \sqcup P[\text{cmd}_1]
\]

where \( f_2 \sqcup f_1 = f_2 \sqcup f_1 \)

\[
P[\text{if exp then cmd}_1 \text{ else cmd}_2] = \text{cond}^P_{\text{exp}}(P[\text{cmd}_1], P[\text{cmd}_2])
\]

where \( \text{cond}^P_{\text{exp}}(f_1, f_2)(s) = P[\text{exp}](s) \rightarrow f_1(s), f_2(s) \)

\[
P[\text{cmd}_1 \text{ or cmd}_2] = P[\text{cmd}_1] \sqcup P[\text{cmd}_2]
\]

where \( (f_1 \sqcup f_2)(s) = f_1(s) \cup f_2(s) \)

\[
P[\text{while exp do cmd}] = \text{LFP}(\lambda f. \text{cond}^P_{\text{exp}}(f \sqcup P[\text{cmd}], \{\})
\]

This defines a strict monotonic function of functionality as stated. Because \( S_\downarrow \) is of finite height it is (directed) continuous as well. Usually the Plotkin powerdomain is used instead of \( P(S_\downarrow) \) but this does not affect the sets produced by the semantics. (As in /Plo76/ an argument using König's lemma shows that \( P[\text{cmd}](s) \) is either finite or contains \( \bot \).)

To obtain the above semantics as an approximating semantics we must start with a deterministic standard semantics. It will use oracles modelled by infinite strings of 0's and 1's. (We shall discuss this choice further in the next section.) The idea then is to define a semantics

\[
S[\text{cmd}] : S_\downarrow \times O_\downarrow \rightarrow S_\downarrow \times O_\downarrow
\]

where \( O = \{0, 1\}^\omega \) is the set of countably infinite strings of 0's and 1's. When a "nondeterministic branch" is encountered the left branch
will be taken if the oracle begins with 0 and the right branch if the
oracle begins with 1. The branch chosen is then executed with the
remainder of the oracle. This in effect corresponds to viewing
\( \text{cmd}_1 \) or \( \text{cmd}_2 \) as a shorthand for the piece of program displayed earlier.
As for the functionality of \( S[\text{cmd}] \) it is possible to use \( S_1^X O_\downarrow \), but
\( S_1^XO_\downarrow \) is closer to "operational intuitions" and this is preferable
in the next section.

The following operations upon oracles \( O \) and finite strings
\( w \in \{0,1\}^* \) will be needed. By \( O^m \) and \( O^m_\downarrow \) are denoted the first \( m \)
elements of \( O \) and the remainder of \( O \) respectively. We shall extend
this notation to \( O_\downarrow \) by writing \( O^\infty = \bot = I^m_\infty \). The string \( w \) has
length \( |w| \) and \( wo \) is the oracle obtained by prefixing \( w \) to \( O \). The
standard semantics \( S \) is then defined by the following equations:

\[
\begin{align*}
S[\text{fact}] &= \lambda(s, o). (\lambda_\text{fact}(s), o) \\
S[\text{cmd}_1; \text{cmd}_2] &= S[\text{cmd}_2] S[\text{cmd}_1] \\
S[\text{if } \text{exp} \text{ then } \text{cmd}_1 \text{ else } \text{cmd}_2] &= \text{cond}_\text{exp} \left( S[\text{cmd}_1], S[\text{cmd}_2] \right) \\
\text{where } \text{cond}_\text{exp}(f_1, f_2)(s, o) &= S[\text{exp}](s) \to f_1(s, o), f_2(s, o) \\
S[\text{cmd}_1 \text{ or } \text{cmd}_2] &= S[\text{cmd}_1] S[\text{cmd}_2] \\
S[\text{cmd}_1 \text{ or } \text{cmd}_2] &= S[\text{cmd}_1] S[\text{cmd}_2] \\
S[\text{while } \text{exp} \text{ do } \text{cmd}_3] &= \text{LFP}(\lambda f. \text{cond}_\text{exp}(f, S, \text{cmd}_3, \lambda(s, o). (s, o)))
\end{align*}
\]

These equations define a strict function of functionality as stated.
For \( O \) is chosen such that \( O^\infty = \bot \) whenever \( O \in \) and since \( \lambda_\text{fact} \) is a
strict extension of a total function we know that \( S[\text{fact}] \) is of the
correct functionality. In fact \( S[\text{cmd}] \) is continuous and it is
straightforward to show that \( \text{LFP} \) is only needed of continuous
functionals. It is also easy to show that the semantics is "continuous"
in prefixes of oracles:
Fact 5.6.1: If $\text{S}_{\text{cmd}}(s,o) = (s',o') \neq (\bot,\bot)$ there exists a finite prefix $w$ of $o$ such that $o^n|w| = w$ implies that $\text{S}_{\text{cmd}}(s,o^n) = (s',o^n)|w|$.  

THE PARTLY COLLECTING SEMANTICS

The next task is to define a collecting semantics and show its relation to the standard semantics. One might consider

$$C_{\text{cmd}} : \mathcal{F}(S_1 \times O_1) \rightarrow \mathcal{E}(S_1 \times O_1)$$

as this follows the pattern used previously but it will be more appropriate to use

$$C_{\text{cmd}} : S_1 \times \mathcal{E}(O_1) \rightarrow \mathcal{E}(S_1 \times O)$$

which will be called the partly collecting semantics. This is because the idea with the nondeterministic semantics is that we have full information about the states and no information about the oracles. In other words data flow analysis is performed only for the oracles and not for the states. (It should be intuitively clear that $\mathcal{E}(S_1 \times O_1)$ in the range cannot be replaced by $S_1 \times \mathcal{E}(O_1)$ because oracles may influence the resulting states.)

To define $C$ we first consider the necessary modifications of theorem 5.2:5. Let $D$ be a dcpo and $B$ and $C$ augmented dcpos. A function $g : D \times B \rightarrow C$ is $\mathcal{E}$-monotonic in its right argument (ambiguously written $g : D \times B \rightarrow C$) iff for all $d \in D$ that $Y_1 \subseteq Y_2$ implies $g(d,Y_1) \subseteq g(d,Y_2)$. If the augmentations of $B$ and $C$ are complete the function is said to be completely linear in its right argument (written $g : D \times B \rightarrow C$) iff, for each $d \in D$ and nonempty subset $Y$ of $B$, the formula

$$g(d, \bigcup Y) = \bigcup \{ g(d, y) \mid y \in Y \}$$
holds. The function is smash-invariant \((g: D^B \rightarrow C)\) iff \(g = g' \cdot \text{smash}\).

Finally, let \(\tilde{f}: D \times E \rightarrow D \times F\) denote the application of \(f: E \rightarrow F\) upon second components, i.e. \(\tilde{f}(d,e) = (d,f(e))\) and let \(\text{inc}: D \times B \rightarrow D \times B\) be defined by \(\text{inc}(d,Y) = (d,Y)\).

**Lemma 5.6:** Let \(D\) be a dcpo, \(E\) a benign fdcpo and \(C\) a completely augmented dcpo. For each monotonic function \(f: D \times E \rightarrow C\) there exists precisely one monotonic and smash-invariant function \(f': D \times ^E_2(E) \rightarrow C\) that is completely linear in its right argument and fulfils that \(f': \text{inc} = f\). It is given by the formula

\[
f'(d,Y) = \bigcup \{ f(\text{smash}(d,Y)) \mid y \in Y \}
\]

and gives a strict function iff \(f\) is strict. Further \(\lambda \cdot f'\) is a monotonic and \(\mathcal{S}\)-monotonic bijection from \(D \times E \rightarrow C\) to \(D \times ^E_2(E) \rightarrow C\) whose inverse \(\lambda \cdot g' \cdot \text{inc} = \text{inc} = f\). It is given by the formula

\[
f'(d,Y) = \bigcup \{ f(\text{smash}(d,Y)) \mid y \in Y \}
\]

\[
\text{Proof The proof is similar to that of theorem 5.2:5 so the details are omitted.}
\]

The main use of this lemma is to "lift" a function \(f: D \times E \rightarrow F\) to a function \(\tilde{f}: D \times ^E_2(E) \rightarrow ^F_2(F)\) akin to what was done when extending \(\mathcal{G}\) to a functor. For this let \(D\) be a dcpo and \(E\) and \(F\) benign fdcpo's.

Define

\[
\hat{f} = (\{\} \cdot f)'^\mathcal{S} = \lambda(d,Y). \text{CON}(\{f(\text{smash}(d,Y)) \mid y \in Y\})
\]

Clearly \(\hat{f}\) is monotonic, smash-invariant and completely linear (hence \(\mathcal{S}\)-monotonic) in its right argument and it is strict iff \(f\) is. We shall also need to define \(\tilde{g}\) akin to \(\text{LIN}(g)\) defined previously. For this let \(C\) be a completely augmented dcpo and \(g: D \times ^E_2(E) \rightarrow C\) a monotonic function and define \(\tilde{g}: D \times ^E_2(E) \rightarrow C\) by
\[ g = (g \cdot \{ \text{inc} \}^\#) = \lambda(d, y). \{ g(\text{smash}(d, y)) \mid y \in Y \} \]

It is immediate that \( g \) is smash-invariant and is completely linear (hence \( \preceq \)-monotonic) in its right argument. It is strict iff \( g \) is.

We are now ready to define the partly collecting semantics

\[ \overrightarrow{C[cmd]} : \text{S}_{\text{X}} \times \{ 0 \} \rightarrow \overrightarrow{\text{Z}(S \times 0_{\bot})} \]

The equations are:

\[ \overrightarrow{C[\text{act}]} = \mathcal{A}(s, 0). \overrightarrow{C[\text{act}]}(s, 0) \]

\[ \overrightarrow{C[cmd_1; cmd_2]} = \overrightarrow{C[cmd_2]} \circ \overrightarrow{C[cmd_1]} \]

where \( g_2 \circ g_1 = (g_2 \cdot \{ \text{inc} \}^\dagger) \cdot g_1 \)

\[ \overrightarrow{C[\text{if exp then cmd_1 else cmd_2}]} = \text{cond}_{\text{exp}}^C(\overrightarrow{C[cmd_1]}, \overrightarrow{C[cmd_2]}) \]

where \( \text{cond}_{\text{exp}}^C(g_1, g_2)(s, x) = \text{cond}_{\text{exp}}(s) \rightarrow g_1(s, x), g_2(s, x) \)

\[ \overrightarrow{C[\text{cmd_1 or cmd_2}]} = \overrightarrow{C[cmd_1]} \circ \overrightarrow{C[cmd_2]} \]

where \( (g_1 \circ g_2)(s, x) = \begin{cases} g_1(s, \{ 0 \downarrow 1 \mid \text{oEX} \wedge 0_{\bot} \downarrow 1 = 0 \}) \\ g_2(s, \{ 0 \downarrow 1 \mid \text{oEX} \wedge 0_{\bot} \downarrow 1 = 1 \}) \end{cases} \}

\{(1, \bot) \mid \bot \in X\}

with the convention that \( g_1(s, 0) = 0 \) and that \( 0_{\bot} = 0 \)

\[ \overrightarrow{C[\text{while exp do cmd}]} = \text{LFP}(\lambda g. \text{cond}_{\text{exp}}^C(g \circ \overrightarrow{C[cmd]}, \mathcal{A}(s, 0). (s, 0))) \]

It is straightforward to verify that this defines a strict and monotonic function that is \( \preceq \)-monotonic in its right argument. Because \( \preceq \)-monotonicity in the right argument is an admissible property the least upper bounds implicit in the definition of LFP are pointwise.

