A Semantical Framework for Practical Program Flow Analysis

Paul Rouleau

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

March 1988

ISBN 0-315-49118-3

ABSTRACT

A Semantical Framework for Practical
Program Flow Analysis

Paul Rouleau

We argue that Cousot's lattices for data flow analysis should be identified with Scott domains. Then we use a domain isomorphic to its own function space to write in a neat denotational style data flow analysis programs that handle unrestricted high order functions. A sample flow analysis of LISP is provided. We also develop Cousot's abstract semantics into categories that formally define consistency of distinct semantics and make the consistency relation transitive. Unlike usual styles of denotational semantics, data flow analysis requires us to interpret the top and bottom elements of the lattice as significant information. A naive implementation of the resulting semantics will not terminate its execution for a large number of useful programs because the bottom element occurs as an infinite loop. We avoid the difficulty with the help of an homomorphic image of the relevant Scott domain. Under some conditions imposed on the image domain, we can use the least fixed point theorem to find an image of the nonterminating semantics that terminates for every syntactically correct program, including explicitly written infinite loops. We show the conditions imposed on the image domain are reasonable for data flow analysis purposes, providing sample data flow analysis of both dynamically and lexically scoped lambda-calculus.

TABLE OF CONTENTS

Chapter Page

TABLE OF MATHEMATICAL SYMBOLS

Boldface numbers denote the definition or the semantics where the symbol is defined in the text. The abreviation "p." precedes the page number where an unnumbered definition is found . Many standard symbols are present in this table without formal definitions in the text.

Typographical Conventions

Boldface usually denote a domain used to define a semantics. `Typewriter` face are used to denote a syntactic variable, terminal or sentential form in a semantic equation. Whenever we need to enclose these syntactic elements within parentheses, we use $[\![$`double square brackets`$]\!]$ as a form of quoting. These brackets are not different from normal parentheses otherwise. The symbol ▮ is used to mark the end of proofs and of long semantics.

Symbol	Description
$\wedge$	logical "and"
$\vee$	logical "or"
$\Longrightarrow$	logical implication

$\Longleftrightarrow$	logical equivalence
$\forall$	universal quantifier
$\exists$	existential quantifier
$x = y$	equality of x and y
$x \neq y$	negation of equality of x and y
$x \leq y$	arithmetic x is less than or equal to y
$x < y$	arithmetic x is less than y
$x \geq y$	arithmetic x is greater than or equal to y
$x > y$	arithmetic x is greater than y
$x \in y$	x is an element of y
$x \notin y$	x is not an element of y
$x \subseteq y$	the set x is included in or equal to y
$x \not\subseteq y$	the set x is not included in or equal to y
$x \subset y$	the set x is strictly included in y
$x \not\subset y$	the set x is not strictly included in y
$\emptyset$	the empty set
$x \cup y$	the union of x and y
$\bigcup S$	the union of all sets belonging to S
$\{x_1 \ldots x_n\}$	the set whose elements are $x_1 \ldots x_n$
$\{x \mid \ldots\}$	the set of all x such that ...
$\langle x_1 \ldots x_n \rangle$	the tuple containing $x_1 \ldots x_n$

$x \sqsubseteq y$	lattice theoretic x is less than or equal to y(2.1, p.13)
$x \not\sqsubseteq y$	lattice theoretic x is not less than or equal to y (2.1, p. 13)
$x \sqcup y$	the lattice theoretic join of x and y (2.1, 2.4)
$\bigsqcup S$	the lattice theoretic join of all elements of S (2.4)
$x \sqcap y$	the lattice theoretic meet of x and y (2.4)
$\top$	the top element of a lattice (p.20)
$\bot$	the bottom element of a lattice (p.20)
$\sqsubseteq_D, \sqcup_D, \sqcap_D, \bot_D, \top_D$	all lattice theoretic operators may be indexed by lattice name (p.23)
Δ	the trivial proposition of an information system (2.2)
$\vdash$	the entailment relation on an information system (2.2)
$A = \langle D_A, \Delta_A, Con_A, \vdash_A \rangle$	components of information systems may be indexed by information system names
$\langle D_{3.1}, \Delta_{3.1}, Con_{3.1}, \vdash_{3.1} \rangle$	components of information systems may be indexed by an abreviation of the induced domain name (p.96–97)
$Cl(x)$	the closure of x under entailment (2.5)
$\lvert A \rvert$	the domain induced by the information system A (2.3)
$A \bowtie \mathbf{D}$	A is the information system underlying the domain $\mathbf{D}$ (p.95)
$\downarrow, \Downarrow$	auxiliary functions for the abstraction from semantics 3.1 to

	semantics 3.8 (3.9)
$\Uparrow$	**inverse of $\Downarrow$ (3.9)**
$\mathcal{ONE}$	**the one element information system (2.19)**
$\mathcal{BIT}$	**the two element information system (2.20)**
BIT	**the domain $\lvert\mathcal{BIT}\rvert$ (p.83)**
0	**the bottom element of BIT (p.33)**
1	**the top element of BIT (p.33)**
$FLAT(S)$	**the flat domain constructed from set S (2.21)**
$\{nil\}$	**abreviation for $FLAT(\{nil\})$**
$VP(\mathbf{D})$	**the set of very proper elements of domain $\mathbf{D}$ (3.3)**
$\{'\ldots\}$	**set emulated with domain $\mathbf{Ex} \longrightarrow \mathbf{BIT}$ (3.6)**
$x \uplus y$	**abreviation for $\langle f\varepsilon t(x), snd(x) \sqcup \{'y\}\rangle$ (3.8)**
$x \biguplus y$	**same as $x \uplus y$ but on different types of arguments (3.12)**
$BH(A)$	**the black-hole functor applied on A (2.35)**
$A \times B$	**the cartesian product of A and B (2.13)**
$\langle x, y\rangle$	**the pair of x and y (2.15, 2.16)**
$fst(x)$	**the first element of the pair x (2.16)**
$snd(x)$	**the second element of the pair x (2.16)**
$A + B$	**the disjoint union of A and B (2.29)**
$lft(x)$	**seleting x as the first component of a disjoint union (2.30)**
$rht(x)$	**seleting x as the second component of a disjoint union (2.30)**

$inl(x)$	injecting x as the first component of a disjoint union (**2.30**)
$inr(x)$	injecting x as the second component of the disjoint union (**2.30**)
$\longrightarrow$	production delimiter for context free grammars
$A \longrightarrow B$	the space of approximable mappings from A to B (**2.7**)
$f(x)$	application of the function induced by the approximable mapping f (**2.9**)
$\mathbf{A} \longrightarrow \mathbf{B}$	the function space from domain **A** to domain **B** (**2.32**)
$f \circ g$	the composition of f and g (**2.11**)
$\langle a, c\rangle \circ \langle b, d\rangle$	the composition of abstractions (**2.43**)
$f^i(x)$	iterating function f on x a number i of times (**2.35**)
$\lambda x.f$	the abstraction of variable x from expression f
$K(\cdot)$	the constant map (**2.12**)
$I(\cdot)$	the identity map (**2.10**)
I	the interpretation function of Cousot frameworks (**2.1**, **2.41**)
i_S	the identity abstraction from semantics S to itself (**2.44**)
fix	Tarski's least fixed point operator (**2.35**)
$\mathbf{Y}$	Curry's least fixed point operator (p.119)
fix', fix''	computational versions of fix (**4.5**, **4.11**)
if x **then** y **else** z	the conditional combinator (p.83)
$e[\mathbf{v} := x]$	assigning the value x to variable $\mathbf{v}$ in environment v (p.59)

$I_{3.1}, \mathbf{E}_{3.1}$	functions and domains indexed by semantics number (p.78)
$Uncurry(\cdot)$	the uncurrying combinator (p.105)
$LISP(\cdot)$	the LISP syntax functor (p.105)
$Lamb(\cdot)$	the lambda–calculus syntax functor (p.122)

CHAPTER 1

INTRODUCTION

We will investigate interprocedural data flow analysis. Data flow analysis means the process of analysing the propagation of information within a computer program in order to get information about its behaviour. One traditional application of such analysis is in compiler design. For example, an arithmetic expression such as x/x can be replaced by the compiler with the constant value 1 provided the variable x never takes the value zero. Data flow analysis must be used to verify that no expression that may have the value zero can propagate to become the value of variable x. Then we decide to optimize or not to optimize the program according to the result of the verification.

One especially delicate kind of data flow analysis we will consider in this work is interprocedural data flow analysis. This kind of analysis occurs when we have to handle the propagation of information across the various procedures and functions through the mechanisms of parameter passing, global variables and values returned by functions. The problem becomes extremely difficult in the presence of recursion and functions or procedures that propagate other functions and procedures through the above mechanisms before calling them. Such difficulties occur with programming languages like LISP.

Data flow analysis is further complicated by the requirements of termination and efficiency. This may sound trivial because any practical algorithm must terminate within a reasonable time period. However, one widely used approach, adopted in this work, is to view data flow analysis as giving a program an abstract semantics different from the usual semantics. Put this way, efficiency and termination mean that any program will execute quickly when interpreted according to the semantics.

Unfortunately, interprocedural analysis tends to become a nonterminating process because abstract semantics representing data flow reflects and sometimes amplifies the operational properties of the analysed program, including nontermination. This means the analysis of a nonterminating program is not likely to terminate itself unless the analysing algorithm is carefully designed. This idea is central to the work in chapters 4 and 5.

The standard solution to this problem is approximation. Getting back to the x/x example, we do not always need the exact set of possible values for x. Any superset of this set of possible values would do, because whenever zero does not belong to such a superset, we know x cannot be zero and optimisation is possible. However we run the risk of zero creeping into the superset, but not belonging to the exact set. This would prevent a valid optimisation, but would not endanger the correctness of the analysis because only valid optimisations are still performed. Provided the superset is easier to compute than the exact set, we sacrifice the ability to perform all valid optimisations to gain the ability to perform some of them within a reasonable time period. The art of data flow analysis is to find a good compromise between speed and completeness.

The present work started from a practical problem: the stack allocation problem. In functional languages such as LISP, the storage for variables should be allocated in memory. Two allocation methods can be considered: allocating the storage in a stack or in a heap. If stack allocation is used, memory management is performed quickly, but this method can be used only if memory is to be deallocated in the reverse of the allocation order. If the memory is deallocated in any other order, a stack can't handle it. If the memory is allocated in a heap, it could be deallocated in any order, but memory management will impose a high overhead on program execution. The problem was to find a way to use a mixed method, allocating memory in the stack when the last in first out requirement could be met and in a heap otherwise. This provides the speed of stack allocation, when possible, together with the generality of heap allocation, when required. Data flow analysis is required to determine the order of memory deallocation in order to figure out which variables to allocate in a stack and which variables to allocate in a heap.

Several analysis algorithms were considered. We wanted to prove that they correctly compute the order of deallocation and terminate. We quickly discovered we needed a formal framework to support such proofs. Finding such a framework proved to be a very difficult task. Every candidate seemed to bring us closer to good formalism, but also revealed new difficulties we had not previously considered relevant. To solve the foundational problems, we had to strip data flow analysis from the practical considerations about the stack allocation problem to look at the foundations of data flow analysis per se. We slowly turned our research away from practical considerations and started working on general considerations about how to write correct data flow analysis.

This work contains no reference to stack allocation except in this report on the origin of our interest on the topic. Our intention now is to demonstrate how interprocedural data flow analysis should be formalized and how we can insure the algorithms teminate.

Chapter 2 will present the traditional framework for data flow analysis problems first described in Cousot (Cousot and Cousot 1978). Formal definitions will be given. We also introduce the idea that Cousot's lattices should be equated with Scott's domains. We intend to show that these lattices are relevant when comes the time to analyze high order functional programs, so much of this chapter is devoted to a presentation of the lattice structure of Scott's domains and its relationship with Cousot's framework for data flow analysis.

Chapter 3 is devoted to an example of data flow analysis done along the lines of Chapter 2. We show the analysis of high order functions can indeed be done as we claimed and provide examples of the basic definitions.

Chapter 4 is devoted to the termination problem. It will be shown that the standard techniques for approximation used in data flow analysis greatly increase the severity of nontermination problems in the presence of high-order functions, and even introduce nonterminating loops in the approximation where terminating loops were used in the original program. Fortunately, explicit use of some lattice theoretic properties of Scott's domains can be used to force termination at places where normal order reduction would not. The key idea is that a non terminating function f over a domain D may have a terminating image f' over an homomorphic domain D'. It is perfectly acceptable to substitute f' for f in an actual compiler because the homomorphism allows all relevant information about the value of f to be retrieved. It is shown this is sufficient to enforce termination of the analysis of any dynamically scoped LISP program.

Chapter 5 is devoted to the analysis of lexically scoped lambda calculus and lexically scoped LISP programs in general. The techniques in chapter 4 are not sufficient to handle its semantics because of non termination problems, unless some clever approximations are introduced to fix this. We describe the required approximation and explain how and why they work. Similar approximations can be written for other high level languages.

Chapter 6 is the conclusion, wrapping up the key results and the future work to be done.

CHAPTER 2

DATA FLOW FRAMEWORKS AND SCOTT DOMAINS

The usual formalization of data flow analysis was first introduced by Cousot and Cousot (1978). The original definition is as follow:

Definition 2.1 Data Flow Frameworks

A data flow framework is a tuple $\langle P, D, F, I\rangle$ where P is a set of "program points", D is a lattice with partial order $\sqsubseteq$ and join operation $\sqcup$, F is the space of all functions $D \longrightarrow D$ that are distributive over $\sqcup$ and I is an interpretation function $P \longrightarrow D$ defined by a set of equations $I(x_i) = t(I(x_1)\ldots I(x_n))$, where $x_i \in P$, $1 \leq i \leq n$, and where t represents a term involving only functions in F.

Cousot's idea is to express the semantics of a program as the solution of a set of equations. The set P of program points are usually taken as the set of vertices of the control flow graph of the program. The lattice D is usually a lattice of environments $N \longrightarrow V$, each mapping variables occurring in the program to some element in a domain V of data flow values relevant to the particular problem under consideration. It follows from these assumptions the function $I\colon P \longrightarrow D$ must be defined according to the flow of data along the edges of the control flow graph induced by the language semantics. Cousot has shown this function turns out to be a set of equations of the required form. The function space F is chosen to restrict the set of acceptable equations to insure the solution always exists.

We shall not give an example of the use of this framework until chapter 3. Impatient readers may go to Cousot's original paper where the issues involved are discussed in greater detail. We will only remark that Scott's style of denotational semantics for programming languages often fits into Cousot's frameworks. The idea is not new since Cousot and Cousot themselves used such a semantics as an example for their work.

2.1 Scott Domains

In this chapter, we will be concerned about the structure of the allowable lattices. In particular we will claim Scott's domains should be identified with Cousot's lattices and his distributive function space F should be replaced by the category of approximable functions on Scott's domains. This idea is intuitively easy to justify. If data flow analysis is to be implemented on a computer, any data type involved can be considered to be a domain for the denotational semantics of the language used to write the data flow analysis program. Therefore there is no loss of generality in identifying Scott's domains with Cousot's lattices. Cousot's space $F: D \longrightarrow D$ of distributive functions is introduced to force fixed points to exist. Such fixed points ensure the existence of the function I that solve the equations induced by the data flow problem. We suggest using the category of approximable maps instead of F

because it also has that property and in addition, it supports arbitrarily high order functions, whereas Cousot's original definition does not. Such high order functions are very useful in interprocedural data flow analysis. This was already noticed by Sharir (1981). He extended Cousot's framework to use not only the distributive function space F but to also take advantage of the fact F is a lattice to introduce a space $H: F \longrightarrow F$ of second order distributive functions. This allowed him to handle interprocedural data flow analysis with the exception of functional arguments. We will show that in the presence of arbitrarily high order functions we can handle any kind of interprocedural data flow analysis, including functional arguments that resist Sharir's methods.

Let's look more closely at Scott's domains. They are defined in different ways in different papers (Scott 1976, 1981, 1982; Stoy 1977). It seems to be a common attitude among computer scientists to ignore the details of the construction of a domain. The reason is given by Stoy (1977): "So our purpose is to demonstrate that a domain can be constructed which satisfies the formal requirements, to give us confidence that our theory is not vacuous. ... once we have demonstrated that our requirements are satisfiable we can safely return to thinking of our value domains in any more intuitively natural way we please." This is why Scott's work is both famous and little known. Computer scientists want to be confident their work has proper formal ground and providing such formal ground is exactly what Scott's theory of domains does. However the methods used are difficult to understand and remote from everyday programming experience so most people prefer to use the results without looking at the details of the constructions.

At least one author, Dana Scott himself doesn't share this view. He really believes that not only his theorems but also the underlying theory of domains is significant to computer science. This is one of the reasons that motivated him to write so many different versions of his theory of domains (Scott 1976, 1981, 1982). He was looking for a presentation that looks intuitively natural to computer scientists in the hope the foundations of his work would be more widely understood and used. Scott's personal view on the problem is found in (Scott 1982).

It should be clear that for data flow analysis purpose, the details of the construction of a domain are relevant. This can be seen immediately from the observation that Cousot's framework requires us to be aware of the lattice structure of our domains. However if we select the version of Scott's theory of domains founded on information systems (Scott 1982), we will find that Scott's partial order over his domains are very significant in data flow analysis terms. We will devote an important part of the present chapter to that question. We will summarize some important steps of Scott's theory of domains. Most results will be cited without proof unless they are original. The reader may look up the references for more information. The key point here is not to develop the theory but to look at the details of its application to data flow analysis. The first step is to clearly establish what are the lattices involved, what they mean in terms of data flow analysis and what are the partial orders, join and meet operations, bottom and top elements in any of these lattices. One of the keys to Scott's methods for domain construction is the use of the category theory to be able to talk about general properties of domains without having knowledge of their structures. As far as data flow analysis is concerned, this practice is useful but insufficient. When we face a specific problem, we cannot solve it unless we know the exact structure of the relevant lattice. Therefore we will not only discuss the constructions involved in the abstract but we will also tell how to perform the reverse operation of concretizing the categorical ideas into more explicit lattice theory.