The connection with the standard semantics is given by:

**Theorem 5.6:3.** \( \overrightarrow{C[cmd]} = \overrightarrow{S[cmd]} = \lambda(s, x). \{ \overrightarrow{S[cmd]}(\text{smash}(s, 0)) \mid oEX \} \) //

**Proof** The proof follows the pattern used in proving lemma 5.3:2 and theorem 5.3:3.
Define the predicate $P(f, g)$ by $g'^{\sim} \cdot \text{inc} = \{f\}$. We first show by structural induction on commands $\text{cmd}$ that $P(S[\text{cmd}], C[\text{cmd}])$ holds. The result is immediate for atomic actions. Suppose next it holds for $\text{cmd}_1$ and $\text{cmd}_2$. For $\text{cmd} = \text{cmd}_1 ; \text{cmd}_2$ calculate

$$C[\text{cmd}] \cdot \{f\} \cdot \text{inc} =$$

(using the definition of $C[\text{cmd}]$ and the induction hypotheses)

$$(\{ \} \cdot S[\text{cmd}_2]) \cdot \{f\} \cdot C[\text{cmd}] =$$

(using theorem 5.2:5 and the definition of $S[\text{cmd}]$)

$$\{f\} \cdot S[\text{cmd}]$$

For $\text{cmd} = \text{if} \ \text{exp} \ \text{then} \ \text{cmd}_1 \ \text{else} \ \text{cmd}_2$ it easily follows by doing case analysis upon $S[\text{exp}](s)$ that

$$C[\text{cmd}] \cdot \{\text{inc}(s, o)\} = S[\text{cmd}](s, o)$$

holds for all $(s, o) \in S_1 \times O_1$. For $\text{cmd} = \text{cmd}_1$ or $\text{cmd}_2$ a similar result follows by doing case analysis upon $S_1$. Finally consider

$\text{cmd} = \text{while} \ \text{exp} \ \text{do} \ \text{cmd}_1$ and let $F$ and $G$ be the functionals of which the least fixed points are required (for $S$ and $C$ respectively). Reasoning as above it is straightforward to show that

$$P(f, g) \Rightarrow P(F(f), G(g))$$

The result then follows by the induction principle once it is shown that $P$ is admissible. It is immediate that $P(\{, \})$ holds and for the other condition note that $\{\}$ is directed continuous and the least upper bounds (of directed set) in $S_1 \times \mathcal{L}(O_1) = \mathcal{L}(S_1 \times O_1)$ are pointwise.

To obtain the result of the theorem it suffices to show that $C[\text{cmd}]$ is completely linear in its right argument. This is shown by a straightforward structural induction. Taking $\text{while} \ \text{exp} \ \text{do} \ \text{cmd}$ as an example it is immediate that $G(g)$ is always completely linear.
in its right argument. Hence the result follows for \( \text{LFP}(G) = G(\text{LFP}(G)) \).

---

**Corollary 5.6:** For \( G \) as defined in the above proof we have

\[
\text{LFP}(G) = \bigcup_{n=0}^\infty G^n(\bot)
\]

---

**Proof** Using \( F \) defined above we calculate

\[
\text{LFP}(G) = \text{LFP}(F) = \\
\]

\( (F \text{ and } I) \text{ are continuous} \)

\[
\left( \bigcup_n \{ \bigcap_n I^n(\bot) \} \right)^\# = \\
(\text{a straightforward numerical induction})
\]

\[
\left( \bigcup_n G^n(\bot) \cdot \bigcap_n \text{inc} \right)^\# = \\
(\xi \text{-monotonicity is admissible so the least upper bound is pointwise})
\]

\[
\left( \bigcup_n G^n(\bot) \cdot \bigcap_n \text{inc} \right)^\# = \\
\bigcup_n G^n(\bot)
\]

as was to be shown.

---

**THE PARTLY INDUCED SEMANTICS**

To obtain the nondeterministic semantics the idea is to replace \( \xi(\bot) \) and \( \xi(S \bot \bot) \) by less informative augmented dcpo's \( B \) and \( C \) respectively. This amounts to defining a partly induced semantics

\[
\tilde{\mathcal{L}}^{\text{cmd}} : S \times B \to C
\]

The connection between these augmented dcpo's is given by the following pairs of abstraction and concretization functions

\[
\text{(abs: } \xi(\bot) \to B, \text{ con: } B \to \xi(\bot) )
\]

\[
\text{(abs': } \xi(S \bot \bot) \to C, \text{ con': } C \to \xi(S \bot \bot) )
\]

We shall assume that all these functions are monotonic and \( \xi \)-monotonic.
(but adjoined-ness will fail as will be seen). The partly induced
semantics is then defined by the following equations:

\[
\begin{align*}
\mathbb{I} [\text{fact}] &= \text{abs'} \cdot \lambda (s, o). (\mathbb{I} [\text{fact}](s), o) \cdot \text{con} \\
\mathbb{I} [\text{cmd}_1 \circ \text{cmd}_2] &= \mathbb{I} [\text{cmd}_1] \circ \mathbb{I} [\text{cmd}_2] \\
\text{where } h_2 &= \text{abs'} \cdot ((\text{con'} h_2 \cdot \text{abs}) \cdot \text{con} \\
\mathbb{I} [\text{if } \exp \text{ then } \text{cmd}_1 \text{ else } \text{cmd}_2] &= \text{cond}^I \exp (\mathbb{I} [\text{cmd}_1], \mathbb{I} [\text{cmd}_2]) \\
\text{where } \text{cond}^I \exp (h_1, h_2) &= \text{abs'} \cdot \text{cond}^C \exp (\text{con'} h_1 \cdot \text{abs}, \\
&\phantom{= \text{abs'} \cdot \text{cond}^C \exp} \text{con'} h_2 \cdot \text{abs}) \\
\mathbb{I} [\text{cmd}_1 \text{ or } \text{cmd}_2] &= \mathbb{I} [\text{cmd}_1] \text{ or } \mathbb{I} [\text{cmd}_2] \\
\text{where } h_1 \text{ or } h_2 &= \text{abs'} \cdot ((\text{con'} h_1 \cdot \text{abs}) \text{ or } (\text{con'} h_2 \cdot \text{abs})) \\
\mathbb{I} [\text{while } \exp \text{ do } \text{cmd}] &= \text{LFP} (h, \\
\text{cond}^I \exp (h \cdot \mathbb{I} [\text{cmd}]), \text{abs'} \cdot \lambda (s, o). (s, o) \cdot \text{con})
\end{align*}
\]

We shall assume that the augmentation of \( C \) is admissible. Then
\( \varepsilon \)-monotonicity in the right argument is an admissible property and
therefore \( S , X , B \rightharpoonup C \) is a dcpo. It is then straightforward to verify
that the equations define a monotonic function that is \( \varepsilon \)-monotonic
in its right argument.

To obtain the nondeterministic semantics we define \( B = (\bot, \varepsilon, \varepsilon) \)
with \( \text{abs}(X) = \bot \) and \( \text{con}(\bot) = 0 \) as well as \( C = \varepsilon (S_\bot) \) with

\[
\begin{align*}
\text{abs'}(Z) &= \{ s \mid \exists o : (s, o) \in Z \} \\
\text{con'}(Y) &= \{ \text{smash}(s, o) \mid s \in Y \land o \in O \}
\end{align*}
\]

We shall let \( \mathbb{I} [\text{cmd}] \) refer to the partly induced semantics when \( B 
\) etc. are as above. Because \( 0 \) is not empty it is immediate to verify
that the functions are monotonic and \( \varepsilon \)-monotonic. In fact \( (\text{abs'}, \text{con'}) \)
is a pair of exactly adjoined functions so \( \text{abs'} \) is completely linear.
But \( (\text{abs}, \text{con}) \) is not a pair of adjoined functions because although
\( \text{abs'con} = \lambda 1.1 \) the other condition fails.
Remark One way to view the failure of adjoined-ness is that the choice of $B$ accomplishes two goals rather than one. The first goal is to replace $\varphi (0_L)$ by a less informative structure and

$$\{0,0_L\}, E_{EM}, C$$

might be a suitable choice. The second goal is to remove unneeded elements (i.e., $0_L$) from $B$ in order to come close to the nondeterministic semantics. It therefore might be considered to accomplish these two goals in two separate steps but this is not needed to demonstrate the connection between nondeterminism and abstract interpretation. ///

Clearly $S_B$ and $B$ are isomorphic so we shall subsequently identify them. We then have:

**Theorem 5.6.5.** $I[\text{cmd}] = P[\text{cmd}]$ for all commands cmd. ///

**Proof** The proof is by structural induction upon cmd.

**Case** $\text{cmd} = \text{act}$. We have

$$I[\text{act}](s) =$$

$$\text{abs'}(\{\text{act}(s) \circ_0 0\}) =$$

$$\{\text{act}(s)\} =$$

$$P[\text{act}](s)$$

**Case** $\text{cmd} = \text{cmd}_1 ; \text{cmd}_2$. We have

$$I[\text{cmd}] =$$

$$P[\text{cmd}_2] \circ^\top P[\text{cmd}_1] =$$

$$\text{abs'}(\text{con} \cdot P[\text{cmd}_2] \cdot \text{abs'} \cdot \text{inc} \cdot \text{con} \cdot P[\text{cmd}_1] \cdot \text{abs'} \cdot \text{con} =$$

(by $\text{abs'}$ completely linear and $\text{abs'} \cdot \text{con}$ and $\text{abs'} \cdot \text{con}$ the identities)

$$(P[\text{cmd}_2] \cdot \text{abs'} \cdot \text{inc} \cdot \text{con} \cdot P[\text{cmd}_1] =$$

(by $\text{h} \cdot \text{abs'} \cdot \text{inc} = \text{h} \cdot \text{abs'}$ as is straightforward to show)

$$P[\text{cmd}_2] \cdot \text{abs'} \cdot \text{con} \cdot P[\text{cmd}_1] = P[\text{cmd}]$$

280
Case \( \text{cmd} = \text{if } \text{exp} \text{ then } \text{cmd}_1 \text{ else } \text{cmd}_2 \). We have

\[
I_{\text{cmd}} = \\text{abs' 'cond}_C^{\text{exp}} (\text{con'} P_{\text{cmd}_1} \widetilde{\text{abs}}, \ldots) \widetilde{\text{con}} = \\
\text{(by abs' strict)} \quad \text{cond}_C^{\text{exp}} (\text{abs' 'con'} P_{\text{cmd}_1} \widetilde{\text{abs}} \text{ 'con}, \ldots) = \quad \text{cond}_C^{\text{exp}} (P_{\text{cmd}_1}, P_{\text{cmd}_2}) = \\
I_{\text{cmd}}
\]

Case \( \text{cmd} = \text{cmd}_1 \text{ or } \text{cmd}_2 \). We have

\[
I_{\text{cmd}} = \\text{abs' '}(\text{con'} P_{\text{cmd}_1} \widetilde{\text{abs}}) \text{ or } C \ldots \text{ 'con} = \\
\text{(using the convention described when defining or}^C \text{ and the complete linearity of abs')} \quad \lambda s. (\text{abs' 'con'} P_{\text{cmd}_1} \widetilde{\text{abs}})(s, \{o^1 | o \in 0 \land o^1 = 0\}) \cup \ldots \cup \{4 | l \in 0\} = \\
\text{(because there are } o', o'' \in 0 \text{ such that } o^1 = 0 \text{ and } o''^1 = 1 \text{ and because } l \in 0) \quad \lambda s. F_{\text{cmd}_1}(s) \cup F_{\text{cmd}_2}(s) = \\
I_{\text{cmd}}
\]

Case \( \text{cmd} = \text{while } \text{exp} \text{ do } \text{cmd}_1 \). Let \( F_{\text{cmd}} = \text{LFP}(F) \) and \( I_{\text{cmd}} = \text{LFP}(H) \). Reasoning as above it follows that \( F(f) = H(h) \) whenever \( f = h \). The result then follows by the induction principle of section 5.1 because equality is an admissible predicate.

The following theorem establishes the final result needed to verify the claim that the nondeterministic semantics essentially is the result of performing an abstract interpretation upon the deterministic standard semantics. So far it has not been essential that \( 0 = \{0, 1\}^\omega \) and the development could have been performed for any subset \( 0 \) of \( \{0, 1\}^\omega \) such that \( o^1 \) is an element of \( 0 \) whenever \( o \) is and \( 0 \) contains
oracles of the form $0o'$ and $1o"$. In the proof below more properties of $0$ will be used.

**Theorem 5.6.6.** $I_{\{cmd\}}^o = abs'(C_{\{cmd\}})^c$ holds for all commands $cmd$.

**Proof** We begin with defining the following predicate $P$ that is central to the proof. It is defined upon

$$(S, x) \rightarrow (S, x \cdot 0, \cdot 1)$$

and $P(g, h)$ is defined to hold iff all of

(a) $h = abs'g\cdot con$

(b) $g : S \times (S, x \cdot 0, \cdot 1) \rightarrow (S, x \cdot 0, \cdot 1)$

(c) for all $s \in S$: $g(s, 0) = \{\text{smash}(s', 0) | o \in O \exists o'(s', o') \in g(s, 0)\}$

hold. Condition (a) is the desired result but (b) and (c) will be needed to prove it. When $g = C_{\{cmd\}}$ it is immediate from theorem 5.6:3 and the observations after lemma 5.6:2 that condition (b) holds. We shall see that (c) holds as well. It may be phrased as saying that whenever one oracle is possible then all are. An equivalent formulation of (c) is that the range of $g\cdot con$ is a subset of the range of $con$.

The proof proceeds by structural induction on $cmd$ and shows

$P(C_{\{cmd\}}, I_{\{cmd\}})$

Case $cmd = act$ is straightforward.

Case $cmd = cmd_1; cmd_2$. This amounts to assuming that $P(g_1, h_1)$ and $P(g_2, h_2)$ hold and then show that $P(g_2 C_{g_1}, h_2 \cdot h_1)$ holds. For (a) calculate

$h_2 \cdot h_1 =$

282
(as in the proof of theorem 5.6.5)
\[ h_2^\dagger h_1 = (\text{abs}' g_2 \cdot \bowtie )^\dagger \cdot \text{abs}' g_1 \cdot \bowtie = \]

(abs' completely linear and \( h^\dagger \cdot \text{abs}' = (h \cdot \text{abs}' \cdot \bowtie )^\dagger ) \]
\[ \text{abs}' (g_2 \cdot \bowtie \cdot \text{abs}' \cdot \bowtie )^\dagger \cdot g_1 \cdot \bowtie = \]

(to be argued below)
\[ \text{abs}' (g_2 \cdot \bowtie )^\dagger \cdot g_1 \cdot \bowtie = \]
\[ \text{abs}' (g_2 \cdot \bowtie )^\dagger \cdot g_1 \cdot \bowtie = \]

For the missing step it suffices to show that
\[ (g_2 \cdot \bowtie \cdot \text{abs}' \cdot \bowtie )^\dagger \cdot \text{con}' = (g_2 \cdot \bowtie )^\dagger \cdot \text{con}' \]
because \( g_1 \) satisfies condition (c), i.e. \( g_1 (\bowtie (s, \lambda)) \) equals some \( \text{con}'(Y) \). Abbreviate this equation to \( \text{LHS} = \text{RHS} \). By complete linearity it suffices to consider singletons. We have
\[ \text{LHS}\{s\} = g_2(s, o) \]
\[ \text{RHS}\{s\} = \bigcup\{g_2(s, o) \mid o \in o\} \]
and these are equal because \( g_2 \) satisfies (b).

It is straightforward to verify condition (b). For condition (c) it follows from the calculations above that
\[ (g_2 \cdot \bowtie \cdot g_1) (\bowtie (s, \lambda)) = \]
\[ \text{LHS}(Y) = (\text{for some } Y) \]
\[ \bigcup\{g_2 \cdot \bowtie (s, \lambda) \mid s \in Y\} = \]
\[ \bigcup\{\text{con}'(Y) \mid s \in Y\} = (\text{for some } Y) \]
\[ \text{con}'(\bigcup\{Y \mid s \in Y\}) \]
This shows condition (c).

Case \( \text{cmd} = \text{if exp then cmd_1 else cmd_2} \). This amounts to assuming that
\( P(g_1, h_1) \) and \( P(g_2, h_2) \) hold and then show that
\( P(\text{cond}_{\exp}^c(g_1, g_2), \text{cond}_{\exp}^i(h_1, h_2)) \) holds. The calculations for (a) are
\[
\text{cond}_{\exp}^i(h_1', h_2') =
\]
(as in the proof of theorem 5.6:5)
\[
\lambda(s, \lambda). B(s) \Rightarrow h_1(s, \lambda), h_2(s, \lambda) =
\]
\[
\text{abs}''((\lambda(s, X). B(s) \Rightarrow \text{cond}_{\exp}^c(g_1(s, X), g_2(s, X))). \text{con} =
\]
\[
\text{abs}''(g_1' \text{con}) (s, \lambda) \lor (g_2' \text{con}) (s, \lambda) =
\]
(because \( \{0 \downarrow 1 \mid 0 \circ j = 1 \} \) equals 0 for \( j = 0 \) and \( j = 1 \))
\[
\text{abs}''(g_1 \lor g_2) '. \text{con}
\]
Condition (b) is straightforward to verify. For condition (c) it follows from above that \( (g_1 \lor g_2) '. \text{con} \) equals
\[
\lambda(s, \lambda). (g_1' \text{con}) (s, \lambda) \lor (g_2' \text{con}) (s, \lambda)
\]
and by the assumptions about \( g_1 \) condition (c) easily follows.

**Case cmd = cmd \_1 or cmd \_2**. This amounts to assuming that \( P(g_1, h_1) \) and \( P(g_2, h_2) \) hold and show that \( P(g_1 \lor g_2, h_1 \lor h_2) \) holds. For (a) we calculate
\[
h_1 \lor h_2 =
\]
(as in the proof of theorem 5.6:5)
\[
\lambda(s, \lambda). (\text{abs}''(g_1' \text{con}) (s, \lambda) \lor (\text{abs}''(g_2' \text{con}) (s, \lambda) =
\]
\[
\text{abs}''((\lambda(s, X). g_1(s, X) \lor g_2(s, X)). \text{con} =
\]
(because \( \{0 \downarrow 1 \mid 0 \circ j = 1 \} \) equals 0 for \( j = 0 \) and \( j = 1 \))
\[
\text{abs}''(g_1 \lor g_2) '. \text{con}
\]
Condition (b) is straightforward to verify. For condition (c) it follows from above that \( (g_1 \lor g_2) '. \text{con} \) equals
\[
\lambda(s, \lambda). (g_1' \text{con}) (s, \lambda) \lor (g_2' \text{con}) (s, \lambda)
\]
and by the assumptions about \( g_1 \) condition (c) easily follows.

**Case cmd = while \_ \exp \_ \do \_ cmd \_1**. Let \( F, G \) and \( H \) be the functionals such that \( \llbracket \text{cmd} \rrbracket = \text{LFP}(F), \llbracket \text{cmd} \rrbracket = \text{LFP}(G) \) and \( \llbracket \text{cmd} \rrbracket = \text{LFP}(H) \). It follows from the cases above that
\( P(g, h) \) implies \( P(G(g), H(h)) \)  

(1)

We first show that \( P(\text{LFP}(G), \text{LFP}(H)) \) follows from

\[
\bigcup_{n=0}^{\infty} G^n(\lambda) \cdot \con = \con \cdot \bigcup_{n=0}^{\infty} H^n(\lambda)
\]

(2)

which we shall abbreviate to \( \text{LHS} = \text{RHS} \). First, condition (a) of

\[
P(\bigcup_n G^n(\lambda), \bigcup_n H^n(\lambda))
\]

follows because \( \text{abs}' \cdot \text{con}' \) is the identity. Condition (b) is immediate by the properties of \( \lambda g, \bar{g} \) and condition (c) is immediate. Next \( P(\text{LFP}(G), \bigcup_n H^n(\lambda)) \) follows because of corollary 5.6.4. Using (1) this shows that

\[
\text{H}(\bigcup_n H^n(\lambda)) = \text{abs}' \cdot \text{LFP}(G) \cdot \con = \bigcup_n H^n(\lambda)
\]

and therefore \( \bigcup_n H^n(\lambda) \) equals \( \text{LFP}(H) \). Hence \( P(\text{LFP}(G), \text{LFP}(H)) \) follows from (2).

To establish (2) we first rewrite

\[
\text{RHS} = \bigcup_n \text{con}' H^n(\lambda)
\]

To prove this it suffices to show that \( \text{con}' \) is directed continuous. Clearly it is monotonic and completely linear. This implies continuity because \( S_L \) and \( S_L \cdot 0_L \) are flat. To see this let \( \gamma \) be a directed subset of \( \bar{\gamma}(S_L) \). If all \( \gamma \in \gamma \) contain \( L \) then all \( \text{con}'(\gamma) \) contain \( (\lambda, \lambda) \) and

\[
\text{con}'(\bigcup \gamma) = \text{con}'(\bigcup \gamma) = \bigcup \{ \text{con}'(\gamma) \mid \gamma \in \gamma \} = \bigcup \{ \text{con}'(\gamma) \mid \gamma \in \gamma \}
\]

Otherwise \( \bigcup \gamma \) is an element of \( \gamma \) and continuity follows by monotonicity.

Both \( \text{LHS} \) and \( \text{RHS} \) are strict functions so it suffices to fix \( s \in S \)

and show \( \text{LHS}(s, L) = \text{RHS}(s, L) \). We have, using the connection between \( F \) and \( G \), that
\[ \text{LHS}(s, L) = \bigcup \{ \bigcup_n F^n(L) (s, o) \mid o \in O \} = \bigcup_n F^n(L) (s, o) \mid o \in O \] 

Since (1) easily gives \( P(G^n(L), M^n(L)) \) we have

\[ \text{RHS}(s, L) = \bigcup_n G^n(L) (s, O) = \bigcup_n \{ G^n(L) (s, o) \mid o \in O \} = \bigcup_n F^n(L) (s, o) \mid o \in O \] 

It is straightforward to show that LHS(s, L) and RHS(s, L) contain the same maximal elements (i.e., elements that are not \((L, L))\). It is also straightforward to show that if LHS(s, L) contains \((L, L)\) then so does RHS(s, L).

Next assume that RHS(s, L) contains \((L, L)\) and show that LHS(s, L) also does. From the assumption it follows that for each \(n\) there is \(o \in O\) such that \(G^n(L) (s, o_n) = (L, L)\). We must find \(o \in O\) such that

\[ \bigcup_m F^m(L) (s, o) = (L, L) \] 

If no \(o_n\) can be used for \(o\) we construct \(o\) as follows.

For each \(n\) there is a minimal number \(m_n\) such that \(F^m_n(L) (s, o_n) \neq (L, L)\). It follows that \(m_n > n\) and it is not hard to show that this implies that

\[ F^n(f) (s, o_n) = f(S_{\text{cmd}_L} D^n(s, o_n)) \] 

Since \(f = F^{m-n}_n(L)\) is strict it follows that \(S_{\text{cmd}_L} D^n(s, o_n) \neq (L, L)\). By \(n\) applications of fact 5.6:1 this gives \(k_n\) such that

\[ o_n^{k_n} = o_n^{f_{k_n}} \] 

implies \( F^n(L)(s, o) = L \)

The set \( \{ o_n^{f_{k_n}} \mid n \geq 0 \} \) must be infinite. For if it was finite there would be some \(n\) such that \(o_j^{f_{k_j}} = o_n^{f_{k_n}}\) holds for infinitely many \(j\).
Then
\[ F^j(\lambda)(s, o_n) = (\lambda, \lambda) \]
would hold for infinitely many \( j \) and this contradicts the assumption that no \( o_n \) could be used for \( o \). One may view the set \( \{ o_n \mid n \geq 0 \} \) as a tree ordered by prefix. It is an infinite but finitely branching tree. By König's lemma there must exist an infinite path \( o \in \{0, 1\}^\omega \).

By choice of \( o \) we have \( o \in o \). Since \( o^k_m = o^k_m \) holds for infinitely many \( m \) we have that \( \bigcup_m F^m(\lambda)(s, o) = (\lambda, \lambda) \).

/ / /

5.7 OTHER NOTIONS OF NONDETERMINISM

In the previous section it was assumed that all countably infinite strings of 0's and 1's were possible oracles. From a data flow analysis point of view it is natural to investigate what happens if only a certain set of strings is possible. For example if "the choice of the path is done according to certain unknown parameters in the system"/Eg175/ one might want to consider only "computable" strings. Similarly fairness considerations may restrict attention to only "fair" strings. This section investigates consequences of letting the set 0 of oracles be a subset of \( \{0, 1\}^\omega \).