Definition 2.2 Information Systems

An information system is a tuple $\langle D, \Delta, Con, \vdash \rangle$ where D is a set, the set of data objects or propositions; Δ is a distinguished member of D, the least informative member of D; Con is a set of finite subsets of D, the set of consistent sets; and $\vdash$ is a binary relation between members of Con and members of D, the entailment relation for objects. An information system must also satisfy the following axioms:

$\forall u, v \subset D$ and $\forall X \in D$

i) $u \in Con$ whenever $u \subset v$ and $v \in Con$

ii) $\{X\} \in Con$

iii) $u \sqcup \{X\} \in Con$ *if* $u \in Con$ and $u \vdash X$.

$\forall u, v \in Con$ and $\forall X \in D$

iv) $u \vdash \Delta$

v) $u \vdash X$ *if* $X \in u$

vi) *if* $\forall Y \in u, \quad v \vdash Y$ *and* $u \vdash X$ *then* $v \vdash X$

Scott recommends interpreting the elements of D as propositions that can be true about the elements of its domains. As a sort of corollary, an element of the domain is characterized by the set of propositions that are true about it. If we assume no two elements satisfy the same set of true propositions, (a very reasonable assumption for otherwise we probably do not use an information system that suits our data flow problem) we can go further and identify every element x with the subset of D containing all true propositions about that element x. In this context the set Con should be interpreted as the set of finite consistent sets of propositions. More precisely, if the set u of propositions is in Con, then there is an element x in our domain that satisfies at least all the propositions in u. The proposition $\Delta \in D$ is the trivial proposition that is satisfied by every element of the domain. The entailment relation represents the logical dependencies between propositions. When a set of propositions u entails a proposition X, this is interpreted as saying that X can be deduced in some way from u. As a result, for every element e that satisfies all propositions in u, that element e will also satisfy the entailed proposition X. Given this interpretation of information systems, the axioms should sound intuitively true to the reader. Note also that the identification of domain elements with the set of propositions that are true about them leads to the following definition.

Definition 2.3 Domains and Elements of the Domains

Let $A = \langle D, \Delta, Con, \vdash\rangle$ be an information system. The domain generated by A is denoted $|A|$ and is defined as follow:

$$|A| = \{x \mid \textit{every finite subset of } x \textit{ is in } Con \textit{ and If } u \subseteq x \wedge u \vdash X \textit{ then } X \in x\}$$

With this definition of domains, domain elementhood is identical to set theoretical elementhood.

This definition means only consistent information is true about an element of a domain and the set of true propositions about an element is closed under deduction. We assume the information system is so designed that we can identify elements with closed sets of propositions. When the time comes to design the set D of proposition for an information system, it is intuitively mandatory that every element must be characterized by a unique closed set of propositions. Otherwise there would be elements without any way to distinguish them. We require the converse to be true: every closed set of proposition must denote an element. This allows some closed sets to apply to more than one "real" element of the domain. The world "real" here means the elements we originally intended to include in the domain when we designed the set D of allowable propositions. Definition 2.3 implicitly introduces a new class of elements, the class of partial elements. They correspond to those closed sets that do not identify with a unique "real" element.

They are called partial because they are incompletely defined with respect to our original intention. We don't have enough information to tell which "real" element we are talking about.

This identification of elements with sets of true propositions is interesting from the point of view of data flow analysis. In this context the interesting property of abstract semantics is not the particular approximation we are computing but the information conveyed by that approximation. For instance in the case of the (x/x) optimization problem what we want to know is if some particular value can or cannot be taken by a variable. The sets used in the abstract semantics are of no interest in themselves. They are interesting because they represent the set of propositions that such variables can take such and such values in some execution of the program, and these propositions are the correct formal ground to make the decision of optimizing the program or not. This is one more justification to the proposed identification of Cousot's lattices and Scott's domains.

2.1.1 The Lattice Structure of Domains

Every domain $|A|$ induced by an information system $A = \langle D, \Delta, Con, \vdash\rangle$ is partially ordered under set inclusion because it is a collection of sets; namely the collection of all consistent subsets of D that are closed under entailment. This is the lattice structure of the domains. We can immediately infer the correct interpretation of the partial ordering: $x \sqsubseteq y$ if and only if all propositions that are true about x are also true about y, or equivalently $x \sqsubseteq y$ if all information contained in x is also contained in y. This interpretation agrees perfectly with the intended purpose of data flow analysis. For example, we may be looking for a quick way to compute some element y that contains all the information in x, ignoring the fact that y may contain information that does not apply to x. This amounts to computing a value y such that $x \sqsubseteq y$ as a valid approximation to x. Some other problems may have the opposite requirement. We may want to compute a value y that contains only information that applies to x disregarding the fact that some information applying to x does not apply to y. In this case we are computing a value y such that $y \sqsubseteq x$ as a valid approximation to x.

The first notion of approximation is used when we want to be safe on the true side, that is if a proposition is true, our approximation must give the correct answer. On the other hand, we do not mind concluding that a proposition is true when it is not. This is expressed by the relation $x \sqsubseteq y$, all propositions true about x are also true about y. For instance, the (x/x) optimization problem requires this kind of approximation. We want to be able to detect every place where x can be equal to zero in order to know where the optimization is invalid. We do not mind if our approximation tells us some variables can take the value zero when it is not the case for a non optimized program is still a correct program.

The second kind of approximation is used when we are in the reverse situation. We want to be able to say only true things about an element but do not mind if not all information about that element is available. It is often useful in data flow analysis. This kind of approximation is also crucial in the theory of domains. We will look at the details after a few definitions.

Definition 2.4 Lattice Theoretic Meet and Join

Let $x, y \in |A|$, $x \sqcup y$ denotes the least element of $|A|$ such that $x \sqsubseteq x \sqcup y$ and $y \sqsubseteq x \sqcup y$ when such an element exists. This means $\sqsubseteq$ is the lattice theoretic join on $|A|$. Similarly $x \sqcap y$ denotes the greatest element of $|A|$ such that $x \sqcap y \sqsubseteq x$ and $x \sqcap y \sqsubseteq y$; that is, $\sqcap$ is the lattice theoretic meet on $|A|$. Note that unlike $x \sqcup y$, $x \sqcap y$ always exists in $|A|$, see (Scott 1982) *for the details. These operators will generalize as usual over sets of elements instead of pairs, for instance one can write $\bigsqcup\{\ldots\}$ to denote the join of all elements in a set.*

Definition 2.5 Finite and Infinite Elements, Closure of a Set

If $A = \langle D, \Delta, Con, \vdash \rangle$ is an information system, and $u \in Con$, we define the closure

$$Cl(u) = \{X \mid u \vdash X\}$$

We can prove (Scott 1982) that $\forall u \in Con$, $Cl(u) \in |A|$. All such elements are called the finite elements of $|A|$. Elements that are not of the form $Cl(u)$ for some finite u are called infinite elements of $|A|$.

Proposition 2.6 (Scott 1981, 1982)

Let $A = \langle D, \Delta, Con, \vdash \rangle$ an information system. The following hold for all x in $|A|$.

$$x = \bigsqcup\{Cl(u) \mid u \in Con \wedge u \sqsubseteq x\}$$

The reader should not confuse finite elements and finite sets of propositions. An element is finite in the sense there is a finite set of propositions that entails all other propositions that belong to that element. Of course there could be an infinite number of such entailed propositions and a finite element may well turn out to be an infinite set. When we talk about finite elements we do not mean the finite cardinality of the element. We mean the element can be generated from a finite consistent set of propositions with the help of the closure operator Cl.

Now let's look back at Scott's definition of approximation. An element y is a valid approximation of x if and only if $y \sqsubseteq x$. This means if a consistent set u is included in an element x, then $Cl(u) \sqsubseteq x$ and therefore $Cl(u)$ can be called a finite approximation of x. Proposition 2.6 says every element x, finite or infinite, is the join of all its finite approximation. This idea leads us to an interesting theory of computation. We can equate the finite elements with the elements that have a finite computer representation. This involves no loss of generality because we can organize the set of propositions in our information system to define our elements in terms of their computer representation. Finite data structures can be described with finitely many statements and conversely a finite number of statements can be stored in a finite amount of space. The question is how do we carry out computation on an infinite element if it has no finite computer representation? The answer is that we seldom need to know an infinite amount of information about an element. A large enough finite approximation will usually do. The normal procedure is to enumerate an increasing sequence u_i of finite approximations of x such that $\bigsqcup u_i = x$. Sooner or later we will find a large enough u_i for our purpose. Let's look at an example.

Assume we want an information system that defines the set of all functions $\mathbf{N} \longrightarrow \mathbf{N}$ where $\mathbf{N}$ is the set of natural numbers. The information system $A = \langle D, \Delta, Con, \vdash \rangle$ can be defined as:

$D = \{Int\} \cup \{\,Val(x,y) \mid x, y \in \mathbf{N}\}$

where *Int* is the proposition "the value of the function will be an integer" and $Val(x,y)$ is the proposition "when the function is applied to x it gives the value y".

$\Delta = Int$

$Con = \{u \subset D \mid u \text{ is finite and } Val(x,y),\ Val(x,z) \in u \Longrightarrow y = z\}$

$u \vdash X \iff X = Int \vee X \in u$

The information we are interested in knowing about a function is the trivial proposition *Int* that states it gives an integer value and the more useful statement $Val(x,y)$ that tells us what the function value is when given the argument x. A total element f of the domain $|A|$ is the set of propositions that tells us the value of $f(x)$ for each integer x (as well as the proposition *Int*). If not enough propositions are provided, the function is only partially defined. The set *Con* is defined to ensure that an element assigns a unique value to each defined argument so that the elements of the domain will indeed be functions. The entailment relation is the minimal reflexive relation that will infer Δ from any proposition since we cannot infer the value of the function at one point from its values at other points. The relation $f \sqsubseteq g$ on $|A|$ means that g is defined at more points than f, and whenever both are defined, they have the same value. The finite elements are exactly those sets of propositions including *Int* that define a function at finitely many points and leave it undefined everywhere else. In this example a finite element turns out to be a finite set. This happens because the entailment relation $\vdash$ is very minimal; as we have said, it is not true in the general case. The question we want to answer is what is the computational significance of the finite elements.

Let's consider the factorial function. In the domain $|A|$ this is the set recursively defined as:

$$Fact = \{Int\} \sqcup \{Val(0,1)\} \sqcup \{Val(x+1,(x+1)y) \mid Val(x,y) \in Fact\}$$

The reader can check that this is an infinite element. In Pascal-like languages, this function will be coded:

```
y := 1
```

```
For $i := 1$ to $x$ do $y := y \times i$;
```

We can view this as a program to find a proposition $Val(x, y) \in Fact$ that gives the value of factorial for the argument x. By definition $\{ Val(0, 1)\} \in Fact$. The program represents this knowledge by assigning 1 to y in the first line and skipping the body of the For loop if x happens to be 0. If we iterate the loop knowing that if $Val(i, y) \in Fact$, then we gain the knowledge that $Val(i + 1, (i + 1)y) \in Fact$ on the next iteration. At any point in time we know only finitely many values of the factorial function and the element represented by this knowledge is a finite element that partially defines $Fact$. The more we iterate the loop, the more defined is our approximation but we never reach the point where the sum of all computed information constitutes an infinite element. We simply exit the loop when the finite approximation computed so far is large enough to tell us what is the value of the factorial for some particular value of x.

This example illustrates how an infinite element is represented by an increasing infinite sequence of finite elements for computational purposes. Proposition 2.6 says that every infinite element can be represented in this way. This idea that infinite elements are always represented by such sequences (or more generally speaking directed sets [Scott 1976, 1981]) of finite elements for computational purposes lies at the foundation of Scott's theory of domains. This concludes our discussion on the significance of the partial ordering $\sqsubseteq$ over domains.

We did introduce in definition 2.4 the lattice theoretic meet and join over domains. The element $x \sqcap y$ is simply the set of all propositions that belongs to both x and y. That intersection is itself an element and therefore the meet over a domain is exactly the set theoretic intersection. The meet $x \sqcap y$ always exists because there is a least element in the domain, namely the set $\perp = Cl(\{\Delta\})$ of all trivialy entailed propositions. Intuitively $x \sqcap y$ is the largest partial object that approximates both x and y.

The join $x \sqcup y$ over a domain is not the same as the set theoretical union of propositions. For instance there may be a proposition $X \notin x \cup y$ and a set $u \subseteq x \cup y$ such that neither $u \subseteq x$ nor $u \subseteq y$ but such that $u \vdash X$. In that case the set theoretical union $x \cup y$ is not closed under entailment and cannot be an element. When we join the two elements x and y, new deductions are possible when we combine propositions about x with propositions about y. The closure $Cl(x \cup y)$ is closed under entailment and is the join over the domain of x and y provided that set is consistent. If $x \cup y$ has an inconsistent subset, the domain theoretic join does not exist for x and y. When this join exists, it is always the least element $x \sqcup y$ that can be approximated by both x and y.

To complete the analysis of the significance of the lattice structure we still need to discuss the meaning of the bottom and the top elements when they exist. We have already mentioned that $Cl(\{\Delta\})$ is $\perp$ and is the set of all trivially true propositions in the information system. The top element $\top$ does not always exist. When it exists it is the set $\bigsqcup\{x \mid x \in Con\}$ because every consistent set must generate an element that is less than or equal to $\top$. It turns out that $\top$ contains all propositions that are not trivialy false for every element of the information system. These elements have some additional significance. We shall not discuss them right now because we lack some useful results that illuminate their role in Cousot's framework. We will return to the topic in due time, when those results are available.

This completes for the moment our discussion of the significance of the lattice theoretic structure of the domains. Before leaving the topic the reader should be aware that approximations are used in two different ways. In data flow analysis we use approximations because we do not need to find the exact set of propositions that characterizes an element. In Scott's theory of partial objects, we use approximations because infinite elements cannot be exactly computed but any finite amount of information we may ever want about them can be generated by infinite increasing sequences of finite approximations. In Scott's theory of partial objects we say that "x approximates y" iff $x \sqsubseteq y$. In Cousot's framework for data flow analysis we say that "x approximates y" iff either $x \sqsubseteq y$ or $y \sqsubseteq x$ depending of which is relevant to the problem under consideration. We immediately see that in some cases, Cousot's approximation matches Scott's and in other cases it doesn't. These two uses of the same relation $\sqsubseteq$ are intertwined but distinct and must not be confused.

2.2 Approximable Mappings and Approximable Functions

We are now ready to start discussing the significance of the category of approximable mappings over domains in the context of data flow analysis.

Definition 2.7 Approximable Mappings

Let $A = \langle D, \Delta, Con, \vdash \rangle$ and $B = \langle D', \Delta', Con', \vdash' \rangle$ be two information systems. An approximable mapping $f: A \longrightarrow B$ is a binary relation between Con and Con' such that

i) $\emptyset \; f \; \emptyset$

ii) $u' \vdash u$, $u \; f \; v$, *and* $v \vdash' v'$ *always imply* $u' \; f \; v'$

iii) $u \; f \; v$, *and* $u \; f \; v'$ *always imply* $v \sqcup v' \in Con'$ *and* $u \; f \; v \sqcup v'$

In ii) $u' \vdash u$ *means* $\forall X \in u, \quad u' \vdash X$ *and similarly,* $v \vdash' v'$ *means* $\forall Y \in v'$, $v \vdash' Y$

The approximable mapping f determines a function $f: |A| \longrightarrow |B|$ *as follow:*

$$f(x) = \bigsqcup \{v \in Con' \mid \exists u, \; u \; f \; v\}$$

or equivalently

$$f(x) = \{Y \in D' \mid \exists u \sqsubseteq x, \; u \; f \; \{Y\}\}$$

An approximable mapping $f: A \longrightarrow B$ is defined as a binary relation between the consistent sets of A and the consistent sets of B. Intuitively f should describe a function from $|A|$ to $|B|$, this binary relation relates for every x the consistent sets of propositions that are true about x to the consistent sets of propositions that are true about $f(x)$. Obviously the empty set of propositions is always true about both x and $f(x)$. This expressed by condition i). Condition ii) states that we can use the fact that $u\ f\ v$ to deduce things that are true about $f(x)$ from things that are true about x. In other words, the binary relation f is a deduction relation between propositions in A that characterize x and propositions in B that characterize the value $f(x)$ at every point x where $f(x)$ is defined. The role of condition iii) is to guarantee we are indeed working with a well defined function in the sense that the set of propositions that characterizes the value of the function is always a consistent set. The next proposition formally justifies the notation $f(x)$ introduced in 2.7.

Proposition 2.8 Formal Properties of Approximable Mappings (Scott 1982)

Let $f, g\colon A \longrightarrow B$ be approximable mapping where A and B are as in 2.7, then

i) $\text{if } x \in |A| \text{ then } f(x) \in |B|$

ii) $f = g \iff \forall x,\quad f(x) = g(x)$

iii) $f \subseteq g \iff \forall x \in |A|,\quad f(x) \sqsubseteq g(x)$

iv) $x \sqsubseteq_{|A|} y \Longrightarrow f(x) \sqsubseteq_{|B|} f(y)$

v) $\forall u \in Con,\ v \in Con',\quad u\ (\ f)v \iff Cl(v) \sqsubseteq_{|B|} f(Cl(u))$

In 2.8 we introduce the notation $\sqsubseteq_{|A|}$ to mean the partial order defined over $|A|$. Remember this relation happens to be identical to set inclusion. Subscripts are introduced only for clarity.