First we consider what properties 0 is to fulfil in order that the results of the previous section still hold and a partial answer is given. Next there is the problem of how to formulate a nondeterministic semantics, i.e. a variant of \( P \), such that it corresponds to the effect of some chosen set 0 of oracles. It turns out that for some choices of 0 there is no modification of \( P \) (in a certain class of possibilities) that yields the desired semantics. This class contains the partly induced semantics and therefore abstract inter-
pretation (as developed in section 5.6) is of no help in this. It is therefore studied how to obtain a modification \( A \) of \( P \) such that the desired semantics will be "between" \( P \) and \( A \): it converges if \( P \) says so and diverges if \( A \) says so. In particular, whereas \( P \) specifies divergence if the program may diverge \( A \) only does so if the program may have to diverge. For this reason \( A \) is called the "angelic semantics".

**On Replacing \( \{0,1\}^\omega \) by an "Equivalent" Subset**

Let \( \mathcal{O} \) be a subset of \( \{0,1\}^\omega \). Throughout this section we shall assume that \( \mathcal{O} \) satisfies the following acceptability conditions:

\[
0 \neq 0 \\
0 = \{0^{n+1} \mid 0 \in \mathcal{O} \land 1 \in \mathcal{O} \}
\]

\[
0 = \{0^{n+1} \mid 0 \in \mathcal{O} \land 1 \in \mathcal{O} \}
\]

The last two conditions may be rephrased: if \( o \in \mathcal{O} \) then all of \( 0^n \), \( 0o \) and \( 1o \) are elements of \( \mathcal{O} \). This is probably a rather natural condition to assume and we shall investigate some consequences shortly. The nondeterministic semantics corresponding to using \( \mathcal{O} \) as the set of oracles is

\[
\mathcal{S}_\mathcal{O} \rightarrow \mathcal{E}(\mathcal{S}_1)
\]

It is defined by

\[
\mathcal{S}_\mathcal{O} \rightarrow \mathcal{E}(\mathcal{S}_1) = \lambda s. \{s' \in \mathcal{S}_1 \mid \exists o, o': \mathcal{S}[\text{cmd}](\text{smash}(s, o)) = \text{smash}(s', o')\}
\]

but this is not a denotational definition in the sense of Scott and Strachey because the definition is not compositional. Clearly \( \mathcal{S}_\mathcal{O} \rightarrow \mathcal{E}(\mathcal{S}_1) \) is of the correct functionality when \( \mathcal{O} \) is acceptable because then \( \mathcal{S}[\text{cmd}] \) specializes to the functionality \( \mathcal{S}_1 \rightarrow \mathcal{S}_1 \rightarrow \mathcal{O} \). The set
O = \{0,1\}^w\ used in the previous section is acceptable and, using the results in section 5.6, it is immediate to see that $D_{\{0,1\}^w}$ equals $P$.

The motivation behind assuming $O$ to be acceptable is primarily to obtain the following two lemmas. A remark below shows that this cannot be achieved under weaker assumptions. The first lemma states that $P$ and $D_O$ agree if divergence (i.e., $\perp$ of $S_\perp$) is ignored.

**Lemma 5.7: 1.** If $O$ is acceptable then

$$D_O[cmd](s) - \{1\} = P[cmd](s) - \{1\}$$

holds for all commands $cmd$ and $s \in S_\perp$.

**Proof** Since $P$ is $D_{\{0,1\}^w}$ the inclusion $\subseteq$ is immediate. The other inclusion follows from fact 5.6: 1 and acceptability of $O$. For if $1/s' \notin P[cmd](s)$ there is $w$ such that $S[cmd](s,w) = (s',o)$ for all $o \in [0,1]^w$ and hence some $o \in O$. But $w \in O$ and therefore $s' \in D_O[cmd](s)$.

Acceptability also guarantees that "whenever one oracle is possible then all are" (as was considered in the proof of theorem 5.6:6).

**Lemma 5.7: 2.** If $O$ is acceptable then

$$\{\text{smash}(s',o') \mid \text{s'} \in D_O[cmd](s) \land o' \in O\} = G[cmd](s,o)$$

holds for all $cmd$ and $s \in S_\perp$.

**Proof** Clearly $(\perp,\perp)$ is an element of the lefthand side iff it is an element of the righthand side. Next let $(s',o') \neq (\perp,\perp)$ be an element of the lefthand side. By fact 5.6:1 there is $w$ such that $S[cmd](s,w) = (s',o')$. Since $w \in O$ it follows that $(s',o')$ is an element of the righthand side. Conversely let $(s',o') \neq (\perp,\perp)$
be an element of the righthand side. Then there is $o \in O$ such that $S[s][cmd](s, o) = (s', o')$. By fact 5.6:1 and acceptability of $O$ it follows that $o' \in O$. Hence $(s', o')$ is an element of the lefthand side.

Remark Under reasonable assumptions it is necessary to assume that $O$ is acceptable in order to obtain the previous lemma. First note that $O \neq \emptyset$ is required for $D_O$ to be of the correct functionality. Next assume that there are atomic actions $x := 0$ and $x := 1$ such that $\forall s \in S: \mathcal{O}[x := 0](s) \neq \mathcal{O}[x := 1](s)$

Since $S$ is non-empty we may choose $s \in S$. Suppose that $O$ is chosen such that the equality in the previous lemma holds. Using it with $s$ and $cmd = x := 0$ or $x := 1$

shows the remaining conditions in the definition of acceptability.

Large parts of the development in section 5.6 can be carried through with any acceptable set $O$ rather than only $O = \{0, 1\}^\omega$. In particular theorem 5.6:3 ($C = S$) and theorem 5.6:5 ($I = F$) will still hold. This implies that theorem 5.6:6 may be rephrased as

$P[s][cmd] = D_O[s][cmd]$

or equivalently as $D[\{0, 1\}^\omega][cmd] = D_O[s][cmd]$. This equality may fail under reasonable assumptions about the expressiveness of atomic actions and boolean expressions. As an example take

$cmd = x := 1$ ; while $x \neq 0$ do ( $x := x + 1$ or $x := x - 1$ )

$O = \{w110110110 \ldots \mid w \in \{0, 1\}^\omega \}$

It follows from the previous lemma that the only difference possible
is that \( D_0(\text{cmd}\{s\}) = P(\text{cmd}\{s\}) - \{1\} \), i.e. that \( P \) specifies divergence too often.

It is not clear what conditions to place on \( O \) such that \( P \) equals \( D_0 \). The use of \( O = \{0,1\}^\omega \) is not necessary however. To show this we shall assume that \( S \) and the sets of atomic actions and boolean expressions are all countable. Then the set of pairs \((\text{cmd}, s)\) is countable and so is the set of pairs such that \( P(\text{cmd}\{s\}) = 1 \). Construct \( X \subseteq \{0,1\}^\omega \) by including for each such \((\text{cmd}, s)\) a string \( o \) such that \( S(\text{cmd}\{\text{smash}(s, o)\}) = (1,1) \). Clearly \( X \) is countable and therefore is countable and acceptable. Since \( O \) contains \( X \) it is immediate that \( P \) equals \( D_0 \).

If "the choice of the path is done according to certain unknown parameters"/Egl75/ one might expect the choice to be made by some algorithm. In the present formulation this would imply that oracles should be "computable". The following lemma shows that if this view is taken then one cannot obtain \( P \). Define an oracle \( o \in \{0,1\}^\omega \) to be recursive iff

\[ \theta(o) = \{n \mid \text{the n'th element of } o \text{ is 1}\} \]

is a recursive set. An oracle is recursive iff there is a Turing Machine that lists all finite prefixes of \( o \). We shall assume that the atomic actions and boolean expressions are sufficiently expressive that the language of commands may simulate arbitrary Turing Machines.

**Lemma 5.7.3.** If \( O \) is an acceptable set such that (for all \( \text{cmd} \))

\[ P(\text{cmd}) = D_0(\text{cmd}) \]

then \( O \) must contain the recursive oracles as a proper subset. ///
Proof We first show that all recursive oracles are needed. Let $o$ be a recursive oracle. The concept of a Turing Machine with oracle $X$ (a subset of the integers) is explained in /Rog67/ and a similar concept in /HoU179/. Consider the following informal description of such a Turing Machine:

\[
\begin{align*}
n &:= 1 \\
\text{while } n > 0 \text{ do compute the } n\text{'th element of } o \text{ by simulating the Turing Machine for } o; \\
&\quad \text{ask the oracle whether } n \text{ is in it and let } a \text{ be 1 if it is and 0 if not;}
\end{align*}
\]

\[
\begin{align*}
\text{if } a \text{ differs from the } n\text{'th element then } n &:= 0 \\
\text{else } n &:= n + 1
\end{align*}
\]

By the assumptions about the expressiveness of the atomic actions and boolean expressions there is a program $cmd$ and initial state $s \in S$ such that $S[cmd](s, o') = (1, 1)$ iff the Turing Machine loops upon oracle $\Theta(o')$. For example the query of the oracle may be implemented by

\[
\begin{align*}
a &:= 0 \text{ or } a := 1
\end{align*}
\]

Clearly the Turing Machine loops iff it is supplied with the oracle $\Theta(o)$. Therefore $P[cmd](s) \exists! 1$ and

\[
P[cmd](s) = D[cmd](s)
\]

shows that $o \in \mathcal{O}$.

We next show that some $o \in \mathcal{O}$ must be non-recursive. Recall that there are recursively enumerable subsets $X$ and $Y$ of the integers such that $X \cap Y = \emptyset$ and such that no recursive set $R$ satisfies $X \subseteq R$ and $R \cap Y = \emptyset$ /Rog67/. Let $M_X$ and $M_Y$ be Turing Machines recognizing $X$ and $Y$ and write $X \not\exists^k j$ and $Y \not\exists^k j$ for whether these accept $j$ in at most
k steps. Let \( T: \mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N} \) be a recursive enumeration of the pairs of integers such that \((i, j) = T(n)\) implies \(i \in \mathbb{N}\) and \(j \in \mathbb{N}\). Consider the following description of a Turing Machine \( M \) with oracle:

set up an empty list; \( n := 1; \)

while \( n > 0 \)
do \((j, k) := T(n); \)

ask the oracle whether it contains \( n \) and append 1 to the list, if the answer is yes and otherwise append 0;

if \( X \exists^k j \) and the \( j \)'th element on the list is 0
or \( Y \exists^k j \) and the \( j \)'th element on the list is 1
then \( n := 0 \) else \( n := n + 1 \)

Then \( M \) will loop upon oracle \( X \) but halt on any recursive oracle \( R \).

As before there is a program \( \text{cmd} \) and initial state \( s \) such that

\[ S[\text{cmd}](s, o') = (1, 1) \text{ iff } M \text{ loops upon } O(o'). \]

Since \( P[\text{cmd}](s) \neq 1 \) it follows that 0 must contain some oracle that is not recursive. ///

It has already been shown that there is an acceptable set \( O \) such that \( D \subseteq O \) differs from \( P \). The previous lemma makes it feasible to define a whole sequence of such examples. Let \( (0^j 1)^\omega \) mean the infinite string consisting of \( j \) 0's and a 1 etc. and define

\[ O_i = \{ w(0^j 1)^\omega \mid w \in \{0, 1\}^\omega, \forall j \} \]

Note that each \( O_i \) is acceptable and is a proper subset of both \( O_{i+1} \) and the set of recursive oracles. It follows that for all \( \text{cmd} \) and \( s \):

\[ D_{0_{i}}(s, \text{cmd}) \neq D_{0_{i+1}}(s, \text{cmd}) \neq \ldots \]

and using the proof of the previous lemma it follows that for all \( i \) there are \( \text{cmd}_{i} \) and \( s_{i} \) such that
\[ D_{0_1}^{\text{cmd}_1}(s_i) \neq D_{0_{i+1}}^{\text{cmd}_1}(s_i) \]

Hence all \( D_{0_i} \) are different from one another. (Other examples of such a family \( \{0_i\} \) may be obtained by defining \( 0_i \) to be oracles of certain complexity classes.)

**ANGELIC SEMANTICS**

It follows from section 5.6 that the partly induced semantics \( I \) equals \( P \) even when \( O = \{0, 1\}^\omega \) is replaced by an arbitrary acceptable set \( O \). This means that abstract interpretation (as developed in section 5.6) cannot be used to obtain a (compositional) denotational definition of \( D_O \) when it is not \( P \). In the remainder of this section we shall therefore study \( D_O \) for its own sake. A first question one may ask is whether a compositional definition can exist at all. Hopes for such a definition are motivated by the following lemma but none the less we shall see that the answer is no for some acceptable sets \( O \).

**Lemma 5.7.4.** Let \( O \) be an acceptable set of oracles. Then

\[
\begin{align*}
D_{O}[\text{act}] &= \{1\}' \downarrow \text{act} \\
D_{O}[\text{cmd}_1; \text{cmd}_2] &= D_{O}[\text{cmd}_2] \uparrow P D_{O}[\text{cmd}_1] \\
D_{O}[\text{if exp then cmd}_1 \text{ else cmd}_2] &= \text{cond}^p_{\text{exp}}(D_{O}[\text{cmd}_1], D_{O}[\text{cmd}_2]) \\
D_{O}[\text{cmd}_1 \text{ or cmd}_2] &= D_{O}[\text{cmd}_1] \text{ or } D_{O}[\text{cmd}_2] \\
D_{O}[\text{while exp do cmd}_1] &\text{ is a fixed point of} \\
&\text{af. } \text{cond}^p_{\text{exp}}(f D_{O}[\text{cmd}_1], 1) \\
\end{align*}
\]

**Proof** The proof proceeds by structural induction.

**Case cmd = act** is straightforward.

**Case cmd = cmd\_1; cmd\_2** follows from calculating
\[ D_0[\text{cmd}](s) = \]

(because \( S[\text{cmd}, I] \) specializes to \( S_1 \cdot 0_1 \to S_1 \cdot 0_1 \))

\[ \{ s' \in S_1 \mid \exists s'' \in S_1: \exists o,o',o'' \in O: \]

\[ \text{smash}(s',o') = S[\text{cmd}, I](\text{smash}(s'',o'')) \}

\[ \text{smash}(s'',o'') = S[\text{cmd}, I](\text{smash}(s,o)) \}

(by lemma 5.7:2)

\[ U \{ D_0[\text{cmd}_2]((s'') \mid s'' \in D_0[\text{cmd}_1](s) \} = \]

\[ (D_0[\text{cmd}_2] \uparrow D_0[\text{cmd}_1])(s) \]

Case \( \text{cmd} = \text{if exp then cmd}_1 \text{ else cmd}_2 \) is straightforward.

Case \( \text{cmd} = \text{cmd}_1 \text{ or cmd}_2 \) is straightforward because \( O \) is acceptable.

Case \( \text{cmd} = \text{while exp do cmd}_1 \). The proof is easiest when assuming there is an atomic action \( \text{skip} \) such that \( \text{NN}(\text{skip}) = \lambda s.s. \). Then reasoning as above shows that

\[ \text{cond}^P_{\text{exp}}( f \uparrow D_0[\text{cmd}], \}) \]

equals

\[ D_0[\text{if exp then (cmd;while exp do cmd) else skip}] \]

when \( f = D_0[\text{while exp do cmd}] \). But this equals \( f \) because \( S[\text{while ... I} \) clearly equals \( S[\text{if ... I} \). - If there is no atomic action \( \text{skip} \) the result is only slightly more cumbersome. ///

This lemma might be taken to mean that the compositional definition of \( D_0 \) should be obtained from the equations defining \( P \) by replacing \( \text{LFP} \) by some other fixed point operator. It is not possible to do so for all acceptable choices of \( O \). As an example define

\[ \text{exp} = x \]

\[ \text{cmd}_1 = x := \text{false or skip} \]

\[ \text{cmd}_2 = \text{skip or x := false} \]
and

\[ \emptyset = \{\ldots 000 \mid w \in [0, 1]^\infty\} \]

It is immediate that \( D_0[\text{cmd}_1] \) equals \( D_0[\text{cmd}_2] \) and hence the two functionals (of which some fixed point is to be taken) are equal.

So \( D_0[\text{while exp do cmd}_1] \) would equal \( D_0[\text{while exp do cmd}_2] \) and this is clearly wrong. Hence \( D_0 \) cannot be defined along the lines suggested.

Remark It may well be argued that the example above is "unnatural" but it is not clear how to obtain any compositional definition of some \( D_0 \) that is not \( P \). In particular, the approaches of /Par81/ and /deR81/ of defining a fair denotational semantics does not follow the approach outlined above. (One might take \( \{0, 1\}^\omega \sim \{000, ..., 111, \ldots\} \) to be the set of "fair" oracles.) There \( \llbracket \text{cmd}_1 \lor \text{cmd}_2 \rrbracket \) essentially is \( (\llbracket \text{cmd}_1 \rrbracket, \llbracket \text{cmd}_2 \rrbracket) \) and the analogue of the semantics for while describes, using least and greatest fixed points, a fair interleaving of \( \llbracket \text{cmd}_1 \rrbracket \) and \( \llbracket \text{cmd}_2 \rrbracket \). However, in this approach \( \llbracket \text{cmd}_1 \lor \text{cmd}_2 \rrbracket \) differs from \( \llbracket \text{cmd}_2 \lor \text{cmd}_1 \rrbracket \) and this is contrary to the intuition leading to \( P \).

///

Since it is unclear how to define \( D_0 \) compositionally we shall define "an upper bound" \( A \) along the lines suggested. The idea is that \( P \), \( D_0 \) and \( A \) are equal when \( \perp \) of \( S_\perp \) is ignored and that if \( A \) specifies divergence then so does \( D_0 \) and \( P \). The main task is to define a fixed point operator \( \text{AFP} \). Define

\[ \text{AFP}(F) = \bigcup \{ f : S_\perp \rightarrow \xi(S_\perp) \mid F(f) = f \land f \in \text{LFP}(F) \} \]

for each monotonic and \( \xi \)-monotonic function \( F \) from \( S_\perp \rightarrow \xi(S_\perp) \) to itself. We shall see shortly that this definition makes sense. For
an informal motivation of the definition let

\[
f = D_0 \{ \text{while } \exp \text{ do } \cmd \}
\]

\[
F = \lambda f. \text{cond}^P_{\exp} (f \circ^P D_0 [\cmd], \{\})
\]

It follows from the previous lemma that \(f\) is a fixed point of \(F\) and hence \(LFP(F) \subseteq f\). That \(f \subseteq LFP(F)\) now means that \(LFP(F)(s)\) may specify divergence (i.e. contain \(\perp\) of \(S_\perp\)) in situations where \(f(s)\) does not but otherwise they are equal. This corresponds to the difference between \(P\) and \(D_0\).

That \(A FP(F)\) is well-defined follows from:

**Lemma 5.7:5.** When \(F\) is monotonic and \(\xi\)-monotonic

\[
\{ f : S_\perp \to \mathcal{E}(S_\perp) \mid F(f) = f \land f \subseteq LFP(F) \}
\]

is a complete lattice when ordered as in \(S_\perp \to \mathcal{E}(S_\perp)\). The least element is \(LFP(F)\) and the greatest element is \(A FP(F)\).

**Proof** Let \(D\) be \(\{ f \mid F(f) = f \land f \subseteq LFP(F) \}\) ordered by \(\xi\) as in \(S_\perp \to \mathcal{E}(S_\perp)\) and let \(E\) be \(\{ f \mid LFP(F) \subseteq f \subseteq LFP(F) \}\) ordered similarly. Clearly \(D\) is the set of those elements of \(E\) that are fixed points of \(F\). The assumptions about \(F\) guarantee that \(F\) specializes to a function from \(E\) to \(E\). Throughout this proof \(\bigcup\) refers to least upper bounds relative to \(S_\perp \to \mathcal{E}(S_\perp)\) and \((\bigcup Y)(s)\) therefore is \(\bigcup \{ f(s) \mid f \in Y \}\). If \(Y\) is a nonempty subset of \(E\) then \(\bigcup Y\) exists and is an element of \(E\). Existence is because \(f \in Y\) implies that \(f(s)\) is \(LFP(F)(s)\) or \(LFP(F)(s) - \{\perp\}\) and these two elements are comparable. Concerning membership of \(E\) note that \(LFP(F) \subseteq Y \) is immediate and \(Y \subseteq LFP(F)\) is because both \(LFP(F)(s)\) and \(LFP(F)(s) - \{\perp\}\) are subsets of \(LFP(F)(s)\).

Clearly \(LFP(F)\) is the least element of \(D\). Let now \(Y\) be a subset of \(D\). If \(Y\) is empty it is immediate that \(LFP(F)\) is the least upper
bound of $Y$ in $D$. If $Y$ is not empty $\bigcup Y$ exists and is an element of $E$ but it is conceivable that it is not a fixed point of $F$. We therefore define by transfinite induction

$$F(\lambda) = F(\bigcup \{F^{(k)} \mid k < \lambda \} \cup Y))$$

It is a straightforward transfinite induction to show that each $F(\lambda)$ exists and is an element of $E$ (so that $\bigcup \ldots$ exists). It is straightforward to show that if $k \leq \lambda$ then $F^{(k)} \subseteq F(\lambda)$. As in section 5.1 this gives an ordinal $\lambda$ such that $F(\lambda)$ is a fixed point of $F$. Hence $F(\lambda)$ is an element of $D$ and it is clearly an upper bound of $Y$. If $f$ is any other element of $D$ that is an upper bound of $Y$ it is straightforward to prove that $F(\lambda) \subseteq f$. Hence $Y$ has a least upper bound in $D$.

It is possible to show that $A_{F}(F)$ depends monotonically on $F$ but this will not be needed. The following formula will be used in the proof of the theorem below.

**Lemma 5.7:6.** $A_{F}(F)$ equals some $F(\lambda)$ where

$$F(0)(s) = \begin{cases} \{1\} & \text{if } LFP(F)(s) = \{1\} \\ LFP(F)(s) - \{1\} & \text{otherwise} \end{cases}$$

$$F(\lambda) = F(\bigcap \{F^{(k)} \mid k < \lambda \}) \quad \text{for } \lambda > 0$$

**Proof** We shall use the terminology introduced in the proof of the previous lemma. Analogously to what was shown there it can be shown that every nonempty subset $Y$ of $E$ has greatest lower bound $\bigcap Y$ in $E$ and that it is calculated pointwise. Since $F(0)$ is an element of $E$ it is a straightforward transfinite induction to show that each $F(\lambda)$ is defined and is an element of $E$. Since $k \leq \lambda$ implies $F^{(k)} \subseteq F(\lambda)$ there is a $\lambda$ such that $F(\lambda)$ is a fixed point of $F$. Hence $F(\lambda)$ is an
element of $D$ and $F_\alpha \in \text{AFP}(F)$. That $\text{AFP}(F) \subseteq F_\alpha$ follows by transfinite induction because $\text{AFP}(F) \subseteq F_\alpha(0)$.  

The semantics $A$ is then defined by using AFP instead of LFP.

In detail:

$$A[cmd] : S_\bot \rightarrow \mathcal{E}(S_\bot)$$

is defined by

$$A[\ell] = \{\ell\}, A[act]$$


$$A[\text{if } \text{exp } \text{then } cmd_1 \text{ else } cmd_2] = \text{cond}^p \circ A[cmd_1] P A[cmd_2]$$

$$A[cmd_1 \text{ or } cmd_2] = A[cmd_1] \upharpoonright A[cmd_2]$$

$$A[\text{while } \text{exp } \text{do } cmd] = \text{AFP}(A[\text{if } \text{exp } \text{then } cmd \text{ else } cmd_1 \text{ or } cmd_2])$$

It is immediate to verify that $\text{cond}^p$ and $\upharpoonright$ are monotonic and $\subseteq$-monotonic. Therefore the functional of which AFP is taken satisfies the requirements. It is then easy to see that the equations define a strict function of functionality as stated.

It may not be very intuitive what $A[cmd]$ accomplishes. The following "operational characterisation" of $A[cmd]$ may therefore help. It is explained in more detail after the proof.