There is one more identity that connects the notion of approximable mappings and Scott's theory of partial objects. Remember that definition 2.7 states that

$$f(x) = \bigsqcup\{v \in Con' \mid \exists u, \quad u\ f\ v\}$$

If x happens to be an element, then by 2.8.i $f(x)$ is also an element and therefore $f(x)$ is closed under entailment. We can immediately infer from this the following proposition:

Proposition 2.9

Let $f\colon A \longrightarrow B$ be an approximable mapping and x be an element in $|A|$, then

$$f(x) = \bigsqcup\{Cl(v) \mid \exists u \sqsubseteq x, \quad u\ f\ v\}$$

Remember that proposition 2.6 states $x = \bigsqcup\{Cl(u) \mid u \sqsubseteq x\}$, this was interpreted as the statement that x can be represented by the set of all its finite approximations. What 2.8 means is that given such a representation of x, we can compute a similar representation of $f(x)$ and, therefore, the function f allows us to explicitly compute the finite approximations of $f(x)$ from the finite approximations of x. For every u such that $Cl(u)$ is an approximation of x, the binary relation f

associates a set of values v such that $Cl(v)$ is an approximation of $f(x)$. Approximable mappings are the correct definition of what a computable function is in light of Scott's theory of partial objects. Scott (1976, 1981) elaborates on that claim, connecting his theory of domains to the more standard theory of partial recursive functions and recursive enumerability. We will skip these questions because they are not relevant to data flow analysis.

2.2.1 The Category of Approximable Mapping

The following properties are relevant to data flow analysis and bring in the categorical structure of Scott's function space.

Definition 2.10 The Identity Function (Scott 1982)

Let $A = \langle D, \Delta, Con, \vdash \rangle$ be an information system. Then the following formula defines an approximable mapping $I: A \longrightarrow A$.

i) $\forall u, v \in Con, \quad u \, (I) v \iff u \vdash v$

For all $x \in |A|$, this mapping has the property:

ii) $I(x) = x$

Definition 2.11 Composition of Functions (Scott 1982)

Let $A, B = \langle D, \Delta, Con, \vdash \rangle$ and C be information systems. Let $f: A \longrightarrow B$ and $g: B \longrightarrow C$ be two approximable mappings. Then we can define an approximation mapping $g \circ f: A \longrightarrow C$ as follow:

i) $u \, (g \circ f) \, w \iff \exists v \in Con, \quad (u \, f \, v) \wedge (v \, g \, w)$

For all $x \in |A|$, this mapping has the property:

ii) $(g \circ f)(x) = g(f(x))$

These two propositions show that information systems with domains as objects and approximable mappings as arrows satisfy the usual axioms for category theory (MacLane 1971). Therefore approximable mappings work as smoothly and naturally as a function space as any one could hope.

There is an important class of approximable mappings, the constant maps:

Definition 2.12 Constant Maps (Scott 1982)

Let A, B, C, D be information systems, let $b \in |B|$, there is a unique approximable mapping $K(b)\colon A \longrightarrow B$ such that:

i) $\forall x \in |A|, \quad K(b)(x) = b$

ii) $\forall f\colon B \longrightarrow C, \quad f \circ K(b) = K(f(b))$

iii) $\forall g\colon D \longrightarrow A, \quad K(b) \circ g = K(b)$

That mapping is induced by the binary relation over consistent sets of propositions:

iv) $u\, K(b)\, v \iff v \sqsubseteq b$

These constant maps define the usual constant functions from $|A|$ to $|B|$. Other useful functions like join and meet require multiple arguments. These can be brought into the category with the help of the product of information systems:

Definition 2.13 The Product of Information Systems (Scott 1982)

Let $A = \langle D_A, \Delta_A, Con_A, \vdash_A\rangle$ and $B = \langle D_B, \Delta_B, Con_B, \vdash_B\rangle$ *be information systems. We define* $A \times B = \langle D_{A\times B}, \Delta_{A\times B}, Con_{A\times B}, \vdash_{A\times B}\rangle$ *to be the product system as follow:*

i) $D_{A\times B} = \{\langle X, \Delta_B\rangle \mid X \in D_A\} \sqcup \{\langle \Delta_A, Y\rangle\rangle \mid Y \in D_B\}$

ii) $\Delta_{A\times B} = \langle \Delta_A, \Delta_B\rangle$

iii) $u \in Con_{A\times B} \iff u$ *is a finite subset of* $D_{A\times B}$ and $\{X \in D_A \mid \langle X, \Delta_B\rangle \in u\} \in Con_A$ and $\{Y \in D_B \mid \langle \Delta_A, Y\rangle \in u\} \in Con_B$

iv') $u \vdash_{A\times B} \langle X, \Delta_B\rangle \iff \{X \in D_A \mid \langle X, \Delta_B\rangle \in u\} \vdash_A X$

iv") $u \vdash_{A\times B} \langle \Delta_A, Y\rangle \iff \{Y \in D_A \mid \langle \Delta_A, Y\rangle \in u\} \vdash_B Y$

Because elements are identified with sets of statements about them, we ask what kind of statements are relevant when we talk about a pair of objects. One obvious answer is that such statements must talk about at least either the first or the second component of the pair. This is exactly what our definition does. An allowable statement is either of the form $\langle X, \Delta_B\rangle$ where X talks about the first component and Δ_B says trivial things about the second component or it is of the form $\langle \Delta_A, Y\rangle$ where Y talks about the second component while Δ_A says only trivial things about the first. We could have allowed compound statements of the form $\langle X, Y\rangle$ but these would have been functionally equivalent with the consistent set $\{\langle X, \Delta_B\rangle, \langle \Delta_A, Y\rangle\}$. There is no point in introducing such redundancy in our system. This justifies our definition of $D_{A\times B}$ in 2.13. The pair $\langle \Delta_A, \Delta_B\rangle$ is clearly the least informative pair of statements we could say about a pair of elements, so this is defined to be $\Delta_{A\times B}$. A set of pairs of propositions is consistent if and only if all propositions about the first component of the pair are consistent and all propositions about the second component are also consistent. This is exactly how $Con_{A\times B}$ is defined. The entailment relation $\vdash_{A\times B}$ is likewise defined in a point wise fashion. For instance $u \vdash_{A\times B} \langle X, \Delta\rangle$ iff X is entailed from propositions in u about the first component of the pair. Similarly $u \vdash_{A\times B} \langle \Delta_A, Y\rangle$ iff Y is entailed from the propositions in u about the second component of the pair. We can now see how $D_{A\times B}$ indeed behaves like a cartesian product of information systems.

Proposition 2.14 Pairing and Projection, Function Notation (Scott 1982)

If A and B are information systems, then so is $A \times B$ and there are two approximable mappings $fst\colon A\times B \longrightarrow A$ and $snd\colon A\times B \longrightarrow B$ such that for every approximable mappings $f\colon C \longrightarrow A$ and $g\colon C \longrightarrow B$, there is one and only one approximable mapping $\langle f, g\rangle\colon C \longrightarrow A\times B$ such that $fst \circ \langle f, g\rangle = f$ and $snd \circ \langle f, g\rangle = g$.

The mapping fst and snd are the usual projections that selects the first and second component of the pair respectively. The function $\langle *, *\rangle$ is the usual pairing operator. This proposition may be hard to understand for many people because, according to categorical usage, the notation talks about functions rather than about domain elements. Therefore we will translate the above result in more readable terms:

Definition 2.15 Pairing, Normal Notation (Scott 1982)

The ordered pair of two elements x and y belonging to the domains $|A|$ and $|B|$ respectively is defined as:

$$\langle x, y\rangle = \langle K(x), K(y)\rangle(Cl(\{\Delta\}))$$

for some suitably chosen Δ belonging to some conveniently fixed information system. The notation $f(x, y)$ will stand for $f(\langle x, y\rangle)$ for every function $f\colon A \times B \longrightarrow C$.

Note we have overloaded the $\langle x, y\rangle$ notation. In 2.14 it was applied to functions of type $A \to C$ and $B \to C$ respectively. In 2.15 it is applied to elements in A and B in the left hand side and it is applied on functions similarly to 2.14 on the right hand side. From now on the notation $\langle x, y\rangle$ will mean the pair of x and y unless explicitly stated.

Proposition 2.16 Properties of Projections and Pairings (Scott 1982)

For every $x \in |A|$ and $y \in |B|$, we can show:

i) $\langle x, y\rangle \in |A \times B| = \{\langle X, \Delta_B\rangle \mid X \in x\} \sqcup \{\langle \Delta_A, Y\rangle\} \mid Y \in y\}$

ii) $fst(x, y) = x$

iii) $snd(x, y) = y$

iv) $z = \langle fst(z), snd(z)\rangle, \quad \forall z \in |A \times B|$

Using the notation of 2.14:

v) $\langle f, g\rangle(t) = \langle f(t), g(t)\rangle, \quad \forall t \in |C|$

This should convince the reader our pairing and selection operators do indeed work as expected. We can generalize these to n-tuples the standard way using the notation $\langle x_1 \ldots x_n\rangle$ to form an n-tuple and the notation u_i to select the ith component of the n-tuple u. We will not give the details of the definitions. Readers willing to know more on the topic are referred to any standard text book on set theory or category theory.

With the help of product domain, we can now state the following:

Proposition 2.17 Meet and Join (Scott 1976, 1981)

Let A be an information system. There is an approximable mapping $M\colon A \times A \longrightarrow A$ such that $\forall x, y \in |A|$, $M(x, y) = x \sqcap y$. If in addition $\forall x, y\, \exists(x \sqcup y) \in |A|$, then there is an approximable mapping $J\colon A \times A \longrightarrow A$ such that $J(x, y) = x \sqcup y$.

Functions M and J are induced by the following binary relations over consistent sets of propositions:

i) $u\, M\, v \iff (fst(u) \sqcap snd(u)) \vdash v$

ii) $u\, J\, v \iff (fst(u) \sqcup snd(v)) \vdash v$

Let's summarize the categories we use or may need in the future.

Proposition 2.18 Useful Categories

There are three categories involving information systems, domains, approximable mappings and relations depending on what we chose for the objects and the arrows of the categories:

i) Information systems as objects and approximable mappings over consistent sets as arrows.

ii) Domains as objects and approximable functions over elements as arrows.

iii) Domains as objects and relations over elements arrows.

Category ii) is a proper subcategory of category iii).

The reader should not confuse categories i) and ii). Every information system induces a domain and every approximable mapping over consistent sets induces an approximable function over the domain elements. If we assume as is implicit in Scott's work (Scott 1982) that conversely every domain is induced by an information system and every approximable function is induced by an approximable mapping, then the two categories are isomorphic and the properties of one transports to the other. This is why they are so easily confused. We need both of them, category i) is used when we look at the meaning of the elements, category ii) is relevant when we look at computable functions over elements, and we continually work with both categories, implicitly transporting every result we find about one category to the other on the ground they are isomorphic. The risk of confusion is great but if the reader takes a little care, context will usually sort things out. Category iii) is introduced for later use, mainly to be able to formalize Cousot's notion of abstract semantics on categorical grounds. We shall return to this later.

2.3 Constructing Domains

This completes our discussion of what the functions allowed in Scott's category of approximable mappings are. Up to now, we have introduced the concept of Cousot's data flow framework $\langle P, D, F, I\rangle$. We have discussed what the lattices D and the function space F in the abstract are, that is we discussed in general terms what the elements of D and F are and how they are structured. However, data flow analyst needs more. He wants to know exactly what his lattice is, not only the general properties. The next step is to show some examples of simple useful lattices and some methods to build the more complex lattices from the more simple ones. For every such simple lattice or construction method, we will stress the relevant partial order, the meet and join operator and the special elements $\perp$ and $\top$, so that the reader will be aware of the details of the structures involved.

2.3.1 Simple Domains

We will first discuss some simple useful domains.

Definition 2.19 The One–Point Information System

The information system $\mathcal{ONE} = \langle D, \Delta, Con, \vdash\rangle$ *is defined as follow:* $D = \{\Delta\}$; $Con = \{\emptyset, \{\Delta\}\}$ *and the entailment relation is trivialy defined by the statements* $\{\Delta\} \vdash \Delta$ *and* $\emptyset \vdash \Delta$.

The domain $|\mathcal{ONE}|$ has exactly one element, namely $\{\Delta\}$, that happens to be trivialy the top and the bottom element. The partial order and the meet and join operations over $|\mathcal{ONE}|$ are likewise trivial. This domain is seldom useful by itself in data flow analysis. However it can be used in conjunction with disjoint unions to introduce a new distinct element in an existing domain.

Definition 2.20 The Two-point Information System

The information system $\mathcal{BIT} = (D, \Delta, Con, \vdash)$ *is defined as follow:* $D = \{\Delta, T\}$ *where* T *is some proposition distinct form* Δ; $Con = \{\emptyset, \{\Delta\}, \{T\}, \{\Delta, T\}\}$; *and* $u \vdash X \iff X = \Delta \vee T \in u$.

The domain $|\mathcal{BIT}|$ has two elements, $0 = \{\Delta\}$ and $1 = \{\Delta, T\}$. The partial order is the reflexive closure of $0 \sqsubseteq 1$. The bottom element is 0 and the top element is 1. This information system is often used in data flow analysis to represent the existence or the nonexistence of some evidence about a fact. The element 0 means there is no evidence and 1 means we have some evidence that a proposition is true. Obviously 1 is more informative that 0 because if we have no evidence that the proposition is true, as far as we know, it can be either false or true. Once evidence has been provided we know the proposition is true and not false. The more common kind of data flow analysis uses bit vectors. We want to collect information about the truth of a single predicate P when applied to a large range of data objects of type X. We compute a a suitably chosen function $f: X \longrightarrow \mathcal{BIT}$. When $f(x)$ is

1, we know the data flow analysis has uncovered some evidence that $P(x)$ is true. If $f(x)$ is 0 there is no such evidence. For some data flow problems it may be possible to infer from that lack of evidence that the statement $P(x)$ is false, but this is not always the case. It is up to the data flow analyst to figure out the correct interpretation of evidence or lack of evidence for his particular problem on a case by case basis.

Definition 2.21 Miscellaneous Flat Information Systems

Let S be a set, The information system $FLAT(S) = \langle D, \Delta, Con, \vdash\rangle$ is defined as follow: $D = \{\Delta\} \cup S$; the proposition Δ is chosen not to belong to S; the set $Con = \{\emptyset, \{\Delta\}\} \cup \{\{x\} \mid x \in S\} \cup \{\{\Delta, x\} \mid x \in S\}$; and $u \vdash X \iff X = \Delta \vee X \in u$.

The domain $|FLAT(S)|$ contains one bottom element $\{\Delta\}$ plus one element $\{\Delta, x\}$ for every element $x \in S$. There is no top element in $|FLAT(S)|$. The statement $x \sqsubseteq y$ is always false unless $x = \{\Delta\}$ or $x = y$. The meet $x \sqcap y$ of two elements is always $\{\Delta\}$ unless $x = y$ in which case the idempotence law applies. The join of two element $x \sqcup y$ never exists unless $x \sqsubseteq y$ or $y \sqsubseteq x$, in which case $x \sqcup y$ is the larger value. These flat domains corresponds to non structured domains like characters, integers, Pascal-like enumerated types and Boolean. The elements of these domains can only be fully specified or fully unspecified. They cannot have a nontrivial partial specification. Therefore if an element is left unspecified we have the value $\perp$, otherwise we have some fully defined element. Such flat domains are often generated by iteration of disjoint union over the one point domain in the literature.

2.3.2 Constructing Complex Domains

This completes the list of the more common simple domains. Building complex domains is done with the help of a category-theorical device called a functor. Usually a functor maps from one category to another. We will restrict ourself to functors that maps from one category to itself. This makes our definitions more restrictive than the usual ones.

Definition 2.22 Functors (MacLane 1971)

A covariant functor T is a map that associates every object A to the object $T(A)$ and associates to every arrow $f: A \longrightarrow B$ the arrow $T(f): T(A) \longrightarrow T(B)$. The map T is such that if $I: A \longrightarrow A$ the identity arrow over A, then the arrow $T(I): T(A) \longrightarrow T(A)$ is the identity arrow over $T(A)$ and for every two arrows f, g, the equation $T(f \circ g) = T(f) \circ T(g)$ holds.

A contravariant functor T is a map that associates to every object A the object $T(A)$ and that associates to every arrow $f: A \longrightarrow B$ the arrow $T(f): T(B) \longrightarrow T(A)$. The map T is such that if $I: A \longrightarrow A$ the identity arrow over A, then the arrow $T(I): T(A) \longrightarrow T(A)$ is the identity arrow over $T(A)$ and for every two arrows f, g, the equation $T(f \circ g) = T(g) \circ T(f)$ holds.

Definition 2.23 Bifunctors (MacLane 1971)

A bifunctor is a functor in two variables, that is a map T that associates to every pair A, B of objects a new object $T(A, B)$. In addition the bifunctor T is covariant in both variables if it associates to every pair of arrows $f: A \longrightarrow C$ and $g: B \longrightarrow D$ an arrow $T(f, g): T(A, B) \longrightarrow T(C, D)$ in such a way that the identity arrows $I_A: A \longrightarrow A$ and $I_B: B \longrightarrow B$ are mapped to the identity arrow $T(I_A, I_B): T(A, B) \longrightarrow T(A, B)$ and if all composable pairs $\langle f, g\rangle$ and $\langle k, h\rangle$ of arrows satisfy the equation $T(f \circ g, k \circ h) = T(f, k) \circ T(g, h)$.

The bifunctor T is covariant in its first variable and contravariant in its second variable if it associates to every pair of arrows $f: A \longrightarrow C$ and $g: D \longrightarrow B$ the new arrow $T(f, g): T(A, B) \longrightarrow T(C, D)$. Notice the contravariant arrow g has its direction reversed from $B \longrightarrow D$ to $D \longrightarrow B$ when compared with the covariant definition. The bifunctor T must in addition map the identity arrows $I_A: A \longrightarrow A$ and $Ib: B \longrightarrow B$ to the identity arrow $T(I_A, I_A): T(A, B) \longrightarrow T(A, B)$ and must satisfy the equation $T(f \circ g, h \circ k) = T(f, k) \circ T(g, h)$ for all composable pairs $\langle f, g\rangle$ and $\langle k, h\rangle$ of arrows. Notice that the composition of the contravariant arrows h and k is the reverse of the composition of the covariant arrows in the left hand side of the equation.