**Theorem 5.7:** For all commands $cmd$ and states $s \in S_\bot$:

1. $A[cmd](s) \{1\} = \{s' \in S \mid \exists o, o' \in \{0,1\}^* : s[cmd](\text{smash}(s, o)) = (s', o')\}$
2. $A[cmd](s) \Rightarrow \forall \exists \in \{0,1\}^{*} : \forall o \in \{0,1\}^* : s[cmd](\text{smash}(s, wo)) = (1,1)$

**Proof** The first result is an immediate consequence of the following result that is proved by structural induction on $cmd$:

$$P[cmd] \subseteq A[cmd] \subseteq P[cmd]$$
Case cmd = act is immediate.

Case cmd = cmd₁;cmd₂ follows because \( \sigma^P \) is monotonic and \( \llangle \cdot \rrangle \)-monotonic.

Case cmd = if exp then cmd₁ else cmd₂ follows because \( \text{cond}_{\text{exp}}^P \) is monotonic and \( \llangle \cdot \rrangle \)-monotonic.

Case cmd = cmd₁ or cmd₂ follows because or is monotonic and \( \llangle \cdot \rrangle \)-monotonic.

Case cmd = while exp do cmd₁. Abbreviate

\[
F(f) = \text{cond}_{\text{exp}}^P (f \sigma^P F[cmd₁] \llangle \cdot \rrangle ) \\
G(g) = \text{cond}_{\text{exp}}^P (g \sigma^P \text{cond}_{\text{cmd₁}}^P \llangle \cdot \rrangle )
\]

It follows from the above that \( f \llangle g \) implies \( F(f) \llangle G(g) \) and that \( g \llangle f \) implies \( G(g) \llangle F(f) \). Since both \( \llangle \cdot \rrangle \) and \( \llangle \ldots \rrangle \) are admissible it follows by the induction principle of section 5.1 that

\[
\text{LFP}(F) \llangle \text{LFP}(G) \quad \text{and} \quad \text{LFP}(G) \llangle \text{LFP}(F)
\]

Since \( \text{LFP}(G) \llangle \text{AFP}(G) \llangle \text{LFP}(G) \) the result follows.

That the second result holds for all \( s \in S \) is proved directly by structural induction.

Case cmd = act is immediate.

Case cmd = cmd₁;cmd₂. It is straightforward to prove that

\( \llangle \text{cmd} \rrangle (s) \llangle 1 \) holds iff \( \llangle \text{cmd} \rrangle (s) \llangle 1 \) or (using the proof of the first part) \( \llangle \text{cmd} \rrangle (s) \llangle s', \llangle 1 \) and \( \llangle \text{cmd} \rrangle (s') \llangle 1 \). Similarly

\[
\llangle \text{cmd} \rrangle (\text{smash}(s, wo)) = (1, 1) \text{ holds iff } \llangle \text{cmd} \rrangle (\text{smash}(s, wo)) = (1, 1)
\]

or \( \llangle \text{cmd} \rrangle (s, wo) = (s', (wo)^c m) \) and \( \llangle \text{cmd} \rrangle (s', (wo)^c m) = (1, 1) \).

It is then straightforward to use the inductive hypotheses to show "only if" and "if".

Case cmd = if exp then cmd₁ else cmd₂ is straightforward (perform a case analysis upon \( \llangle \text{exp} \llangle (s) \)).
Case cmd = cmd\_1 or cmd\_2 is straightforward.

Case cmd = while exp do cmd\_1. This proof is rather long and we shall consider "if" and "only if" separately.

**Proof of "if".** The result is immediate for s = \_ so assume s ≠ \_ below. Let Q(w, s) abbreviate

∀\o \in \{0, 1\} : S\_ while exp do cmd\_1 (s, wo) = (\_\_, \_\_)

and define

P = \{(w, s) | Q(w, s) ∧ (Q(s', s) ⇒ |w| ≤ |w'|)\}

It suffices to show that

(w, s) \in P implies \_ while exp do cmd\_1 (s) = \_

We shall prove this result by induction on n where |w| = n. Note that Q(w, s) implies that J(exp)\_ (s) is true and hence

\_ while ... J(s) = \_ while ... J(\_ while ... J(s'))

\_ while ... J(s, o) = \_ while ... J(s, o) J(\_ cmd\_1 (s, o))

When n = 0 we have ∀\o: \_ while ... J(s, o) = (\_\_, \_\_) and by (1) it follows that \_ while ... J(s) = \_ (as it cannot be empty). When n > 0 we partition

P(n) = \{(w, s) \in P | |w| = n\}

into a disjoint union of three sets P\_1(n), P\_2(n), P\_3(n) and prove the result for each. For P\_1(n) take those elements (w, s) of P(n) such that ∀\o: \_ cmd\_1 (s, wo) = (\_\_, \_\_) and hence \_ while ... J(s) = \_.

301
To define $P_2(n)$ and $P_3(n)$ we need some additional notation. For each $(w, s)$ in $P(n) - P_1(n)$ there is $s' \in S$ and $o, o'$ such that 

$$S_{cmd}((s, wo)) = (s', o') \neq (\bot, \bot).$$

By fact 5.6:1 there exists $u = u_{w, s, s'}$, such that

one of $u$ and $w$ is a prefix of the other and

$$\forall o: S_{cmd}(s, wo) = (s', o) \neq (\bot, \bot)$$

For $P_2(n)$ take those $(w, s)$ in $P(n)$ such that some $|u_{w, s, s'}|$ is greater than 0. For $P_3(n)$ take those $(w, s)$ in $P(n)$ such that all $|u_{w, s, s'}|$ are 0. Clearly $P(n)$ is the disjoint union of $P_1(n)$, $P_2(n)$ and $P_3(n)$.

Consider $(w, s) \in P_2(n)$. Choose $s'$ such that $|u_{w, s, s'}| > 0$ and write $u = u_{w, s, s'}$. By the definition of $u$ it follows from (1) that $S_{cmd}(s) \exists s'$. Suppose $|u| \leq |w|$ and let $w'$ be chosen such that $uw' = w$. Then

$$\forall o: (\bot, \bot) = S_{while ...}((s, wo)) = S_{while ...}((s', w' o))$$

shows $Q(w', s')$. Since $|w'| < |w|$ it follows from the numerical induction that $S_{while ...}(s) \exists$ and this shows the result. Suppose next that $|u| > |w|$. Then

$$\forall o: (\bot, \bot) = S_{while ...}((s', o))$$

and the result follows much as before.

To prove the result for elements of $P_3(n)$ we proceed as follows. Define $S(n) = \{ s | \exists w: (w, s) \in P_3(n) \}$ and note that $S_{exp}(s)$ is true when $s \in S(n)$. Define a preorder upon the elements of $S(n)$ by

$s \rightarrow s'$ iff $u_{w, s, s'}$ exists for some $w$ (of length $n$)

We know that if such $u_{w, s, s'}$ exists it must have length 0. The
preorder is well-founded. For otherwise there is an infinite chain
\( s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \) and this would imply that
\[ \forall o: s(w, o) = (s', o) \]
contrary to \( n > 0 \) and \( (w, s_0) \in P_3(n) \) for some \( w \). It therefore suffices to show that \( \forall \text{while} \ldots J(s) \equiv \bot \) by induction on this well-founded order. So consider \( s \in S(n) \) and suppose the result holds for all \( s' \) such that \( s \rightarrow s' \) (or rather \( s \rightarrow^+ s' \)).

If there is some \( s' \) such that \( s \rightarrow s' \) we have \( \forall \text{while} \ldots J(s') \equiv \bot \) by the hypothesis of the well-founded induction. Also \( \forall \text{cmd}, J(s) \equiv s' \) follows by (1) and this shows the result. Next suppose there is no \( s' \) such that \( s \rightarrow s' \). By construction of \( P_3(n) \) there is \( s' \) (but not in \( S(n) \)) and \( w \) (of length \( n \)) such that \( u_{w, s, s'} \) exists. Since \( |u_{w, s, s'}| = 0 \) we have
\[ \forall o: s(w, o) = (s', o) \]
\[ \forall o: s(\text{while} \ldots J(s, o) = \text{while} \ldots J(s', o) \]
From the first equation and (1) it follows that \( \forall \text{cmd}, J(s) \equiv s' \).
Since \( Q(w, s) \) the second equation shows \( Q(w, s') \) and hence there is \( w' \) such that \( (w', s) \in P \). Since \( s \notin S(n) \) and \( |w'| \leq n \) the tuple \( (w', s') \) must have been considered previously in the numerical induction or when considering \( P_1(n) \) or \( P_2(n) \). Therefore \( \forall \text{while} \ldots J(s') \equiv \bot \) and this shows the result.

Proof of "only if": The result is immediate for \( s = \bot \) so assume \( s \neq \bot \) below. Define \( D = \bigcup_k D_k, D_0 = \{s\} \) and
\[ D_{k+1} = \forall \text{cmd}, J^+(\{s \in D_k | \exists \exp(s) \text{ is true}\}) \]
If \( s' \in D \) there is a minimal \( k \) such that \( s' \in D_k \). Also there is a list \( s = s_0, \ldots, s_k = s' \) with \( s_i \in D_i \) and
\(\exists \exp_I(s_i)\) is true when \(i<k\)

\(\exists \cmd_I(s_i)\exists s_{i+1}\) when \(i<k\)

Since \(s_i \neq \perp\) when \(i<k\) it follows by (1) and fact 5.6:1 that there are \(w_i\) such that

\(\forall_0: \exists \cmd_I(s_i, w_i) = (i+1, 0)\)

for \(i+1 < k\). This holds for \(i = k-1\) as well if \(s' \neq \perp\).

First suppose that \(s' \neq \perp\). Let \(s_i' = \perp\) and let \(s_i, w_i, k\) be as above. Since \(s_i' \neq \perp\) we know \(k>0\). Then \(\exists \cmd_I(s_{k-1}) \neq \perp\) so by the hypothesis of the structural induction there is \(w_{k-1}\) such that

\(\forall_0: \exists \cmd_I(s_{k-1}, w_{k-1}) = (1, 1)\)

Hence

\(\forall_0: \exists \while \exp \ do \ cmd_I(s_0, w_0 \ldots w_{k-1} 0) = (1, 1)\)

and this shows the result.

Next suppose \(s' = \perp\) but that there is \(s' \neq \perp\) with \(\exists \while I(s') = \{1\}\). By (1) it is immediate that

\(\forall_0: \exists \while \ldots I(s', 0) = (1, 1)\)

We have \(s_i, w_i\) and \(k\) as before and this gives

\(\forall_0: \exists \while \ldots I(s, w_0 \ldots w_{k-1} 0) = (1, 1)\)

thereby showing the result.

Finally we show that one of the two previous cases must apply when \(\exists \while \ldots I(s) \neq \perp\). For suppose

\(\forall s' \neq \perp: \exists \while \ldots I(s') \neq \{1\}\)

Using the notation of lemma 5.7.6 we have \(\forall s' \neq \perp: F_0(s') \neq \perp\). We
show by transfinite induction that \( \forall s' \in D: F_\Lambda(s') \neq \bot \). By lemma
5.7:6 this then shows that the previous two cases are exhaustive.

So let \( \lambda > 0 \) and \( s' \in D \). If \( \exists \text{exp}(s') \) is false

\[
F_\Lambda(s') = \{ s' \} \neq \bot
\]

If \( \exists \text{exp}(s') \) is true then

\[
F_\Lambda(s') = \bigcap \{ F_{(k)} : k \leq \lambda \} \uparrow (A_{\text{cmd}}(s'))
\]

\[
= \bigcup \{ F_{(k)}(s'') : k < \lambda \} \uparrow (s'' \in D)
\]

which does not contain \( \bot \). Finally note that \( \exists \text{exp}(s') \) cannot be
\( \bot \) when \( s' \in D \) as then \( A_{\text{while}... \exists}(s') = \{ s' \} \).

The first result stated in the theorem says that \( A_{\text{cmd}}(s) \) and
\( P_{\text{cmd}}(s) \) are equal if one ignores divergence (i.e. \( \bot \) of \( S_L \)). The
second result gives the conditions under which \( A_{\text{cmd}}(s) \) specifies
divergence. This can be explained more intuitively as follows.