There are similar definitions for bifunctors contravariant in the first variable and covariant in the second variable and bifunctors that are contravariant in both variables. They are left to the reader.

If we remember that domains and approximable mappings are a category, we will agree that a functor can be used as a function that returns a domain value when given a domain argument. In this way it can be used to build complex domains out of more simple ones. Functors are also much more that that. They also transport the approximable functions to the newly constructed domains in a structure preserving way. In that sense the algebraic properties of the lattice theoretic operators are transported from the simple domains to the compound domains. Every time we introduce a new functor to build a new domain, we will also define what the bottom and top elements of the constructed domains will be by transporting the relevant constant maps with the help of the same functor. The meet and join operations will be similarly transported into the new domain. The partial order relation itself can be so transported. However the fact that $\sqsubseteq$ is not an approximable mapping requires either the functor to belong to the category of relations over domains or the partial order to be defined as the equivalence between $x \sqcup y = x$ and $y \sqsubseteq x$. This ability to keep track of the structures of the domains we constructs is of high interest in data flow analysis. Not only do we make our formalism sit on sound formal grounds, creating complex lattices with very highly abstract domain constructions, we are also able to go from the abstract definitions down to the lower level of details that are relevant to the particular implementation problems we may want to consider.

There is one pitfall to avoid when using functors in the way we describe. A functor F embeds the structure of a domain A into the domain $F(A)$. This means a part of $F(A)$ has the same structure as A but there may be portions of $F(A)$ that are structured differently. For instance if F is a functor such that for every element $y \in F(A)$, there is an element $x \in A$ such that the constant map $K(y) = F(K(x))$ then everything works fine because every element of $F(A)$ is the image of an element of A. The algebraic properties of A are transported to every element of $F(A)$ because of the identity $F(f \circ g) = F(f) \circ F(g)$. It is this property of functors that allow us to infer the lattice theoretic property of $F(A)$ from the corresponding properties of A. On the other hand if there is a $y \in F(A)$ such that $\forall x \in A,\ K(y) \neq F(K(x))$; then there is no identity that will use the algebraic properties of A to tell us how y fits in $F(A)$. The functor method allow us to infer the lattice theoretic properties of only a subdomain of $F(A)$. When a functor introduces new elements in the compound domain, we have to check what their lattice theoretic behavior is by other means.

The above method works fine with functions in one variable like constant maps. We also have to deal with functions in two variables like meet and join. The meet function is especially important since we plan to deal with the partial ordering $\sqsubseteq$ with the functor method by using the equivalent relation $x \sqcap y = x$. Fortunately we can apply the next theorem to reduce functions in two variables to functions in one variables.

Proposition 2.24 (MacLane 1971)

For every approximable function $f: A \times B \longrightarrow C$, there is a unique pair of approximable functions $f': A \longrightarrow B \longrightarrow C$ and $f'': B \longrightarrow A \longrightarrow C$ such that we have $\forall x \in A,\ y \in B,\quad f(x,y) = f'(x)(y) = f''(y)(x)$. Conversely if for every $x \in A$ and $y \in B$ there are functions $f': A \longrightarrow B \longrightarrow C$ and $f'': B \longrightarrow A \longrightarrow C$ such that $\forall x \in A, y \in B$, $f'(x)(y) = f''(y)(x)$, then there is a unique approximable function $f: A \times B \longrightarrow C$ such that

$$\forall x \in A, y \in B,\ f(x,y) = f'(x)(y) = f''(y)(x).$$

This completes our discussion of functors as a mean to construct complex domains. It is now time to look at some useful functors.

2.3.3 The *BH* Functor

We have said several times that the join of two elements in a domain does not always exist. This is most unfortunate because joining is a very frequent operation in data flow analysis. Our first functor will transform every domain into a complete lattice.

Definition 2.25 The BH Functor

Let $A = \langle D, \Delta, Con, \vdash\rangle$ be an information system. The information system $BH(A) = \langle D', \Delta', Con', \vdash'\rangle$ is defined as follow:

i) $D' = D$

ii) $\Delta' = \Delta$

iii) $Con' = \{u \mid u \text{ is finite and } u \subseteq D\}$

iv) $u \vdash' X \iff u \vdash X \vee u \notin Con$

The functor BH also maps every approximable mapping $f: A \longrightarrow B$ to the mapping $BH(f): BH(A) \longrightarrow BH(B)$ defined as follow:

v) $u\ BH(f)\ v \iff$ either $u \in Con \wedge (u\ \ f\ v)$ or *else* $u \notin Con$ and $v \sqsubseteq Cl(\bigcup\{v' \mid \exists u' \in Con,\quad u' \sqsubseteq u,\quad (u' f v')\})$

This functor BH just plugs a top element on any lattice that has no top. The set D of propositions and the least informative member Δ of that set are left unchanged by the functor. We introduce an element about which all propositions in D are going to be true, even normally contradictory ones. This element will be the top of the lattice. Every finite set of propositions will belong to Con' because for every such set, there is at least one element, namely $\top'$, that will satisfy all of them. Sets that are inconsistent in A entail every proposition in $BH(A)$ because only the top element will satisfy all of them. Any other proposition will be also true about $\top'$, therefore every proposition is logically derived from such an inconsistent set. Entailment is not changed for sets of propositions that are consistent in A because the set of elements that satisfies them is not changed except for the additional top element. Therefore the set of propositions that should be logically derived from a consistent set is not changed.

As long as we deal in $BH(A)$ with sets of propositions that are consistent in A, entailment in $BH(A)$ will be identical to entailment in A. Therefore as long as we use non–top elements of $|BH(A)|$, that domain is structurally identical to $|A|$. We may ask what the significance of $\top'$ is. Let's compare it with $\perp'$, the bottom element of $|BH(A)|$. There are no propositions that are true about $\perp'$ that are not true about every other element. Therefore $\perp'$ makes no significant information available. If we can bring more information about an element approximated as $\perp'$, we stop talking about $\perp'$ because significant information was brought in. On the other hand, every proposition is true about $\top'$, even if some are mutually contradictory. Therefore all propositions are equally insignificant when we talk about $\top'$. Like $\perp'$, the element $\top'$ makes no significant information available. Unlike $\perp'$, bringing more information about $\top'$ does not bring more significant information about an element because we already know that everything is trivially true. The element $\top'$ is a "black–hole of insignificance", an insignificant object that will void any supplemental information we may ever bring about it.

Some readers may ask why we may want to introduce inconsistency in such an explicit way and use it as if it were a valid object. This has to do with the very approximate nature of algorithms involved in data flow analysis. There are situations where available information gets irrevocably screwed up; for instance, we may be joining two non joinable elements because they occur as two inconsistent approximations of some consistent elements. Remember we talk about Cousot's notion of approximation here, not Scott's. When the case arises, the black–hole $\top'$ best describes the result of our computation, our algorithm is not good enough to discover significant information.

Now let's look at how the functor transports the lattice theoretic structure. Remark that if the domain A already has a top element, then every finite set of propositions would be consistent (Scott 1982). It is easy to check in this case that $BH(A) = A$ and for every function $f: A \longrightarrow B$, the function $BH(f): BH(A) \longrightarrow BH(B)$ is set theoretically identical to f. The functor BH does not change domains that already have a top element. The interesting case is when A does not have a top. It is immediate from the definition of BH that $BH(f)(x) = f(x)$ when $x \in |A|$. In other word BH doesn't change the behavior of a function when we do not use the additional top element. The next theorem will clarify the last remaining case, where A doesn't have a top element and we want to compute $BH(f)(\top')$.

Theorem 2.26

If $|A|$ is a domain and $f: A \longrightarrow B$ an approximable mapping over information systems A and B; if $\top$ is the top element of $|A|$ then the following identity is true:

$$BH(f)(\top) = \bigsqcup\{BH(f)(x) \mid x \in |BH(A)|\}$$

Proof:

Theorem 2.9 states that

$$(a) \qquad BH(f)(\top) = \bigsqcup\{Cl(v) \mid \exists u \sqsubseteq \top,\ u\ BH(f)\ v\}$$

Theorem 2.9 also states that

$$\bigsqcup\{BH(f)(x) \mid x \in |BH(A)|\} = \bigsqcup\{g(x) \mid x \in |BH(A)|\}$$

where $$g(x) = \bigsqcup\{Cl(v) \mid \exists u \sqsubseteq x,\ u\ BH(f)\ v\}$$

It turns out that the conjunction of $u \sqsubseteq x$ and $x \in |BH(A)|$ for some x is equivalent to $u \sqsubseteq \top$ for nothing prevents that x may be equal to $\top$. We can write:

$$(b) \qquad \bigsqcup\{BH(f)(x) \mid x \in |BH(A)|\} = \bigsqcup\{Cl(v) \mid \exists v \sqsubseteq \top,\ u\ BH(f)\ v\}$$

The right-hand side of (a) is identical to the right-hand side of (b), proving the theorem. ▮

We can now verify that BH behaves naturally. We can show from 2.26 that BH maps constant functions to constant functions, more exactly: $\forall x, y \in A$, $BH(K(x))(y) = K(x)(y)$. We can similarly show that for every x, $BH(\sqcap'(x))(\top) = x$ where $\sqcap'$ is the curried version of the meet of elements. Similarly we always have $BH(\sqcap''(x))(\top) = x$ where $\sqcap''$ is the curried version of $\sqcap$ with parameters reversed. We can invoke proposition 2.24 to construct the binary function $BH(\sqcap)$ that will behave as expected, in particular the partial ordering $x \sqsubseteq y$ is indeed equivalent to $x\ BH(\sqcap)\ y = x$. The join operation does not exist as an approximable relation in A unless $|A|$ has a top element; therefore we leave that operator out of the present discussion. These remarks suffice to show the functor BH preserves the lattice theoretic properties of the domains to which it is applied. This completes our discussion of the relation between the structures of A and $BH(A)$.

2.3.4 Cartesian Products

We have already introduced a bifunctor, namely the cartesian product. Let's look more closely at it.

Definition 2.27 The Cartesian Product Functor

The cartesian product $A \times B$ of information systems A and B is a bifunctor where $A \times B$ is defined as in 2.13 and for every pair of approximable mappings $f: A \longrightarrow C$ and $g: B \longrightarrow D$ the product mapping $f \times g: A \times B \longrightarrow C \times D$ is defined as $\langle u, v\rangle (f \times g) \langle x, y\rangle \iff (u \; f \; x) \wedge (v \; g \; y)$. This definition implies $(f \times g)(x, y) = \langle f(x), g(y)\rangle$.

The identity $K(u) \times K(v) = K(\langle u, v\rangle)$ shows that every element $\langle u, v\rangle$ is the image of a pair of elements u and v. Therefore the lattice theoretic structure of $A \times B$ can be totally inferred from the structures of A and B by the functor method.

Proposition 2.28

Let A and B be domains.

i) $\bot_{A \times B} = \langle \bot_A, \bot_B\rangle$

ii) $\top_{A \times B} = \langle \top_A, \top_B\rangle$ *whenever both* $\top_A$ *and* $\top_B$ *exist.*

iii) $\langle u, v\rangle \sqcap_{A \times B} \langle x, y\rangle = \langle u \sqcap_A x, v \sqcap_B y\rangle$

iv) $\langle u, v\rangle \sqcup_{A\times B} \langle x, y\rangle = \langle u \sqcup_A x, v \sqcup_B y\rangle$ *whenever join is defined on both A and B.*

v) $\langle u, v\rangle \sqsubseteq_{A\times B} \langle x, y\rangle \iff u \sqsubseteq_A x \wedge v \sqsubseteq_B y$.

Subscripts on lattice–theoretic operators refer to its corresponding lattice. This convention will be used in the rest of the thesis.

Putting 2.28 in English, the lattice theoretic operators are applied point wise on pairs in $|A \times B|$.

2.3.5 Disjoint Unions

Another useful functor is the disjoint union of information systems.

Definition 2.29 The Disjoint Union (or Separated Sum) of Information Systems (Scott 1982)

Let $A = \langle D_A, \Delta_A, Con_A, \vdash_A\rangle$ *and* $B = \langle D_B, \Delta_B, Con_B, \vdash_B\rangle$ *be two information systems. If* $*$ *is a proposition neither in* D_A *nor in* D_B *then the disjoint union* $A + B = \langle D_{A+B}, \Delta_{A+B}, Con_{A+B}, \vdash_{A+B}\rangle$ *of A and B is defined as follow:*

i) $D_{A+B} = \{\langle X, *\rangle \mid X \in D_A\} \cup \{\langle *, Y\rangle \mid Y \in D_B\} \cup \{\langle *, *\rangle\}$

ii) $\Delta_{A+B} = \langle *, *\rangle$

iii) $Con_{A+B} = \{\langle u, \emptyset\rangle \mid u \in Con_A\} \cup \{\langle \emptyset, v\rangle \mid v \in Con_B\}$

iv') $u \vdash_{A+B} \langle X, *\rangle \iff u \in Con_A, \wedge\, lft(u) \neq \emptyset \wedge lft(u) \vdash_A X$

iv") $u \vdash_{A+B} \langle *, Y\rangle \iff u \in Con_B \wedge rht(u) \neq \emptyset \wedge rht(u) \vdash_B Y$

iv"') $u \vdash_{A+B} \langle *, *\rangle$ always holds.

where the functions lft and rht are defined as follow:

$lft(u) = \{X \in D_A \mid \langle X, * \rangle \in u\}$

$rht(u) = \{Y \in D_B \mid \langle *, Y \rangle \in u\}$

For two approximable mappings $f\colon A \longrightarrow C$ and $g\colon B \longrightarrow D$, the disjoint union $f + g\colon A + B \longrightarrow C + D$ of the mapping is defined as follow:

$$\begin{aligned} u\,(f+g)\,v \iff & \text{ either } u \in A,\ v \in C \text{ and } u\ f\ v; \\ & \text{ or } u \in B,\ v \in D \text{ and } u\ g\ v; \\ & \text{ or } v = \langle *, * \rangle \end{aligned}$$

For the sake of readability, most of the time we omit explicitly writing the injections of the components into the disjoint unions.

We use the information system $A + B$ when we want to talk about elements that belong to either the domain $|A|$ or the domain $|B|$ but not both. We have to introduce a proposition $*$ that means that either Δ_A or Δ_B is true, or any other similarly trivial proposition that is true about all elements in A and all elements in B but is not already in D_A or D_B. The proposition $*$ will be used a the new least informative proposition because Δ_A and Δ_B implicitly convey the information that we are talking about an element in $|A|$ or $|B|$ respectively. Therefore both Δ_A and Δ_B conveys significant information about the elements of $|A + B|$, namely they identify the component of the sum where the elements belong. Ideally the propositions in D_{A+B} should be the sum of the propositions in D_A and in D_B plus the newly introduced $*$. This does not build a proper separated sum because in the construction of sums like $A+A$ the sets D_A and D_B are identical and the sum is no longer separated. We fix the problem with a coding trick: every proposition about an element in $|A+B|$ will be represented by an ordered pair of propositions where at least one component must be $*$. The order where the $*$ and the significant proposition appear is used to separate the component of the sum. The least significant proposition is represented by $(*,*)$. The set Con_A is constructed to insure that a consistent set of propositions in $A+B$ provides consistent information about an element of at most one component of the sum. The entailment relation $\vdash_{A+B}$ preserves the separation between the relations $\vdash_A$ and $\vdash_B$ in both components of the sum.

The sum of two functions naturally follows from the above considerations. The pair of functions $f\colon A \longrightarrow C$ and $g\colon B \longrightarrow D$ is summed into a function $f+g\colon A+B \longrightarrow C+D$ such that if x is an element that belongs either to the A or the B component, then $(f+g)(x)$ will be the element corresponding in $C+D$ to $f(x)$ or $g(y)$ respectively.

Proposition 2.30 Constructors and Selectors for Disjoint Unions (Scott 1976)

Let A and B be domains. There exists four approximable functions $lft\colon A+B \longrightarrow A$, $rht\colon A+B \longrightarrow B$, $inl\colon A \longrightarrow A+B$ and $inr\colon B \longrightarrow A+B$ such that:

i) $lft(inl(x)) = x, \qquad \forall x \in A$

ii) $inl(lft(x)) = x, \qquad \forall x \in A + B,\ x \neq \bot_{A+B}$

iii) $rht(inr(x)) = x, \qquad \forall x \in B$

iv) $inr(rht(x)) = x, \qquad \forall x \in A + B,\ x \neq \bot_{A+B}$

v) *For every* $f\colon A \longrightarrow C$ *and* $g\colon B \longrightarrow C$*; there is a unique function* $h\colon A + B \longrightarrow C$ *such that*

v') $h \circ inl = f$

v") $h \circ inr = g$

v"') $h(\bot_{A+B}) = \bot_C$

vi) *For every* $f\colon A \longrightarrow C$ *and* $g\colon B \longrightarrow D$ *we have*

vi') $(f + g)(inl(x)) = inl(f(x)), \qquad \forall x \in A$

vi") $(f + g)(inr(y)) = inr(g(y)), \qquad \forall y \in B$

vi"') $(f + g)(\bot_{A+B}) = \bot_{C+D}$

Inferring the lattice structure of $A + B$ using the functor method is not straightforward as it was for $A \times B$. The function $f + g$ where f and g are constant maps is not a constant map. For example $K(\bot_A) + K(\bot_B)$ will return either $inl(\bot_A)$, $inr(\bot_B)$ or $\bot_{A+B}$ depending on its argument. Similarly, $K(\top_A) + K(\top_B)$ will return either $inl(\top_A)$, $inr(\top_B)$ or $\bot_{A+B}$. If we work out functions in two arguments with the help of proposition 2.24, we will find out that functions like $\sqcap\colon (A + B) \times (A + B) \longrightarrow A + B$ expects to have both arguments either in the A component or in the B component of the sum, otherwise the result will be $\bot_{A+B}$.

Disjoint unions keep the lattice structure of both components separated, adding only a common bottom element.

Proposition 2.31 The Partial Order Structure of Disjoint Unions

Let A and B be domains, then the following is true:

i) $\bot_{A+B} = Cl(\langle *, * \rangle)$ *where $*$ is the special proposition defined in 2.29.*

ii) There is no $\top_{A+B}$.

iii) The meet operation satisfies all of the following:

iii') $inl(x) \sqcap_{A+B} inl(y) = inl(x \sqcap_A y)$ when $x, y \in A$.

iii") $inr(x) \sqcap_{A+B} inr(y) = inr(x \sqcap_B y)$ when $x, y \in B$.

iii"') $x \sqcap_{A+B} y = \bot_{A+B}$ when x and y are not both in A or both in B.

iv There is no join operation in $A + B$.

v) The $\sqsubseteq_{A+B}$ satisfies all of the following:

v') $inl(x) \sqsubseteq_{A+B} inl(y)$ if $x \sqsubseteq_A y$ when $x, y \in A$.

v") $inr(x) \sqsubseteq_{A+B} inr(y)$ if $x \sqsubseteq_B y$ when $x, y \in B$.

v"') $\bot(A + B) \sqsubseteq_{A+B} y$ is always true.

v"") $x \sqsubseteq_{A+B} y$ is false in every other case.

vi) If both A and B supports a join operation, there is a quasi-join operation, also denoted by $\sqcup_{A+B}$, that satisfies:

vi') $inl(x) \sqcup_{A+B} inl(y) = inl(x \sqcup_A y)$ when $x, y \in A$.

vi") $inr(x) \sqcup_{A+B} inr(y) = inr(x \sqcup_B y)$ when $x, y \in B$.

vi"') $x \sqcup_{A+B} y = \bot_{A+B}$ when neither both $x, y \in A$ nor both $x, y \in B$.

This quasi-join behave like a join provided we stay in the same component of the sum. It is the function $\sqcap_A + \sqcap_B$ constructed by the functor.

We can find in (Stoy 1977) and in (Sco 1976) another version of disjoint union that has a top element and a join operation. It is similar to the disjoint union given in 2.29 except we introduce an aditional $\top$ element in a way similar to what $BH(A+B)$ would do. The details of the definitions are left to the reader's imagination.

2.3.6 Function Spaces

The next functor we investigate will construct function spaces.

Definition 2.32 The Function Space Information System (Scott 1982)

Let $A = \langle D_A, \Delta_A, Con_A, \vdash_A\rangle$ and $B = \langle D_B, \Delta_B, Con_B, \vdash_B\rangle$ be information systems. The function space $A \longrightarrow B = \langle D_{A\longrightarrow B}, \Delta_{A\longrightarrow B}, Con_{A\longrightarrow B}, \vdash_{A\longrightarrow B}\rangle$ is the information system defined as follow:

i) $D_{A\longrightarrow B} = \{\langle u, v\rangle \mid u \in Con_A \ \wedge \ v \in Con_B\}$

ii) $\Delta_{A\longrightarrow B} = \langle \emptyset, \emptyset\rangle$

Let $w = \{\langle u_0, v_0\rangle \ldots \langle u_n, v_n\rangle\}$

iii) $w \in Con_{A\longrightarrow B} \iff \left(\bigcup\{u_i \mid i \leq n\} \in Con_A \Longrightarrow \bigcup\{v_i \mid i \leq n\} \in Con_B\right)$

iv) $w \vdash_{A\longrightarrow B} \langle u', v'\rangle \iff \bigcup\{v_i | u' \vdash_A u_i\} \vdash_B v'$

Also let $f: C \longrightarrow A$ and $g: B \longrightarrow D$ be approximable mappings, then the function $f \longrightarrow g: (A \longrightarrow B) \longrightarrow (C \longrightarrow D)$ is defined as follow:

Let $w = \{\langle u_0, v_0\rangle \ldots \langle u_n, v_n\rangle\} \in A \longrightarrow B$

Let $z = \{\langle x_0, y_0\rangle \ldots \langle x_m, y_m\rangle\} \in C \longrightarrow D$

$w\,(f \longrightarrow g)\,z \iff \forall i \sqsubseteq m,\ \exists j \sqsubseteq n,\quad x_i\, f\, u_j \,\wedge\, v_j\, g\, y_i$

The domain $|A \longrightarrow B|$ has as its elements the set of all approximable mappings from A to B. The propositions of $A \longrightarrow B$ are the statements that $u\ f\ v$ for some element $f \in |A \longrightarrow B|$ and consistent sets of propositions $u \in A$ and $v \in B$ respectively. These propositions are represented by pairs $\langle u, v\rangle$ of set of propositions in A and B respectively, meaning that $u\ f\ v$ for some $f \in |A \longrightarrow B|$. The least informative pair is $\langle \emptyset, \emptyset\rangle$ because $\emptyset\ f\ \emptyset$ is trivialy true for every f. Consistent set of pairs maps consistent sets of propositions to consistent set of propositions. Entailment is best understood when we remember that an approximable mapping can be viewed as a way to deduce information about the result of a function given information about the argument. Entailment simply defines the required transitivity property of such inference.

Refer back to definition 2.11. Given three functions $f: C \longrightarrow A$, $g: B \longrightarrow D$ and $h: A \longrightarrow B$, we have the following identity: $(f \longrightarrow g)(h) = g \circ h \circ f$. This gives the significance of the functor $\longrightarrow$ when applied to mappings. The function $(f \longrightarrow g)$ composes its argument $h: A \longrightarrow B$ with f and g to get a function of type

$C \longrightarrow D$. This result is significant enough to be given a reference number:

Proposition 2.33

Let the approximable mappings $f: C \longrightarrow A$, $g: B \longrightarrow D$ *and* $h: A \longrightarrow B$. *The following identity hold:*

$$(f \longrightarrow g)(h) = g \circ h \circ f$$

We will look now at the lattice theoretic structure of function spaces. What is the image of constant maps under the functor? By 2.12.iii and 2.33 we have $(K(x) \longrightarrow K(y)) = K(K(y))$. In English, we can state equivalently that the image of constant maps is a function that send every approximable mappings in $A \longrightarrow B$ to the constant map $K(y): C \longrightarrow D$. In other words the functor method does not determine the structure of the whole function space. It gives only the structure of the subspace of all constant maps. In addition, the fact that the value of $(K(x) \longrightarrow K(y))$ does not depend on $K(x)$ hints that the structure of $C \longrightarrow D$ depends only on the structure of D, it is independent of the structure of C. The facts are summarized below:

Proposition 2.34 The Lattice Theoretic Structure of Function Spaces (Scott 1976, Stoy 1977)

Let A and B be information systems and $f, g: A \longrightarrow B$ be approximable mappings, then

i) $\bot_{A \longrightarrow B}(x) = \bot_B \quad \forall x \in A.$

ii) $\top_{A \longrightarrow B}(x)$ *exists iff* $\top_B$ *exists. In this case,* $\top_{A \longrightarrow B}(x) = \top_B$ *for every* $x \in A.$

iii) $(f \sqcap_{A \longrightarrow B} g)(x) = f(x) \sqcap_B g(x) \quad \forall x \in A.$

iv) The join over $A \longrightarrow B$ *exists provided it exists over* B. *In this case,* $(f \sqcup_{A \longrightarrow B} g)(x) = f(x) \sqcup_B g(x)$ *for every* $x \in A.$

v) $f \sqsubseteq_{A \longrightarrow B} g \iff f(x) \sqsubseteq_B g(x) \quad \forall x \in A.$

These results are easily obtained for the subspace of all constant maps by the functor method. If we want to generalize them to the whole function space, observe that i) trough iv) can be obtained from v) by working out the lattice theoretic definition of the relevant operator. We can find that v) is true in (Stoy 1977) and in (Scott 1976).

Before we leave this topic, we mention a very important result regarding function spaces:

Proposition 2.35 The Least Fixed Point Theorem (Scott 1976, 1981, 1982; Stoy 1977)

Let A be an information system. There is a unique approximable mapping $\mathit{fix}: (A \longrightarrow A) \longrightarrow A$ *such that:*

i) $f(fix(f)) = f$

ii) $fix(f) \sqsubseteq h(f)$ *for every* h *such that* $f(h(f)) = h(f)$

That operator fix satisfies the very important equality:

iii) $fix(f) = \bigsqcup_{i=0}^{\infty}\{f^i(\perp_A)\}$

where $f^0 = I$ *the identity over* A, *and* $f^{i+1} = f \circ f^i$. *It is shown in (Scott 1982) that this use of the join always gives an existing element in any information system, even when join is only partially defined on* A.

This completes our discussion of function spaces. This also completes our discussion of the elementary functors available to the data flow analyst. The reader should remember equation 2.35.iii. It will play a central role when we will consider some termination problems. Now we will proceed to show how functors can be composed together into equations to make other functors.

2.3.7 Recursive Domains

The last question about domain construction to discuss is the definition of domains by means of the so–called domain equations. What we want to know is the lattice theoretic structure of the solution of such equations. The author does not know of a published formal discussion of that topic given in terms of information systems. However there are many papers (Scott 1976, 1981; Stoy 1977) that discuss the problem in terms of domains. The principal results are summarized below.

Definition 2.36 Retracts

An approximable function $f: A \longrightarrow A$ is a retract over the domain A if and only if it is idempotent under composition , that is it satisfies $f \circ f = f$.

Proposition 2.37 Retracts as Fixed Points

If the function $h: (A \longrightarrow A) \longrightarrow (A \longrightarrow A)$ is an approximable function that maps retracts to retracts, then the fixed point $\mathit{fix}(h)$ is also a retract.

Proposition 2.38 Domains Defined with Retracts

Let $f: A \longrightarrow A$ be a retract over A. The set $\{x \mid \exists y \in A, \quad x = f(y)\}$ is a domain.

Proposition 2.39 The Universal Domain

There is a domain V such that every countably based domain (i.e. domain with countably many finite elements) is isomorphic to a subdomain of V. In addition, for every one of the functors F discussed above, there is an approximable function that maps the retracts generating the domains A and B to the retract generating the domain $F(A, B)$ (or $F(A)$ in the case of functors in one variable).

According to 2.38 retracts are a method of generating new domains from existing ones. Such subdomains are in fact subsets of existing domains and the lattice theoretic structure of the new domain is inherited from the old one as suggested by the inclusion relation. Proposition 2.39 defines a domain V such that every other domain D we could be interested into is a subdomain of V. In addition all the familiar functors turn out to be approximable functions mapping retracts over V to retracts over V. Therefore, we can use the fixed point operator of 2.35 to find the solution of fixed point equations involving domains and functors. This operator computes a new retract that will determine the required subdomain of V.

What are the lattice theoretic properties of the solutions of domain equations? We will not provide a general answer to the question. Remark that for every functor we have introduced, there is a set of equations (or equivalences) that relates the lattice theoretic operators on the domain arguments of the functor to the same operators on its domain value. If we work out these relating equations according to the domain equation we try to solve, we would get a set of recursive equations whose solutions are the lattice theoretic operators on the recursive domain. Let's look at an example.

Define $T = B + (T \times T)$ where B is some well defined domain. The elements of domain T are the binary trees having elements of B as leaves. If we work out the lattice theoretic properties of disjoint union and cartesian products according to that equation we get:

a) $\bot_T$ is a newly generated element not belonging to B nor to $T \times T$.

b) $\top_T$ does not exist.

c) $x \sqsubseteq_T y \iff x = \bot_T$ or $x = inl(u)$, $y = inl(v)$, $u \sqsubseteq_B v \in B$ or $x = inr(u)$, $y = inr(v)$, $u \sqsubseteq_{T \times T} v \in T \times T$.

In addition $u \sqsubseteq_{T \times T} v \in T \times T \iff \exists a, b, c, d \in T$, $u = \langle a, b\rangle$ and $v = \langle c, d\rangle$

and $a \sqsubseteq_T c$ and $b \sqsubseteq_T d$.

d) The meet $\sqcap_T$ is defined as:

$$inl(x) \sqcap_T inl(y) = inl(x \sqcap_B y), \qquad x, y \in B.$$

$$inr(x) \sqcap_T inr(y) = inr(x \sqcap_{T \times T} y), \qquad x, y \in T + T.$$

$$x \sqcap_T y = \bot_T \quad \text{in all other cases}$$

where $\sqcap_{T \times T}$ is defined as

$$\langle a, b \rangle \sqcap_{T \times T} \langle c, d \rangle = \langle a \sqcap_T b, c \sqcap_T d \rangle$$

e) There is no join defined over T but there is a quasi join defines as follow:

$$inl(x) \sqcup_T inl(y) = inl(x \sqcup_B y)$$

$$inr(x) \sqcup_T inr(y) = inr(x \sqcup_{T \times T} y)$$

$$x \sqcup_T y = \bot_T \text{ in all other cases}$$

where $\sqcup_{T \times T}$ is defined as

$$\langle a, b \rangle \sqcup_{T \times T} \langle c, d \rangle = \langle a \sqcup_T b, c \sqcup_T d \rangle$$

Notice how $\sqcap_T$, $\sqcup_T$ and $\sqsubseteq_T$ are defined recursively in term of themselves. Such definitions can be similarly obtained from any domain equation.

This completes our discussion of how to build lattices acceptable for data flow analysis.

2.4 Syntax, Semantics and Abstractions

We have not yet discussed two elements of the tuple $\langle P, D, F, I \rangle$ that constitutes a Cousot's framework for data flow analysis: the lattice of program points

P and the interpretation function I. Scott (1976) has already remarked that the context-free grammar of a language may be used to define the language as a domain. For example, the abstract syntax of lambda-calculus: (omitting delimiters like λ and parentheses)

$\mathbf{Ex} \rightarrow \mathbf{Var}$

$\mathbf{Ex} \rightarrow \mathbf{Var}\ \mathbf{Ex}$

$\mathbf{Ex} \rightarrow \mathbf{Ex}\ \mathbf{Ex}$

can be transformed into the domain equation:

$$\mathbf{Ex} = \mathbf{Var} + (\mathbf{Var} \times \mathbf{Ex}) + (\mathbf{Ex} \times \mathbf{Ex})$$

that recursively defines the domain **Ex** in terms of the domain **Var** of all variables. In general if we associate to every terminal symbol in a context-free grammar a well defined domain and to every non terminal a recursively defined domain, we can transform the grammar into a set of domain equations by replacing concatenation of symbols by cartesian products of the corresponding domains in every production and alternation of productions by the disjoint unions of the right-hand sides. In that sense every grammar defines a domain. We can without loss of generality assume that the lattice P of program points is such a domain.

Now what can we say about the interpretation function I? Suppose that $\mathbf{D}$ is some semantic domain of interest, suppose we have two functions, an injection $inj: (\mathbf{D} \longrightarrow \mathbf{D}) \longrightarrow \mathbf{D}$ and a projection $pro: \mathbf{D} \longrightarrow (\mathbf{D} \longrightarrow \mathbf{D})$ such that the composition $inj \circ pro$ is the identity over $\mathbf{D} \longrightarrow \mathbf{D}$, we can write the semantics:

I has type $\mathbf{Ex} \longrightarrow (\mathbf{Var} \longrightarrow \mathbf{D}) \longrightarrow \mathbf{D}$

$I[\![\mathtt{v}]\!](e) = e[\![\mathtt{v}]\!]$

$I[\![\lambda\mathtt{v}.\mathtt{ex}]\!](e) = inj(\lambda x.I[\![\mathtt{ex}]\!](e[\mathtt{v} := x]))$

$I[\![\mathtt{ex1(ex2)}]\!](e) = pro(I[\![\mathtt{ex1}]\!](e)\ (I[\![\mathtt{ex2}]\!](e)))$

We use the convention that when we write semantics equations, domains are written in **boldface** and syntactic elements are written in `typewriter` fonts. If a syntactic element must be enclosed withtin brackets, we use [\![`double square brackets`]\!] as a form of quoting. These brackets are not otherwise different from normal ones. The notation $e[\mathtt{v} := x]$ means $e[\mathtt{v} := x][\![\mathtt{w}]\!]$ is x when $\mathtt{w} = \mathtt{x}$ and $e[\![\mathtt{w}]\!]$ otherwise.

We readily see that we have a functor $P(\mathbf{D})$ that maps the domain $\mathbf{D}$ to the domain $\mathbf{Ex} \longrightarrow (\mathbf{Var} \longrightarrow \mathbf{D}) \longrightarrow \mathbf{D}$ such that for every $\mathbf{D}$ having the required *inj* and *pro* functions, there is a function $I: P(\mathbf{D}) \longrightarrow \mathbf{D}$ that defines a properly typed semantics for lambda-calculus. If $\mathbf{D}$ is defined by the equation $\mathbf{D} = \mathbf{D} \longrightarrow \mathbf{D}$ and both the *inj* and *pro* functions are the identities over $\mathbf{D}$, we have the standard semantics of lambda-calculus. If $\mathbf{D}$ is chosen to satisfy $\mathbf{D} = \mathbf{A} + (\mathbf{D} \longrightarrow \mathbf{D})$ where $\mathbf{A}$ is a domain of atomic values that are not functions, for instance integers and lists, and if the function *inj* is *inr* and the function *pro* is *rht*, we have a semantics of lambda-calculus with atoms. This intuitive notion of semantics is formalized in a categorical way as follow:

Definition 2.40 T-algebras (MacLane 1971)

Let T be a functor. A T-algebra is a domain **D** *with an approximable function* $I{:}\,T(\mathbf{D}) \longrightarrow \mathbf{D}$.