Imagine an operating system that executes cmd on \( s \). Before starting
execution a time bound is chosen by unbounded nondeterminism. Up to
that time bound or is viewed as nondeterminism and some nondeter-
ministic choice of path is made. After the time bound or is viewed
as the amb operator mentioned in section 5.6. (This can be implemented
by letting the operating system perform a breadth-first search for a
terminating computation path.) So the program will terminate iff
(after the time bound has lapsed) there is a sequence of "choices of
path" such that it terminates. The semantics of cmd run under such
a scheme is given by \( A_{\text{cmd}} \). In a sense \( P_{\text{cmd}} \) corresponds to the
situation where the "time bound" is "infinity".

305
The difference between $A$ and $P$ is illustrated by the following examples.

**Example** Consider the programs

$$\text{cmd}_1 = x := \text{true} \; ; \; \text{while} \; x \; \text{do} \; (x := \text{false} \; \text{or skip})$$

$$\text{cmd}_2 = \text{while} \; \text{true} \; \text{do} \; \text{skip} \; \text{or} \; \text{cmd}_1$$

The "operational semantics" of these programs are best illustrated by the following execution trees:

Since both programs may loop we have

$$P[\text{cmd}_1] = P[\text{cmd}_2] = \lambda s. \{ s[\text{false}/x] \}$$

Further $\text{cmd}_1$ may always be brought to completion so that

$$A[\text{cmd}_1] = \lambda s. \{ s[\text{false}/x] \}$$

This differs from $A[\text{cmd}_2] = P[\text{cmd}_2]$ because in $\text{cmd}_2$ a path may be chosen such that thereafter the program has to loop.

**Example** Consider the programs

$$\text{cmd}_3 = x := \text{true} \; ; \; y := 0 \; ; \; \text{while} \; x \; \text{do} \; (y := y + 1 \; \text{or} \; \text{if} \; y \leq 27 \; \text{then} \; x := \text{false} \; \text{else} \; \text{skip})$$

$$\text{cmd}_4 = x := \text{true} \; ; \; y := 0 \; ; \; \text{while} \; x \; \text{do} \; (y := y + 1 \; \text{or} \; x := \text{false})$$

The semantics of $\text{cmd}_3$ is the same under $P$ and $A$ and is

$$\lambda s. \{ \Gamma \} \cup \{ s[\text{false}/x, m/y] \mid 0 \leq m \leq 27 \}$$

The semantics of $\text{cmd}_4$ is different under $P$ and $A$: 

306
Elcmd 43 = As. [false/x, m/y] | 0| m

ATcmd 41 =

Note that A[cmd] specifies unbounded nondeterminism.

It follows from the theorem and lemma 5.7:1 that all of P[cmd], D[cmd] (for O acceptable) and A[cmd] are equal when divergence (i.e. 1 of S) is ignored. It is clear that if A[cmd](s) specifies divergence then so must D[cmd](s), and if D[cmd](s) specifies divergence then so must P[cmd](s). This shows that


holds for all acceptable sets O. It further follows that P can be used to show absence of divergence (wrt. D) and A to prove presence of divergence.
To assess what has been achieved in this thesis it is appropriate first to recall why data flow analysis (including abstract interpretation) is worth doing. This was explained in section 2.1 and may be summarised as: To associate information with points in the program so as to show the safeness of performing program transformations that in general are not meaning preserving.

Viewing the development in these broad terms there are a number of major issues that have not been addressed.

- **Program points** should be introduced into the semantics in such a way that an approximating interpretation specifies the information possible at each program point. The development in this thesis may be viewed as working with only one program point: the end of the program. For ideas in this direction see /Nie82/.

- **Program transformations** should be formulated and proven safe to apply when the information specified above satisfies certain criteria. For ideas about how this might be done consult /Nie81b/.

- One should consider ways of implementing the data flow analyses described by approximating interpretations. This will be considered further below.
These issues are in a sense orthogonal to the aims of this thesis. (It is to be hoped that they are sufficiently orthogonal that they can be added without too much trouble.) The aims of the thesis were primarily to overcome the limited applicability of previous formulations of abstract interpretation: either they were based on viewing programs as flowcharts, which is hardly a general model, or else a mere study of toy languages. Turning to an assessment based on these aims it is convenient to consider chapters 2 to 4 separately from chapter 5.

Chapters 2 to 4 considered how to extend abstract interpretation to deal with a larger class of languages. This was done by basing the development upon a metalanguage for denotational semantics. The main achievements of this development are:

- The notion of two levels of types allows for a precise description of the method, i.e. of where to introduce power-domains in the collecting semantics.
- The metalanguage gives a reasonable degree of generality.
- The tensor product allows for a generalisation of the "relational method" for cartesian product and smash product.
- A framework is given for how to change between various data flow analysis methods such as "relational" and "independent attribute" methods (section 4.3).
- The notion of "inducing" one data flow analysis from another has been generalised and "expected definitions" have been studied. This should make it more convenient to describe a particular data flow analysis as an approximating interpretation.
There are several possibilities for extension of the development.

The most "urgent" extensions are:

- To allow "storable procedures", i.e. to extend the type structure with \( gt::=ft \). This remedies the major limitation of the metalanguage and is discussed further below.
- To allow \( Y \) not always to be LFP. Some uses of this will be mentioned below.
- Analogously to allow \( rec \) not always to mean REC (i.e. not always to specify the initial solution).

Some further possibilities are:

- Introduce notation for expressions of type \( gt \). This is further considered below.
- All the analyses are "forward" in the sense of propagating information from left to right. Dually one may consider how to handle "backward" analyses (see e.g. /Cou81/).

Even more remote goals are to extend the metalanguage with polymorphism, abstract data types, concurrency etc.

It is only sensible to anticipate that further research may cast doubt upon some of the decisions made in the development so far. It may be that the bottom-level metalanguage is not sufficiently close to operational intuitions. In particular it may be that \( x \) should be discarded and \( u \) used instead. If also \( \lambda \) is removed it would be feasible to require (in section 4.4) that all irreducible elements are either \( \lambda \) or an atom. Some potentially desirable consequences of doing so were outlined in the final remark in section 4.4.
Chapter 5 was concerned with bringing aspects of termination within the world of abstract interpretation. Some achievements of this are:

- More data flow analyses can be handled (e.g. section 5.5).
- Nondeterminism has formally been shown to be a data flow analysis problem. This is mainly of conceptual interest.
- Focus has been placed on the different partial orders $\Xi$ and $\mathcal{S}$. This gives better insight into the essence of abstract interpretation and clarifies some intuitive remarks in /Nie82/. (A first attempt at formalising this appeared already in /Myc81/.)

Many extensions need to be made before the development of chapter 5 can be made more general than just studying toy languages. Some possibilities are mentioned below and they seem to require "hard work":

- The powerdomain should be made more generally applicable (there are too few benign fdcpo's).
- The development should be performed for a metalanguage (as in chapters 2 to 4).
- The idea behind partly collecting and partly inducing is that only some of the arguments are viewed as describing sets of values. It would be natural to allow for gradually viewing more and more arguments this way, e.g. to obtain data flow analysis for a nondeterministic language. We shall return to this below.

One question that may be asked is whether the use of powerdomains, in particular $\mathcal{S}(\ldots)$, is inherently satisfactory. This will be dis-
cussed in some detail below and it seems that the answer is no.

The remainder of this chapter considers in more detail some of the directions for further research listed above. They are grouped under the headings "implementation issues", "extending the power of the metalanguage" (for chapters 2 to 4) and "the strong approach" (for chapter 5).

**Implementation issues**

There are at least two views of how to implement a data flow analysis. One is that there should be a system that directly executes the approximating semantics. This seems to be the view taken in /Don81/. Another is that the approximating semantics is merely a mathematical formulation of the (approximate) effect of computing data flow information by traditional means, e.g. as MFP or MOP solutions to certain systems of equations. This is the view taken in /Nie82/. In the discussion below we shall ignore that the equations can hardly be formulated if program points have not been introduced.

So let $I$ be an approximating semantics, $e$ an expression of type $gt \rightarrow gt$, $env$ an (empty) environment and $l$ and element of $I[gt]$. The desired evaluation system accepts $I$, $e$, $env$ and $l$ and calculates $I[e](env)(l)$. There is a similar situation for the standard semantics and here the SIS system of /Mos79/ can be used to evaluate $S[e](env)(d)$. The SIS system cannot be used to compute $I[e](env)(l)$. Some "minor" reasons are that SIS does not contain powerdomains and does not contain a (binary) least upper bound operation. The latter means that the expected definition of $I$(cond) cannot be defined.
A deeper reason has to do with the way the fixed point operator $Y$ is treated. Essentially SIS is a rewrite system for the typed $\lambda$-calculus and $Y$ is handled by rewriting

$$(Y(\lambda x. e)) \ldots \rightarrow (e[Y(\lambda x. e)/x]) \ldots$$

For a typical example let

$$e = \text{cond}(e_0, x \# e_1, e_2)$$

(corresponding to the semantics of a while-loop) where neither $e_0$, $e_1$ nor $e_2$ contains $x$. The unfolding continues as long as $e_0$ evaluates to true and $Y$ disappears altogether once $e_0$ evaluates to false. Should $e_0$ always evaluate to true the system loops, but one can hardly expect otherwise as the semantics then is $\bot$.

This strategy is not valid for evaluating $\mathbb{I}[e](\text{env})(1)$ and therefore another strategy must be developed. For generally some parts of the argument will always be passed to $x \# e_1$ and other parts to $e_2$. Therefore $Y$ need not disappear and the system will loop. (This is indeed what happens in /Don81/ as was said in chapter 1.) On the other hand if $\mathbb{I}[$gt$]$ is finite it will be the case that $\mathbb{I}[e](\text{env})(1)$ is obtained after a finite number of unfoldings (when all remaining $Y$ are viewed as $\bot$). It is therefore not acceptable that the system loops and it would be interesting to develop a suitable "rewrite rule".

The other approach was to assume that data flow analyses are computed as MOP and MFP solutions to certain equations. We shall write MOP(1) and MFP(1) for the information calculated at the end of the program given it is 1 at the beginning. It is known that

$$\text{MOP}(1) \subseteq \text{MFP}(1)$$

and the author conjectures that it extends to

$$\text{MOP}(1) \subseteq \mathbb{I}[e](\text{env})(1) \subseteq \text{MFP}(1)$$
and that neither inequality is an equality in general. Some intuitive arguments are presented below.

The intuition for why $\mathcal{I}(\text{env})(1)$ is between MOP(1) and MFP(1) but equal to neither is that its definition may be viewed as a mixture of MOP and MFP ingredients. An MFP ingredient may be illustrated for $e = f_1 \circ \text{cond}(\ldots, f_2, f_3)$. Here

$$\mathcal{I}(\text{env})(1) = f_1(f_2(1_{tt}) \cup f_3(1_{ff})) = \text{MFP}(1)$$

$$\text{MOP}(1) = f_1(f_2(1_{tt}) \cup f_1(f_3(1_{ff}))$$

which may differ when $f_1$ is not additive. A MOP ingredient may be illustrated for $e = Y(\lambda x . \lambda y . e')$. Here

$$\mathcal{I}(\text{env}) = \text{LFP}(\lambda f . \lambda l' . \mathcal{I}(\text{env}[f/x, l'/y]))$$

which is close to MOP whereas MFP essentially is

$$\forall l . \text{LFP}(\lambda f . \lambda l' . \mathcal{I}(\text{env}[f/x, (l l')/y]))$$

(The latter claim is based on /Nie82 p. 281/.) When $\mathcal{I}(\lambda y . e')(\text{env})$ is not additive the MFP and MOP solutions may well differ.

In data flow analysis it is commonly agreed that the MOP solution is more accurate than the MFP solution. This means that

$$\text{abs}'(\mathcal{I}(\text{env}))/\text{con} \subseteq \text{MOP} \subseteq \text{MFP}$$

However, it is usually the MFP solution that is computed in practice. For the MFP solution is decidable if all $\mathcal{I}(gt)$ are of finite height (though possibly infinite) whereas the MOP solution is undecidable in some such cases /KaU177/. The author conjectures that $\mathcal{I}(\text{env})$ behaves as MOP. One may therefore consider how to make $\mathcal{I}(\text{env})$ equal to MFP. One possibility is to enforce additivity of functions as then the MOP and MFP solutions agree. Another possibility is to redefine the interpretation of $Y(\lambda x . \lambda y . e')$ so it becomes the formula
(#) given above. Redefinition of \( Y \) is considered further below.