In the above lambda-calculus semantics, $T(\mathbf{D}) = \mathbf{Ex} \times (\mathbf{Var} \longrightarrow \mathbf{D})$.

Definition 2.41 Cousot's Framework for Data Flow Analysis Revisited

A Cousot data flow analysis framework is a tuple $\langle P, D, I\rangle$ *such that P is a functor in the category of domains with approximable functions, D is a domain in that category and I is an approximable function such that D and I form a P-algebra.*

We can immediately see that when we make the proper identifications of Cousot's lattices with Scott's domains and Cousot's distributive functions with Scott's approximable functions, the notion of a Cousot framework identifies with the notion of T-algebra. This result is perfectly correct because T-algebras are the categorical formalization of the notion of semantics and the purpose of Cousot's frameworks is to define what kind of semantics are relevant to data flow analysis. What definition 2.41 says is that any semantics defined in the category of domains with approximable functions is acceptable for data flow analysis purposes. From now on, the following terms will be considered synonymous: Cousot's framework for data flow analysis, semantics of programming languages, T-algebras in the category of domains with approximable functions. When we use the $\langle P, D, I\rangle$ notation to define a Cousot framework, the P stands for the functor T of the T–algebra and the I is the approximable function of definition 2.40. This is the meaning of definition 2.41.

2.4.1 Approximations for Data Flow Analysis

At this point the reader may ask what is the distinction between denotational semantics and data flow analysis. The answer is that data flow analysis involves more than just defining what a language means. We want to relate the standard denotational semantics to an abstract semantics that will give approximate information about the standard language meaning.

Let's start with an example. It has nothing to do with the usual problems of data flow analysis but it will illustrate the basic concepts of abstraction. Assume we have a language, call it INT that deals with integers. We need an integer domain that is organized as a flat lattice. All integers cannot be compared. We have in addition a bottom and a top element. However our computer cannot handle integers larger than, say, 16 bits long. The maximum integer allowed is 65535. Any implementation of INT on that computer doesn't fulfill the semantics because not all integers are correctly handled. We want to know what semantics is actually implemented.

We need an other domain, let's call it 16–bit, also organized as a flat lattice, that contains a bottom element, a top element and all integers from 0 to 65535. We define an abstraction function that maps integers to 16–bits as follow: $\bot$ is mapped to $\perp$, $\top$ is mapped to $\top$, integers from 0 to 65535 are mapped to the corresponding 16–bits number, and larger integers are all mapped to top. Here we use the top element of 16–bits as an overflow indicator. It means computer limitations have screwed up the computation. All arithmetic operations must be defined in 16–bits consistently with that interpretation. Expressions such as 2×60000 yield the result $\top$ because of the overflow. Other expressions such as $65536 - 1$ also yield the result $\top$ because in 16–bits this is equivalent to $\top - 1$. On the other hand, 16–bits operators will give accurate results as long as we stay in the allowable range.

We need a formal definition of "correct results" when we talk about 16–bits integers rather than integers. We define a concretization function c: *16–bits* $\longrightarrow$ **integers** that is a sort of inverse of abstraction. The bottom element is mapped to $\perp$, $\top$ is mapped to $\top$ and the 16–bits numbers 0 to 65535 are mapped to the corresponding integers. There is no way to concretize a 16–bits value to an integer larger than 65535. We say a 16–bits calculation is accurate if and only if the concretization of its result is larger, in a lattice theoretical sense, that the corresponding integer calculation. Flatness of the lattice of integers insures that if the integer computation is well defined, i.e. does not yield the result $\perp$, then value of the corresponding 16–bits computation will concretize either to $\top$ or to the integer value. This particular example has a stricter requirement for correctness than needed in general. We want that all integer computations giving the bottom value also give $\perp$ when done in 16–bits. We will see later in this work other examples where preserving bottom values across abstractions is not desirable. For the sake of generality, we do not include constraints on how to handle $\perp$ in our requirements on the "correctness" of an abstraction.

We have to put together three things to have an abstract semantics: an abstraction relation, a concretization function and some requirements on the "correctness" of the abstract result. This is formalized below.

Definition 2.42 Abstract Semantics, Approximate Semantics and Representations

Let $S = \langle P, D, I\rangle$ and $S' = \langle P', D', I'\rangle$ be two semantics. An abstraction from S to S' is a pair $\langle a, c\rangle$ where a is a relation on $D \times D'$ called the abstraction relation and $c\colon D' \longrightarrow D$ is a function called the concretization function. Notice the word "abstraction" has two uses in this definition. An abstraction is an increasing approximation when the pair $\langle a, c\rangle$ is such that all three of the following conditions hold:

i) $c(x)\ a\ x$ (every $x \in D'$ is the abstraction of its concretization.)

ii) if $x\ a\ y$ then $x \sqsubseteq c(y)$ (every x is less than the concretization of its abstraction.)

iii) There is a relation r on $D \times D$ such that $x\ r\ y$ implies that $x \sqsubseteq y$ and r is such that the following diagram commutes in the category of domains with relations:

$$\begin{tikzcd}
P(D') \arrow[rr, "I'"] & & D' \arrow[d, "c"] \\
P(D) \arrow[u, "P(a)"] \arrow[r, "I"] & D \arrow[r, dashed, "r"] & D
\end{tikzcd}$$

A decreasing abstraction from S to S' is defined by reversing the partial order in clauses ii) and iii) above. A representation from S to S' is defined taking equality instead of a partial order in ii) and iii) above. Obviously a representation is both a decreasing and an increasing approximation and the relation r introduced in clause iii) is the identity over D.

We will say that S' is an increasing approximation, decreasing approximation or representation of S to mean that there is a pair (a, c) that is an increasing approximation, decreasing approximation or representation respectively.

The difficulty with relating two semantics such as S and S' in 2.42 is that there is no ordering that relates the elements of D and D', therefore we cannot state that an element of D' approximates an element of D in a direct way. The definition of abstractions as stated in 2.42 solves that problem. It works as follows.

We assume that the interpretation function I is too hard to compute to be practical when evaluating a program in $P(D)$. We chose instead to evaluate the easier function I' to evaluate an abstract version of the program in the domain $P(D')$. The purpose of the abstraction relation a is to define a set of valid abstract values in D' for the corresponding values in D. Thanks to the functor notation, the relation $P(a)$ extends this abstraction concept to the domains $P(D')$ and $P(D)$ in the obviously natural way. The relation a must be a relation and not a function because a deterministic abstraction operation may be context dependent and the context dependencies do not appear in this definition. Once I' is applied to a suitably chosen value in $P(D')$ and evaluation has completed we have a result in the domain D'. We want to know what this does correspond to when we talk about elements of D. The concretization function c provide the translation from elements in D' to elements in D. The relation r is the distortion the successive operations of abstraction and concretization may introduce in the final value when we compare with the more direct evaluation with the help of function I. If there is no distortion, i.e. r is the identity, S' is a representation of S. If the distortion introduce more information, S' is an increasing approximation of S and conversely if the distortion loose information, S' is a decreasing approximation of S.

We should stress that if $\langle a, c\rangle$ is a representation, then clauses 2.42.i and 2.42.ii assert that a is the inverse relation of c. This implies that c builds an homomorphism from S' to S in the categorical sense. This is consistent with the interpretation of representation as "distortion free" abstract evaluation.

2.4.2 The Categories of Abstract Semantics

We would like to compose abstractions, that is if $S = \langle P, D, I\rangle$, $S' = \langle P', D', I'\rangle$ and $S'' = \langle P'', D'', I''\rangle$ are semantics, then if S' is an abstraction of

S and S'' is an abstraction of S', we want to conclude transitively that S'' is an abstraction of S. We want to show that abstractions are the arrows of a category.

Definition 2.43 Composition of Abstractions

Let $S = \langle P, D, I\rangle$, $S' = \langle P', D', I'\rangle$ and $S'' = \langle P'', D'', I''\rangle$ be semantics. Let $\langle a, c\rangle$ be an approximation from S to S' and $\langle a', c'\rangle$ be an approximation from S' to S''. The composition of abstractions $\langle a', c'\rangle \circ \langle a, c\rangle$ is the abstraction $\langle a'', c''\rangle$ defined as follow:

i) $a'' = a' \circ a$

ii) $c'' = c \circ c'$

This definition is summarised in the diagrams below:

$$\begin{tikzcd}
P''(D'') \arrow[rrr, "I''"] & & & D'' \arrow[d, "c'"] \\
P'(D') \arrow[u, "P(a')"] \arrow[rr, "I'"] & & D' \arrow[r, dashed, "r'"] \arrow[d, "c"] & d' \arrow[d, "c"] \\
P(D) \arrow[u, "P(a)"] \arrow[r, "I"] & D \arrow[r, dashed, "r"] & D \arrow[r, dashed, "k"] & D
\end{tikzcd}$$

Here relation k is defined as $c(x)\ k\ c(y) \iff x\ r'\ y$. Remember that a functor such as P always obeys the identity $P(a) \circ P(a') = P(a \circ a')$. Notice that we still do not know whether the compositions of representations or approximations are representations or a approximations respectively.

Theorem 2.44 The Identity Abstraction

Let $S = \langle P, D, I \rangle$ be a semantics, then the abstraction $i_S = \langle i', i \rangle$ where i' is the identity relation over $P(D)$ and i is the identity relation over D is an abstraction from S to S. In addition for every semantic $S' = \langle P', D', I'' \rangle$, if $\langle a, c \rangle$ is an abstraction from S to S' we have $i_{S'} \circ \langle a, c \rangle = \langle a, c \rangle$ and $\langle a, c \rangle \circ i_S = \langle a, c \rangle$.

This is immediate from the diagrams:

$$\begin{tikzcd}
P'(D') \arrow[rrr, "I'"] & & & D' \arrow[d, "i"] \\
P'(D') \arrow[u, "P(i')"] \arrow[rr, "I'"] & & D' \arrow[r, dashed, "i"] \arrow[d, "c"] & D' \arrow[d, "c"] \\
P(D) \arrow[u, "P(a)"] \arrow[r, "I"] & D \arrow[r, dashed, "r"] & D \arrow[r, dashed, "i"] & D
\end{tikzcd}$$

The preceding diagram establishes $i_{S'} \circ \langle a, c \rangle = \langle a, c \rangle$. The next diagram establishes that $\langle a, c \rangle \circ i_S = \langle a, c \rangle$.

$$\begin{tikzcd}
P'(D') \arrow[rrr, "I'"] & & & D' \arrow[d, "c"] \\
P(D) \arrow[u, "P(a)"] \arrow[rr, "I"] & & D \arrow[r, dashed, "r"] \arrow[d, "i"] & D \arrow[d, "i"] \\
P(D) \arrow[u, "P(i)"] \arrow[r, "I'"] & D \arrow[r, dashed, "i"] & D \arrow[r, dashed, "r"] & D
\end{tikzcd}$$

This completes the proof of 2.44. ∎

Theorem 2.45

The composition of abstraction is associative.

Proof: Immediate from the diagram:

$$\begin{tikzcd}
P'''(D''') \arrow[rrrr, "I'''"] & & & & D''' \arrow[d, "c''"] \\
P''(D'') \arrow[u, "P(a'')"] \arrow[rrr, "I''"] & & & D'' \arrow[r, dashed, "r''"] \arrow[d, "c'"] & D'' \arrow[d, "c'"] \\
P'(D') \arrow[u, "P(a')"] \arrow[rr, "I'"] & & D' \arrow[r, dashed, "r'"] \arrow[d, "c"] & D' \arrow[r, dashed, "k'"] \arrow[d, "c"] & D' \arrow[d, "c"] \\
P(D) \arrow[u, "P(a)"] \arrow[r, "I"] & D \arrow[r, dashed, "r"] & D \arrow[r, dashed, "k"] & D \arrow[r, dashed, "k''"] & D
\end{tikzcd}$$

we have the identities

$$(\langle a'',c''\rangle \circ \langle a',c'\rangle) \circ \langle a,c\rangle = \langle a''',c'''\rangle$$

$$\langle a'',c''\rangle \circ (\langle a',c'\rangle \circ \langle a,c\rangle) = \langle a''',c'''\rangle$$

where $a''' = a'' \circ a' \circ a$

$c''' = c \circ c' \circ c''$

We readily see this is indeed the associativity of composition of abtractions. ▮

The above results shows there is a category with semantics (or data flow frameworks or T-algebras, all these are synonyms) as objects and abstractions as arrows. Therefore we can neatly compose abstractions and obtain abstractions as a result in the most natural way we can expect. This is fine for general abstractions. What about increasing or decreasing approximations? And what about representations? Do they form a subcategory of the category of abstractions? If it is the case this means we can compose either representations or approximations of the same kind in an equally natural way to get either new representations or approximations of that same kind respectively. We shall add some results about this.

Theorem 2.46

The identity abstraction i_S is a representation and both an increasing and decreasing approximation.

Proof:

Every representation is also both an increasing and decreasing approximation. Therefore it is sufficient to prove that i_S is a representation. But $i_S = \langle i', i \rangle$ where i is an identity for every semantics S. This satisfies exactly the definition of a representation. ▮

Theorem 2.47

The composition of representations is a representation.

Proof:

$$\begin{tikzcd}
P''(D'') \arrow[rrr, "I''"] & & & D'' \arrow[d, "c'"] \\
P'(D') \arrow[u, "P(a')"] \arrow[rr, "I'"] & & D' \arrow[r, dashed, "i'"] \arrow[d, "c"] & D' \arrow[d, "c"] \\
P(D) \arrow[u, "P(a)"] \arrow[r, "I"] & D \arrow[r, dashed, "i"] & D \arrow[r, dashed, "i"] & D
\end{tikzcd}$$

Consider the above diagram, we have to show that $a' \circ a$ and $c \circ c'$ satisfy clauses 2.42.i through 2.42.iii with equality in place of partial order. Assuming $a'' = a \circ a'$ we have:

i) By hypothesis we have $c(x)\ a\ x$ and $c'(y)\ a'\ y$, then we have $c(c'(y))\ a''\ y$.

ii) By hypothesis we have both $x\ a\ y \Longrightarrow x = c(y)$ and $y\ a'\ z \Longrightarrow a = c'(z)$. Then we have $x\ a''\ z \Longrightarrow x = c(c'(z))$.

iii) There is a relation r on $D \times D$ such that $x\ r\ y$ implies that $x = y$ and r is such that the following diagram commutes in the category of domains with relations. It turns out that this r is equal to i.

$$\begin{tikzcd}
P(D'') \arrow[rr, "I''"] & & D'' \arrow[d, "c' \circ c"] \\
P(D) \arrow[u, "P(a \circ a')"] \arrow[r, "I"] & D \arrow[r, dashed, "r = i"] & D
\end{tikzcd}$$

This completes the proof of the theorem.▌

This shows that representations form a proper subcategory of abstractions, sharing the same identities and closed under the same associative composition operation. The same is unfortunately not true about approximations of either kind. However if we further restrict the kind of approximations we want to consider, we can find some useful categories.

Definition 2.48 Monotonic Functions

Let A and B two domains. Let f be a relation between elements of A and B. The relation f is monotonic if and only if for every $x, y \in A$ such that $x \sqsubseteq_A y$, $f(x) \sqsubseteq_B f(y)$.

Theorem 2.49

The composition of monotonic relations is monotonic.

Proof:

Let f and g be monotonic functions. Let x, y such that $x \sqsubseteq y$. Then $f(x) \sqsubseteq f(y)$ by monotonicity of f and $g(f(x)) \sqsubseteq g(f(y))$ by monotonicity of g. Therefore $g \circ f$ is monotonic. ∎

Definition 2.50 Monotonic Abstractions

An abstraction $\langle a, c \rangle$ from S to S' is monotonic if and only if the function c is monotonic.

Theorem 2.51 The Category of Monotonic Abstractions

The class of semantics with monotonic abstractions over them form a subcategory of the category of semantics with general abstractions.

Proof:

The identity relation can easily verified to be monotonic, therefore the identity abstraction over S; $i_S = \langle i', i \rangle$ is monotonic by definition. All we have show is the composition of monotonic abstractions $\langle a, c \rangle \circ \langle b, d \rangle = \langle a \circ b, d \circ c \rangle$ is monotonic. But the composition $d \circ c$ of monotonic relations c and d is monotonic. Therefore $\langle a, c \rangle \circ \langle b, d \rangle$ is a monotonic abstraction by definition of monotonicity of abstractions. ▮

Here we have introduced a new category, namely the category of monotonic abstractions, that have apparently nothing to do with our purpose of building two categories of increasing and decreasing approximations. Please be patient, this is only an intermediate step before the results below.

Theorem 2.52 The Categories of Increasing and Decreasing Monotonic Approximations

The composition of monotonic increasing approximations is a monotonic increasing approximation. Similarly the composition of monotonic decreasing approximations is a monotonic decreasing approximation.

Proof:

$$\begin{tikzcd}
P''(D'') \arrow[rrr, "I''"] & & & D'' \arrow[d, "c'"] \\
P'(D') \arrow[u, "P(a')"] \arrow[rr, "I'"] & & D' \arrow[r, dashed, "r'"] \arrow[d, "c"] & D' \arrow[d, "c"] \\
P(D) \arrow[u, "P(a)"] \arrow[r, "I"] & D \arrow[r, dashed, "r"] & D \arrow[r, dashed, "k"] & D
\end{tikzcd}$$

Let $\langle a', c' \rangle$ and $\langle a, c \rangle$ the abstractions we want to compose as in the above diagram. The composition is $\langle a'', c'' \rangle$ defined as follows:

i) $a'' = a' \circ a$

ii) $c'' = c \circ c$

We already know that the composition of monotonic abstractions is monotonic. All we have to show is in presence of monotonicity the composition of increasing or decreasing approximations is an increasing or decreasing approximation respectively. Let's discuss the increasing case first. We want to show $\langle a'', c'' \rangle$ satisfies all clauses 2.42.i through 2.42.iii.

i) $c(x)\ a\ x$ and $c'(y)\ a'\ y$, therefore $c'(c(x))\ a'\ c(x)$ and this imply $c'(c(x))\ a''\ x$ definition of composition of relations a and a'.

ii) $x\ a\ y$ implies $x \sqsubseteq c(y)$ and $y\ a'\ z$ implies $y \sqsubseteq c'(z)$. It follows from monotonicity of c that $x\ a''\ z$ implies that $x \sqsubseteq c(c'(y'))$.

iii) Refer to the diagram above for the definitions of r, r' and k. Assume $x\ r\ y$ implies $x \sqsubseteq y$ and $u\ r'\ v$ implies $u \sqsubseteq v$. We want to show that $x(k \circ r)y$ implies that $x \sqsubseteq y$. Notice that $c(z)\ k\ c(y) \Longrightarrow z\ r'\ y$. But this implies $z \sqsubseteq y$ because of the hypothesis on r'. Monotonicity of c implies that $c(z) \sqsubseteq c(y)$. This together with the hypothesis on r implies that if $x\ y\ c(z)$ then $x \sqsubseteq c(z)$. Therefore if $x(k \circ r)c(y)$ then $x \sqsubseteq c(z)$ as expected.

The decreasing case is handled by reversing uniformly the partial order everywhere in the proof. ▮

Corrolary 2.53

All inceasing approximations $\langle a, c\rangle$ where c is an approximable function are monotonic and can be composed freely to give increasing approximations. The same is true for decreasing approximations $\langle a, c\rangle$ where c is an approximable function.

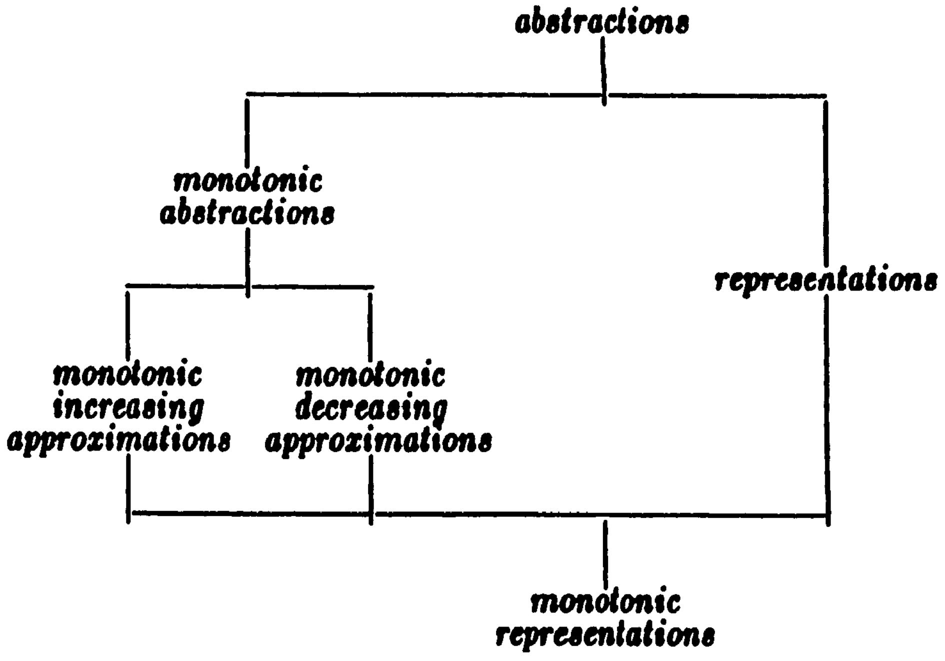
Proof:

It was shown in 2.8.iv that approximable functions are monotonic. The rest follows from 2.52. ▮

2.4.3 A Hierarchy of Categories

We now have a plethora of categories, each of them having semantics as objects and various kinds of abstractions as arrows. Each of these categories has its own use and the reader must be aware of the distinctions involved to avoid confusion. The next diagram shows them all together with the inclusion relationships that holds among them. The more inclusive categories being on top.

Figure 2.54 A Hierarchy of Categories


abstractions
monotonic abstractions
representations
monotonic increasing approximations
monotonic decreasing approximations
monotonic representations

We are now a long way from Cousot's original definitions. Before we leave this chapter let's recapitulate the essential steps we had taken. We started with the four components of a data flow analysis framework: a set of program points P, a lattice of semantic values D, a distributive function space F and an interpretation function I. We expanded the definition of each of these components to make them fit in the more formal framework of Scott's theory of domains. Category theory took a gradually increasing importance. The allowable lattices were identified with Scott's domains and were found to be the objects of several categories. One of these categories, namely the category of approximable functions was recognized as the only acceptable function space F. We found we can replace the set of program points with another categorical notion, the notion of a functor P. And the interpretation function I is required to be a P–algebra over the semantic lattice D. Throughout all this analysis, we took a careful look at the details of the lattice theoretic structure of the allowable domains, avoiding the pitfall, frequently fallen into when using category theory, of forgetting the exact nature of the objects we are working with. We concluded that data flow analysis frameworks are exactly the same thing as semantics for programming languages. We continue looking at relations that holds between such semantics and that are relevant to data flow analysis. Several other categories were found, formalizing several useful notions such as representations of a semantic domain terms of an other and increasing or decreasing approximations. All these results form a coherent and very formal theory of programming languages semantics. We will now proceed in the next chapters to show how useful this theory is, how complete the current tool box of theorems is, and, as we shall see, what is missing to be able to use the full power of the framework.

CHAPTER 3

SAMPLE DATA FLOW ANALYSIS IN LISP

This chapter is devoted to an example of a data flow problem. With the help of that problem, we will show how the formalism developed in chapter 2 can be used to perform data flow analysis.

We are interested in performing forward data flow analysis for a subset of lexically scoped LISP. Forward data flow analysis is a form of semantics where the value of an expression **Exp** is the set of all other expressions in the program whose value propagates to become the value of **Exp**. Forward data flow analysis is frequently used in compiler design for code optimization purposes (Aho and Ullman 1977). We chose a subset of LISP as the target language because it features high order functions. Lexically scoped LISP implements lambda–calculus. Since the denotational semantics of programming languages is written using lambda–calculus (Stoy 1977) it is reasonable to believe that if our approach to data flow analysis applies in this case, it is very likely to apply in most other cases as well. Data flow analysis of high order functions proved to be a very formidable task. We will show in the next chapter that a too naive implementation of the resulting semantics may lead to nontermination. Since two chapters of this thesis are devoted to the exposition of means for avoiding nontermination, a sample data flow analysis of LISP will prepare the reader for this portion of our work.

3.1 The Normal Semantics of LISP

We introduce our subset of LISP in semantics 3.1. Detailed explanations of the semantics will be given immediatelly after. We will introduce several versions of semantics for LISP in this work. Several versions of domains and functions having similar purposes but different definitions bear the same names from one semantics to another. We use the semantics number to keep track of the various versions involved. For instance, $\mathbf{Val}_{3.1}$ means the domain **Val** as defined in semantics 3.1.

Semantics 3.1 A Subset of LISP

Syntax:

Exp $\longrightarrow$ **constant**

Exp $\longrightarrow$ **var**

Exp $\longrightarrow$ **(lambda var Exp)**

Exp $\longrightarrow$ **(Exp1 Exp2)**

Exp $\longrightarrow$ **(letrec (var Exp) Exp1)**

Exp $\longrightarrow$ **(if Exp1 Exp2 Exp3)**

Domains

$\mathbf{Lisp\_constants}$ = *the domain of valid Lisp constants*

$\mathbf{Val_{3.1}} = \mathbf{Lisp\_constants} + \mathbf{Boolean} + (\mathbf{Val_{3.1}} \longrightarrow \mathbf{Val_{3.1}})$

This is the domain of valid LISP values.

$\mathbf{Ex}$ = *The domain of syntactically correct LISP expressions*

$\mathbf{Var}$ = *The domain of LISP variables. This is a subdomain of* $\mathbf{Ex}$

$\mathbf{E_{3.1}} = \mathbf{Var} \longrightarrow \mathbf{Val_{3.1}}$

This is the domain of environments. Every variable is assigned a value within an environment.

Functionality

$I_{3.1}\colon \mathbf{Ex} \longrightarrow \mathbf{E_{3.1}} \longrightarrow \mathbf{Val_{3.1}}$

$I_{3.1}$ *is the function mapping a LISP expression in* $\mathbf{Ex}$ *to its value in* $\mathbf{Val_{3.1}}$ *within the context of an environment in* $\mathbf{E_{3.1}}$.

Note:

In the following equations, injection into and selection from disjoint unions are omitted to improve readability. Context will be sufficient to make the meaning clear.

Semantics

Exp $\longrightarrow$ **constant**

$I_{3.1}[\![\texttt{constant}]\!](e) =$ *value of constant in* **Lisp_constants** *or* **Boolean**

Exp $\longrightarrow$ **var**

$I_{3.1}[\![\texttt{var}]\!](e) = e[\![\texttt{var}]\!]$

Exp $\longrightarrow$ **(lambda var Exp)**

$I_{3.1}[\![\texttt{(lambda var Exp)}]\!](e) = \lambda x.I_{3.1}[\![\texttt{Exp}]\!](e[\texttt{var} := x])$

Exp $\longrightarrow$ **(Exp1 Exp2)**

$I_{3.1}[\![\texttt{(Exp1 Exp2)}]\!](e) = I_{3.1}[\![\texttt{Exp1}]\!](e)(I_{3.1}[\![\texttt{Exp2}]\!](e))$

Exp $\longrightarrow$ **(letrec (var Exp) Exp1)**

$I_{3.1}[\![\texttt{(letrec (var Exp2) Exp1)}]\!](e) = I_{3.1}[\![\texttt{Exp1}]\!](e[\texttt{var} := v])$

Where $v = \mathbf{Y}(\lambda x.I_{3.1}[\![\texttt{Exp}]\!](e[\texttt{var} := x]))$

Exp $\longrightarrow$ **(if Exp1 Exp2 Exp3)**

$I_{3.1}[\![\texttt{(if Exp1 Exp2 Exp3)}]\!](e) = \textbf{ if } I_{3.1}[\![\texttt{Exp1}]\!](e) \textbf{ then } I_{3.1}[\![\texttt{Exp2}]\!](e)$

$\textbf{else } I_{3.1}[\![\texttt{Exp3}]\!](e)$

End of semantics 3.1 ▮

This subset contains most important LISP constructs. Their semantics is the usual lexical scoping semantics. For this reason our subset is nothing but lambda-calculus with some syntactic sugar. We have omitted from the semantics side-effects or side-effect oriented constructs such as **Prog, Setq, Catch and Throw**. The formal treatment of these constructs is fairly complex and will introduce a large number of difficulties that would obscure the illustration of how our data flow oriented categories are working. They would also obscure our planned demonstration of how nontermination occurs in a naive implementation. Therefore it is better to leave side–effects out of the example.

For the benefit of readers unfamiliar with the techniques of semantics writing, we undertake the detailed explanation of the equations of semantics 3.1. Those readers that do not need such an explanation can skip over the next few paragraphs and immediately go reading how we proceed to perform forward data flow analysis for this subset. For the convenience of readers that want to read our explanation, every equation in semantics 3.1 will be repeated before its corresponding commentary.

$\textbf{Exp} \longrightarrow \texttt{constant}$

$I_{3.1}[\![\texttt{constant}]\!](e) =$ value of constant in **Lisp_constants** or **Boolean**

The meaning of a constant expression is the corresponding constant value in the semantic domain.

$\textbf{Exp} \longrightarrow \texttt{var}$

$I_{3.1}[\![\texttt{var}]\!](e) = e(\texttt{var})$

The meaning of a variable is the value the variable takes in the environment of evaluation.

$\textbf{Exp} \longrightarrow (\texttt{lambda var Exp})$

$I_{3.1}[\![(\texttt{lambda var Exp})]\!](e) = \lambda x.I_{3.1}[\![\texttt{Exp}]\!](e[\texttt{var} := x])$

The value of a lambda–expression is a function that expects an argument parameter x. Given that parameter, we evaluate the value of the function as the value of the expression **Exp** in the context of an environment $e[\mathbf{var} := x]$ where the variable **var** is given the value of the function argument x.

Exp $\longrightarrow$ **(Exp1 Exp2)**

$$I_{3.1}[\![(\mathbf{Exp1\ Exp2})]\!](e) = I_{3.1}[\![\mathbf{Exp1}]\!](e)(I_{3.1}[\![\mathbf{Exp2}]\!](e))$$

We evaluate the function part **Exp1**, this should turn out to be the value of some lambda–expression f, i.e. this should belong to the $\mathbf{Val_{3.1}} \longrightarrow \mathbf{Val_{3.1}}$ component of the disjoint union defining the domain $\mathbf{Val_{3.1}}$. Remember we had omitted the operation of selection from the disjoint union to improve readability of the equations. This value is then called with the value of the argument part **Exp2** as an actual parameter. The reader should compare this with the semantics of **lambda** to convince himself that abstraction and application are working together properly.

Exp $\longrightarrow$ **(letrec (var Exp) Exp1)**

$$I_{3.1}[\![(\mathbf{letrec\ \ (var\ Exp2)\ Exp1})]\!](e) = I_{3.1}[\![\mathbf{Exp1}]\!](e[\mathbf{var} := v])$$

$$\text{Where } v = \mathbf{Y}(\lambda x. I_{3.1}[\![\mathbf{Exp}]\!](e[\mathbf{var} := x]))$$

A letrec expression binds the variable **var** to a value v solving the equation $v = I_{3.1}[\![\mathbf{Exp}]\!](e[\![\mathbf{var}]\!](v))$. The solution of this equation is computed with the help of Curry's fixed point combinator **Y**. Once v is computed, we evaluate the body of the **letrec** expression **Exp1** in the context of an environment where **var** is bound to its value v.

Exp $\longrightarrow$ **(if Exp1 Exp2 Exp3)**

$$I_{3.1}[\![(\textbf{if Exp1 Exp2 Exp3})]\!](e) = \textbf{ if } I_{3.1}[\![\textbf{Exp1}]\!](e) \textbf{ then } I_{3.1}[\![\textbf{Exp2}]\!](e)$$
$$\textbf{else } I_{3.1}[\![\textbf{Exp3}]\!](e)$$

We assume the expression Exp1 returns a value in the **Boolean** component of the domain $\mathbf{Val_{3.1}}$. The **Boolean** domains is a flat domain of two proper elements called true and false. There is a combinator called **if** (see [Stoy 1977] for example) in lambda-calculus such that (**if** a b c) will evaluate to b when a is true, it will evaluate to c when a is false and it will evaluate to $\perp_{\mathbf{Val_{3.1}}}$ when a is the bottom element of the **Boolean** domain. We are using this **if** combinator here.

This completes the explanation of the semantics.

3.2 The Domain of Sets of Expressions

Forward data flow analysis amounts to finding for an expression Exp the set of all other expressions in the program whose nonabstracted value propagates to become the value of Exp during the execution of the program. We are not at all interested in the usual values contained in the domain $\mathbf{Val_{3.1}}$. The very first step in building the new semantics is the definition of the proper semantic domain.

We are interested in sets of expressions. This corresponds to the domain **Ex** $\longrightarrow$ **BIT** where **Ex** is the domain of LISP expressions as defined in semantics 3.1 and **BIT** is the two-element domain $|\mathcal{BIT}|$ as defined in definition 2.20. If f belongs to **Ex** $\longrightarrow$ **BIT**, then we say that x is an element of f if and only if

$f(x) = 1$. This kind of convention apparently effectively emulates the behavior of sets. Unfortunately there is a problem. Not all sets of elements of **Ex** have a representation in **Ex** $\longrightarrow$ **BIT** under this convention. For example proposition 2.8.iv states that if $x \sqsubseteq_{\mathbf{Ex}} y$ then $f(x) \sqsubseteq_{\mathbf{BIT}} f(y)$ for every $f\colon \mathbf{Ex} \longrightarrow \mathbf{BIT}$. As a result any sets containing x but not y do not have any valid representations in **Ex** $\longrightarrow$ **BIT**. This is not a problem for our purposes. The next series of definitions and propositions shows that all sets that are interesting for data flow analysis purposes are indeed representable with **Ex** $\longrightarrow$ **BIT**; therefore the domain **Ex** $\longrightarrow$ **BIT** is indeed a correct semantic domain for forward data flow analysis of Lisp.

Definition 3.2 Proper Elements of a Domain

Let **D** *be a domain. The class of the proper elements of* **D** *is the class* $\{x \in \mathbf{D} \mid x \neq \bot_{\mathbf{D}} \wedge x \neq \top_{\mathbf{D}}\}$.

Definition 3.3 Very Proper Elements of a Set of Domains

Let $\mathbf{D}_1 \ldots \mathbf{D}_n$ *be some domains such that for some* i, $1 \leq i \leq n$, *every* $\mathbf{D}_j$ *in* $\mathbf{D}_i \ldots \mathbf{D}_n$ *are recursively defined in terms of* $\mathbf{D}_1 \ldots \mathbf{D}_n$ *with the help of cartesian products and disjoint unions. The classes* $VP(\mathbf{D}_1) \ldots VP(\mathbf{D}_n)$ *of very proper elements of* $\mathbf{D}_1 \ldots \mathbf{D}_n$ *are the smallest classes of elements of* $\mathbf{D}_1 \ldots \mathbf{D}_n$ *such that*

i) For every k, $1 \leq k \leq n$, $VP(\mathbf{D}_k)$ *is the class of proper elements of* $\mathbf{D}_k$.

ii) $\langle x, y\rangle \in VP(\mathbf{D}_u \times \mathbf{D}_v) \iff x \in VP(\mathbf{D}_u) \wedge y \in VP(\mathbf{D}_v)$.

iii) $x \in VP(\mathbf{D}_u + \mathbf{D}_v) \iff x \in VP(\mathbf{D}_u) \vee x \in VP(\mathbf{D}_v)$.