**Extending the power of the metalanguage**

There are other motivations for varying the interpretation of \([Y \ldots]\) than merely to obtain the MFP solution. One might desire even more approximative solutions than the MFP solution, e.g. if not all \([g]t\) are of finite height or if it "takes too long" to compute the MFP solution. This is the motivation behind the "widening" and "narrowing" ideas of \(/CoCo77b/\) and it is be hoped that these ideas can be incorporated by suitably interpreting \( Y \). The interpretation of \( Y \) can be allowed to vary simply by letting an interpretation specify the operator to be used. The expected definition (in the terminology of section 4.4) then is LFP. The predicate \( \mathcal{L}_{con}(t \rightarrow t) \rightarrow t \) will be useful for relating various interpretations of \( Y \). One can extend "inducing" to \( Y \) of type \((t \rightarrow t) \rightarrow t\) by using the suggestion in section 4.2. When \( t = ft \) this gives

\[
\text{abs}^{*}Y(\lambda g. \text{con}^{*}G(\text{abs}^{*}g^{*}\text{con})^{*}\text{abs})^{*}\text{con}
\]

as the induced version of \( Y \) applied to \( G \).

Quite analogously one may consider varying the interpretation of \( \text{rec} \) so that it is not always REC (giving the initial solution).

Taking a recursive domain

\[
\text{rec}X. N \# X + O
\]

as an example one might hope to obtain a finite domain of descriptions of such lists. This may be the way to incorporate the ideas in \(/Jon81b/\) about finding "safe approximate descriptions of computations by algorithms which manipulate recursive data structures". 

315
A major limitation of the metalanguage is that "storable procedures" cannot be accommodated, i.e. that the type structure does not allow \( gt::=ft \). To overcome this it will probably be necessary to find a locally continuous semi-functor \( \Theta \) over \( \text{ACLs} \) such that

\[
\Theta(A) \otimes \Theta(B) = \Theta(A \rightarrow B)
\]
\[
\Theta(f) \otimes \Theta(g) = \Theta(h \cdot g \cdot h')
\]

(ignoring that \( \Theta \) will be contravariant in one argument). This amounts to defining a relational method for the bottom-level domain constructor \( \rightarrow \) and make it work not only for powerdomains. An independent attribute method might be function space \( \rightarrow \) as has been used throughout this thesis. One application of a relational method for \( \rightarrow \) might be to extend abstract interpretation to handle properties like commutativity and associativity of functions. Such properties are frequently used when transforming applicative programs. Another application might be to express that the result of some function is always less than its argument. This may be the way to overcome some of the limitations of showing termination that were discussed at the end of section 5.5.

This situation is similar to the one with the tensor product. So one might try to construct \( A \otimes B \) by completing a quasi order obtained from a logic. Terms might be given by

\[
t ::= \Theta(a,b) | \text{tut'} | \text{tn'}
\]

where \( \Theta(a,b) \) is to be thought of as \( \lambda a'. a' \mapsto b \),. Some axioms might be

\[
\Theta(\text{awa'}, b) = \Theta(a,b) \cap \Theta(a', b)
\]
\[
\Theta(a,b \text{ub}') = \Theta(a,b) \cup \Theta(a,b')
\]
But even for the tensor product it was difficult to argue that it was the "right" generalisation from powerdomains to complete lattices and it is unlikely to be any easier here.

The metalanguage (section 2.1) has notation for expressions of type \( \mathbf{gt} \rightarrow \mathbf{gt} \) but not for expressions of type \( \mathbf{gt} \). It might be convenient to introduce expressions of type \( \mathbf{gt} \) by incorporating

\[ \lambda x. e \quad \text{and} \quad e(e') \]

As long as \( \mathbf{gt} := \mathbf{ft} \) is not allowed there will necessarily be some restrictions upon the use of \( \lambda \)-abstraction. The meaning of these constructs is clear in the standard semantics:

\[
S[\lambda x. e](\text{env}) = \lambda v. S[e](\text{env}[v/x])
\]

\[
S[e(e')](\text{env}) = (S[e](\text{env}))(S[e'](\text{env}))
\]

In an approximating interpretation it may be better to use

\[
I[\lambda x. e](\text{env}) = \text{lin}(\lambda w. I[e](\text{env}[w/x]))
\]

\[
I[e(e')](\text{env}) = (I[e](\text{env}))(I[e'](\text{env}))
\]

The use of \( \text{lin} \) may be important when \( x \) occurs more than once in \( e \).

As an example let \( e \) be \( xx+1 \) and view \( I[\lambda x. e](\text{env}) \) as a function from \( V = \{1, -, 0, +, \tau\} \) to itself. When applied to \( \tau \) one expects to get \( + \) rather than \( \tau \).

A consequence of the above suggestion is to cast some doubt upon the expected definition of composition in the bottom-level metalanguage. (It was defined in section 4.4 as \( f \circ g = f \cdot g \).) Intuitively one would expect \( f \circ g \) and \( \lambda x. f(g(x)) \) to denote the same. This holds in the standard semantics but in an approximating semantics the expected definitions give
\[ I[f \circ g](env) = f^*g \]
\[ \exists x. f(g(x))(env) = \text{lin}(f^*g) \]
where \( f = I(f) \) and \( g = I(g) \). These may differ even when \( f = \text{lin}(f) \)
and \( g = \text{lin}(g) \) as was shown in section 4.1. So one might consider
to let \( \text{lin}(f^*g) \) be the expected definition of \( f \circ g \).

The strong approach

The idea behind the partly collecting semantics was to pass from
\( f:D\times E \rightarrow F \) in the standard semantics to \( g:D \times E \rightarrow \mathcal{E}(F) \). This may be
approximated by an approximating semantics \( h:D \times B \rightarrow C \). At this point
it is conceivable that one may want to extend the data flow analysis
considerations to include also the first argument. So from \( g \) one
would pass to \( g': f(D \times E) \rightarrow \mathcal{E}(F) \) and this may be approximated by using
approximations to \( \mathcal{E}(D \times E) \) and \( \mathcal{E}(F) \). From \( h \) it is less clear what to
pass to. Perhaps it should be \( h': f(D) \star B \rightarrow C \) for \( \star \) some kind of tensor
product. Again this may be approximated by using approximations to
\( \mathcal{E}(D) \). It is not clear how to accommodate such a development. A
potential application is to perform data flow analysis for a nondeter-
ministic language.

To agree with data flow analysis intuitions it should be possible
to view the powerdomain as a certain collection of subsets. For this
reason it may be well not to base the development on the powerdomains
of /Plo82/: for it was conjectured in section 5.2 that these were not
atomically generated. However, it may be that the convexity assumption
used in the definition of \( \mathcal{E}(D) \) excludes subsets of interest. The
following example seems to show this.
Example Consider an imperative language with procedures of one argument and let S and V be sets of states and values respectively. A procedure may be modelled as a function from $S \times V$ to $S$ and evaluation of an argument as a function from $S$ to $S \times V$. Consider the following program:

```scheme
let p(value y) be skip in
let q(name y) be skip in
if x is even then LOOP else skip;
call p(x)
```

The set of states possible before `call p(x)` will include $\bot$ and some $s \in S$. Evaluation of $x$ in these states give $\bot$ and some $v \in V$, respectively. So the set of state and value pairs possible before transfer to $p$ includes $(\bot, \bot)$ and $(s, v)$. By convexity it must include $(s, \bot)$ as well. This is unfortunate because $p$ and $q$ differ upon $(s, \bot)$ although they are equal upon $(\bot, \bot)$ and $(s, v)$. It is safe to replace the call to $p$ by a call to $q$ but the collecting semantics will produce a set from which this cannot be seen!

It is unclear how to get around this problem. If convexity is abandoned then $\mathcal{D}(D)$ is not even partially ordered by the Egli-Milner order. If a powerdomain like $\mathcal{P}(D)$ is to be used it seems that the collecting semantics is useless for some program transformations. Neither outlook is comforting.
REFERENCES


320


/Mac71/ S. MacLane: Categories for the Working Mathematician, Springer-Verlag, 1971.

/Mar76/ G. Markowsky: Chain-complete posets and directed sets with applications, Algebra Universalis 6 (1976), pp. 53-68.


F. Nielson: Program transformations in a denotational setting, DAIMI report PB-140, Denmark 1981.


G. D. Plotkin: Lecture Notes (chapters 1 to 8); chapters 1 to 5 presented in Pisa, Italy, 1978.

G. D. Plotkin: personal communication.


INDEX

A 299
abstraction 13
abs 13
ABS 187
acceptable 288
ACC, ACCs 49
ACL 97
ACLs1, ACLs2 187
ACLI 210
ACLII 216
additive 15
adjoined 17, 165, 258
admissible 48, 63, 229, 240
APP 296
algebra 64
algebraic 44
ALG 96
angelic 299
approximating 174, 255
augmented 240
AUG 244
B 44
benign 235
BFDCPO 244
bottom-level 33

C 128, 136, 250, 276
card 71
cartesian product 45
category 49.
chain 15, 57, 229
CHAIN 60
close 115
closed 35

coalesced sum 46
cofinal 230
collecting 11, 127, 250
completely additive 15
completely augmented 240
complete lattice 15
completely linear 242
composition 49, 51
concretization 13
con 13
CON 187, 234
cone 57
consistently complete 45
continuous 47, 229
contravariant 51
contravariantly pure 41
convex 233
covariant functor 50
CP 41
cpo 44
cpo, CPOs 49
cpo2s 52
cpo-category 53
cross 103, 108

D 288
dcpo 227
def 46
directed chain 229
directed complete 227
directed continuous 229
directed set 44
domain 45
dom 76
down 46

325
embedding 55
Egli-Milner order 233
essential 170
exactly adjoined 165
expected-induce 200
expression part 76
extension 103

initial fixed point 56
interpretation 70, 247
irreducible element 98
irreducibly covered 171
irreducibly generated 171
is 46
isomorphism 54

K 51

LC 94, 234
least fixed point 15
lengthen 188
LFP 16, 47, 228
lifting 46
limiting 58
lin 137
LIN 245
locally continuous 53
locally monotonic 53
lower adjoint 165

IB 98
id 49
Id 51
ideal 94
ideal completion 97
identity 49
in 46
inc 275
inclusion 103
incomparable 234
independent attribute 18, 215
index 71
induce 18, 182, 259
init 115
initial algebra 64

MAX 234
mediate 58
MFP 19
model 107
monotonic 15
MOP 19
morphism 49

N, N_1 15, 44
nat 131
natural equivalence 114
natural transformation 114
naturality conditions 150
object 49
out 46

\( P \) 272
\( P \) 51
\( \ldots \) 41
partly collecting 276
partly induced 279
\( \text{PB} \) 98
pointed quasi order 94
prime element 98
product category 50
pseudo-continuous 256
pseudo-strict 256
\( P \times \) 106
\( P^* \) 106

RC 234
relational method 18, 215
rep 115

\( S \) 71, 249, 273
\( S(...) \) 62
semi-functor 111
semi-lattice 102
seperately 102
sim 142
\( \text{SIM} \) 150
smash 46
smash product 46
smash-invariant 275
standard 71
step, step 115
strict 15, 138
strongly strict 141
subcategory 49
sub cpo-category 63

symmetric 52

\( T_\Lambda \) 44
tensor product 103
top-level 33
tt 11
tupling 51
type part 70
\( T_X \) 104
\( T_X \) 105

U 44
up 46
upper adjoint 55, 165

v 99
view 138

weak product 216

Z 11

\( \rho, \rho', \rho_R, \rho_S \) 11, 92, 93, 94
\( \varepsilon \) 14
\( \cup, \cap \) 15
\( \downarrow, \uparrow \) 15
\( \cdot, \cdot \) 15

\( \ldots \times \ldots \) 34, 36, 39
\( \ldots \left[ \ldots / \ldots \right] \) 40

\( \ldots \downarrow \) 44, 46
\( \times \) 46, 50
\( \Pi \) 46
\( \downarrow \) 46, 52
\( \times \) 46

\( \Sigma \) 46
\( \Rightarrow \) 47, 227