In this definition we assume the recursive definitions involve a single cartesian product or disjoint union instead of complex expressions composing these functors. This involves no loss of generality because if a cartesian product or disjoint union applies to a more complex expression, the subexpression can be eliminated as follows:

1— We introduce a new definition binding a name $\mathbf{D}_{n+1}$ to the subexpression.

2— Then we can substitute that name for the subexpression in the original definition of $\mathbf{D}_1 \ldots \mathbf{D}_n$.

3— We add $\mathbf{D}_{n+1}$ to our set of domains to get a new set of domains $\mathbf{D}_1 \ldots \mathbf{D}_{n+1}$.

Definition 3.4 Flats and Very Properly Flat Domains

A domain $\mathbf{D}$ *is a flat domain if and only if its proper elements cannot be pair wise compared using* $\mathbf{D}$*'s lattice theoretic partial order. The domain* $\mathbf{D}$ *is very properly flat iff its very proper elements cannot be pair wise compared in this fashion.*

Proposition 3.5

Let $\mathbf{D}_1 \ldots \mathbf{D}_n$ *be domains, let* $\mathbf{D}_t \ldots \mathbf{D}_n$ *be the domains that are recursively defined in terms of the others using a single cartesian product or a single disjoint* *union. Complex expressions using these operators are not allowed. If* $\mathbf{D}_1 \ldots \mathbf{D}_{i-1}$ *are flat, then* $\mathbf{D}_1 \ldots \mathbf{D}_n$ *are very properly flat.*

Proof:

We use an induction on the structure of the classes $VP(\mathbf{D}_j)$.

Basis:

Every flat domain is very properly flat. Domains $\mathbf{D}_1$ to $\mathbf{D}_{i-1}$ are very properly flat by definition 3.3.i

Induction:

Let $\langle u, v\rangle$ and $\langle x, y\rangle$ be very proper elements of $\mathbf{D}_p \times \mathbf{D}_q$. By induction hypothesis u and x do not compare in $\mathbf{D}_p$ and v and y do not compare in $\mathbf{D}_q$. Therefore by proposition 2.28.v $\langle u, v\rangle$ do not compare with $\langle x, y\rangle$ in $\mathbf{D}_p \times \mathbf{D}_q$.

Similarly, assume $inl(u)$, $inl(x)$, $inr(v)$ and $inr(y)$ are very proper elements in $\mathbf{D}_p + \mathbf{D}_q$. By induction hypothesis u do not compare with x in $\mathbf{D}_p$ and v do not compare with y in $\mathbf{D}_q$. By proposition 2.31.v neither $u \sqsubseteq_{\mathbf{D}_p+\mathbf{D}_q} x$ nor $v \sqsubseteq_{\mathbf{D}_p+\mathbf{D}_q} y$.

We can invoke 3.3.ii and 3.3.iii to complete the induction. Therefore no two elements of the very proper classes $VP(D_1) \ldots VP(D_n)$ compare. ▮

Let's give some more intuitive sense to these definitions. The domains $\mathbf{D}_1 \ldots \mathbf{D}_n$ correspond to the syntax of a programming language. The domains $\mathbf{D}_i \ldots \mathbf{D}_n$ are the nonterminals of the grammar and the domains $\mathbf{D}_1 \ldots \mathbf{D}_{i-1}$ are the terminals. The method used to establish a correspondence between grammars and domain equations was discussed in chapter 2. The domains of terminals are flat domains. Two distinct lexical elements of a program cannot be compared in terms of informational contents. They have different meanings, that's all. Improper elements of the domains of terminals such as top and bottom don't occur in programs. Programmers use only well defined lexical elements, that is proper elements of the domains. Therefore every sentential form in the program turns out to be very properly defined, they corresponds to very proper elements of the syntactic domains. Proposition 3.5 stipulates that no two distinct subexpressions in the program can be compared in terms of informational content. They are distinct, that's all. In simpler terms, this means the domain **Ex** of LISP expressions doesn't have any partial order imposed on it provided we consider only those elements of **Ex** that can actually occur in a program. Therefore we can expect the domain **Ex** $\longrightarrow$ **BIT** to accurately represents sets of expressions since the constraints imposed by the partial ordering of **Ex** on **Ex** $\longrightarrow$ **BIT** do not affect its very proper elements. This formalized below.

Definition 3.6 Set Theoretical Notation for **Ex** $\longrightarrow$ **BIT**

i) Let $x \in \mathbf{Ex}$, *the singleton* $\{'x\}$ *is defined as follow:* $\{'x\}(u) = 1$ *if* $x \sqsubseteq u$, $\{'x\}(u) = 0$ *otherwise.*

ii) $\{'x_1 \ldots x_n\} = \{'x_1\} \sqcup \{'x_2 \ldots x_n\}$

We use the typographical convention that set notation for elements of **Ex** $\longrightarrow$ **BIT** *are written with a prime symbol after the left brace.*

Proposition 3.7 Set Theoretical Properties of **Ex** $\longrightarrow$ **BIT**

Let $f\colon \mathbf{Ex} \longrightarrow \mathbf{BIT}$, *let* $vpe(f) = \{x \in VP(\mathbf{Ex}) \mid f(x) = 1\}$, *that is* $vpe(f)$ *builds the set of very proper elements of* **Ex** *that are member of* f *when* f *is interpreted as a set. The following formulas are true.*

i) $\forall x_1 \ldots x_n \in VP(\mathbf{Ex}) \quad vpe(\{'x_1 \ldots x_n\}) = \{x_1 \ldots x_n\}$

ii) $vpe(f \sqcup g) = vpe(f) \cup vpe(g)$

Proof:

i) We use an induction on n.

Basis: $vpe(\{'x\}) = \{x\}$ because by 3.5 there are no very proper y in **Ex** distinct from x such that $x \sqsubseteq y$. Therefore by definition 3.6, x is the only very proper element of $\{'x\}$.

Induction: Immediate from case 3.7.ii proven below and 3.6.ii.

ii) Immediate from 2.34.iv and the definition of join over **BIT**. ▌

What 3.7 states is that we can form sets of very proper elements of **Ex** in **Ex** $\longrightarrow$ **BIT** by enumerating the elements. We can also make the union of those sets by the join over $\mathbf{Ex} \longrightarrow \mathbf{BIT}$. We will continue to use the notation $\sqcup$ to mean that "union". This is all we need to perform forward data flow analysis.

3.3 The Computation of LISP Data Flow Information

Now we return to the original problem, namely the writing of an abstract semantics carrying forward data flow analysis of Lisp.

Semantics 3.8 A Representation of the Semantics of Lexically Scoped Lisp

Domains

Lisp_constants = *the domain of valid Lisp constants*

Ex = *The domain of syntactically correct LISP expressions*

Var = *The domain of LISP variables. This is a subdomain of* **Ex**

$$\mathbf{Val_{3.8}} = BH(\mathbf{Lisp\_constant} + \mathbf{Boolean} + (\mathbf{Val_{3.8}} \longrightarrow \mathbf{Val_{3.8}})) \times (\mathbf{Ex} \longrightarrow \mathbf{BIT})$$

The new domain of values. This domain is a cartesian product whose first component is similar to domain $\mathbf{Val_{3.1}}$ *and whose second component is a set of expressions whose values have propagated to become the current Lisp value.*

$$\mathbf{E_{3.8}} = \mathbf{Var} \longrightarrow \mathbf{Val_{3.8}}$$

This is the domain of environments. Every variable is assigned a value within an environment.

Functionality

$I_{3.8}$: $\mathbf{Ex} \longrightarrow \mathbf{E}_{3.8} \longrightarrow \mathbf{Val}_{3.8}$

*$I_{3.8}$ is the function mapping a LISP expression in **Ex** to its value in **Val**$_{3.8}$ within the context of an environment in **E**$_{3.8}$.*

Convention

*Again we omit injection into and selection from a disjoint union to improve readability. We also omit the conversion functions between domains $BH(\mathbf{D})$ and domain **D**.*

Definition

$\uplus: (\mathbf{Val}_{3.8} \times \mathbf{Ex}) \longrightarrow \mathbf{Val}_{3.8}$

$x \uplus y = \langle fst(x), snd(x) \sqcup \{'y\}\rangle$

*The function $\uplus$ accepts a value $x \in \mathbf{Val}_{3.8}$ and an expression y in **Ex** and build a new value in **Val**$_{3.8}$ that includes y among the expressions that propagates to that value.*

Semantics

Exp $\longrightarrow$ **constant**

$I_{3.8}[\![\mathtt{constant}]\!](e) = \langle \textit{value of constant}, \mathtt{constant}\rangle$

$\mathbf{Exp} \longrightarrow \mathtt{var}$

$I_{3.8}[\![\mathtt{var}]\!](e) = e[\![\mathtt{var}]\!] \uplus \mathtt{var}$

$\mathbf{Exp} \longrightarrow \mathtt{(lambda\ var\ Exp)}$

$I_{3.8}[\![\mathtt{(lambda\ var\ Exp)}]\!](e) =$

$$\langle \lambda x.I_{3.8}[\![\mathbf{Exp}]\!](e[\mathtt{var} := x]), \mathtt{(lambda\ var\ Exp)}\rangle$$

$\mathbf{Exp} \longrightarrow \mathtt{(Exp1\ Exp2)}$

$I_{3.8}[\![\mathtt{(Exp1\ Exp2)}]\!](e) = v \uplus \mathtt{(Exp1\ Exp2)}$

Where $v = fst(I_{3.8}[\![\mathbf{Exp1}]\!](e))(I_{3.8}[\![\mathbf{Exp2}]\!](e))$

$\mathbf{Exp} \longrightarrow \mathtt{(letrec\ (var\ Exp)\ Exp1)}$

$I_{3.8}[\![\mathtt{(letrec\ (var\ Exp2)\ Exp1)}]\!](e) = u \uplus \mathtt{(letrec\ (var\ Exp2)\ Exp1)}$

Where $v = \mathbf{Y}(\lambda x.I_{3.8}[\![\mathbf{Exp}]\!](e[\mathtt{var} := x \uplus \mathbf{Exp}]))$

and $u = I_{3.8}[\![\mathbf{Exp1}]\!](e[\mathtt{var} := v \uplus \mathbf{Exp}])$

$\mathbf{Exp} \longrightarrow \mathtt{(if\ Exp1\ Exp2\ Exp3)}$

$I_{3.8}[\![\mathtt{(if\ Exp1\ Exp2\ Exp3)}]\!](e) = v \uplus \mathtt{(if\ Exp1\ Exp2\ Exp3)}$

Where $v =$ **if** $fst(I_{3.8}[\![\mathbf{Exp1}]\!](e)$ **then** $I_{3.8}[\![\mathbf{Exp2}]\!](e)$

else $I_{3.8}[\![\mathbf{Exp3}]\!](e)$

End of semantics 3.8 ▮

This semantics requires some explanation. The domain equation is:

$$\mathbf{Val}_{3.8} = BH(\mathbf{Lisp\_constant} + \mathbf{Boolean} + (\mathbf{Val}_{3.8} \longrightarrow \mathbf{Val}_{3.8})) \times (\mathbf{Ex} \longrightarrow \mathbf{BIT})$$

Let's compare this with the domain equation in semantics 3.1

$$\mathbf{Val}_{3.1} = \mathbf{Lisp\_constant} + \mathbf{Boolean} + (\mathbf{Val}_{3.1} \longrightarrow \mathbf{Val}_{3.1})$$

These equations are almost identical in structure. In fact the domain $\mathbf{Val}_{3.3}$ should be interpreted as a representation of $\mathbf{Val}_{3.1}$ where every value is labeled with the set of all expressions it has propagated from. This is expressed by the fact $\mathbf{Val}_{3.3}$ is a cartesian product whose first component is similar in structure to the definition of $\mathbf{Val}_{3.1}$ (except for the BH functor that adds a top to it) and whose second component is $\mathbf{Ex} \longrightarrow \mathbf{BIT}$. As a result, a value in $\mathbf{Val}_{3.3}$ is the conjunction of all information in the corresponding value in $\mathbf{Val}_{3.1}$ with the history of intermediate expressions involved in its computation. This semantics is intended to relate data flow information with the normal semantics. This is done with the help of the function $\uplus$.

$\uplus : (\mathbf{Val}_{3.3} \times \mathbf{Ex}) \longrightarrow \mathbf{Val}_{3.3}$

$x \uplus y = \langle fst(x), snd(x) \sqcup \{'y\} \rangle$

We see that $\uplus$ accepts a value $x \in \mathbf{Val}_{3.3}$ and an expression y and adds y to the data flow information already included in x without affecting its non data flow oriented informational content. Given this background, the semantics can be understood by relating each clause in the definition of $I_{3.3}$ with the corresponding clause in the definition of $I_{3.1}$.

Exp $\longrightarrow$ **constant**

$I_{3.8}[\![\texttt{constant}]\!](e) = \langle\text{value of constant}, \texttt{constant}\rangle$

$I_{3.1}[\![\texttt{constant}]\!](e) =$ value of constant in **Lisp_constants** or **Boolean**

Here we simply initialize the data flow information with the expression generating the constant value.

Exp $\longrightarrow$ **var**

$I_{3.8}[\![\texttt{var}]\!](e) = e[\![\texttt{var}]\!] \uplus \texttt{var}$

$I_{3.1}[\![\texttt{var}]\!](e) = e[\![\texttt{var}]\!]$

We fetch the value from the current environment but the variable expression must be added to the data flow information.

Exp $\longrightarrow$ **(lambda var Exp)**

$I_{3.8}[\![\texttt{(lambda var Exp)}]\!](e) =$

$$\langle \lambda x. I_{3.8}[\![\texttt{Exp}]\!](e[\texttt{var} := x]), \texttt{(lambda var Exp)}\rangle$$

$I_{3.1}[\![\texttt{(lambda var Exp)}]\!](e) = \lambda x. I_{3.1}[\![\texttt{Exp}]\!](e[\texttt{var} := x])$

We have to initialize the data flow information with the abstraction expression.

Exp $\longrightarrow$ **(Exp1 Exp2)**

$I_{3.8}[\![\texttt{(Exp1 Exp2)}]\!](e) = v \uplus \texttt{(Exp1 Exp2)}$

Where $v = fst(I_{3.8}[\![\texttt{Exp1}]\!](e))(I_{3.8}[\![\texttt{Exp2}]\!](e))$

$I_{3.1}[\![$**(Exp1 Exp2)**$]\!](e) = I_{3.1}[\![$**Exp1**$]\!](e)(I_{3.1}[\![$**Exp2**$]\!](e))$

Here there are two difficulties to watch. First the data flow information must be discarded prior to performing the application in order to get the actual function value. Second the application expression must be added to the data flow information in the result.

Exp $\longrightarrow$ **(letrec (var Exp) Exp1)**

$I_{3.8}[\![$**(letrec (var Exp2) Exp1)**$]\!](e) = u \uplus$ **(letrec (var Exp2) Exp1)**

Where $v = \mathbf{Y}(\lambda x. I_{3.8}[\![\mathbf{Exp}]\!](e[\mathbf{var} := x \uplus \mathbf{Exp}]))$

and $u = I_{3.8}[\![\mathbf{Exp1}]\!](e[\mathbf{var} := v \uplus \mathbf{Exp1}])$

$I_{3.1}[\![$**(letrec (var Exp2) Exp1)**$]\!](e) = I_{3.1}[\![\mathbf{Exp1}]\!](e[\mathbf{var} := v])$

Where $v = \mathbf{Y}(\lambda x. I_{3.1}[\![\mathbf{Exp}]\!](e[\mathbf{var} := x]))$

Here we make sure that the expression **Exp** is recorded in the data flow information for the value of **var** and the **letrec** expression is recorded in the data flow information for its value.

Exp $\longrightarrow$ **(if Exp1 Exp2 Exp3)**

$I_{3.8}[\![$**(if Exp1 Exp2 Exp3)**$]\!](e) = v \uplus$ **(if Exp1 Exp2 Exp3)**

Where $v =$ **if** $fst(I_{3.8}[\![\mathbf{Exp1}]\!](e)$ **then** $I_{3.8}[\![\mathbf{Exp2}]\!](e)$

else $I_{3.8}[\![\mathbf{Exp3}]\!](e)$

$I_{3.1}[\![$**(if Exp1 Exp2 Exp3)**$]\!](e) =$ **if** $I_{3.1}[\![\mathbf{Exp1}]\!](e)$ **then** $I_{3.1}[\![\mathbf{Exp2}]\!](e)$

else $I_{3.1}[\![\mathbf{Exp3}]\!](e)$

Here we have to drop the data flow information from the value of **Exp1** to isolate the **Boolean** component. Once this is done, the selection of the correct alternative can be done as usual and data flow information is added to the resulting value.

3.3.1 The Analysis of the Information Systems

It should seem obvious that the semantics 3.1 and 3.8 are equivalent in some sense. In fact, 3.8 must be a representation of 3.1 in the sense of definition 2.42. To establish that claim, we are required to show how these semantics do fit in a Cousot's framework in the sense of definition 2.41. This is not an easy task. We will examine the information systems underlying $\mathbf{Val}_{3.1}$ and $\mathbf{Val}_{3.8}$ in great detail, more particularly the definitions of the propositions used to define the elements of the domains. Only knowledge of the information systems gives us enough details about the intimate structure of the domains to allow us to write definition 3.9 and the following theorems that are the real meat of the construction of the representation relations.

Here we give the reader a warning. If he feels satisfied with the pair wise comparison of the semantics clauses, he should go immediately to the paragraph after the proof of 3.10. Looking over the structure of domains is tedious and boring.

We want to uses definitions 2.29 and 2.32 to work out the information systems underlying the domains involved, especially the sets of propositions D_x. The notation $D \bowtie \mathbf{D}$ means that D is the information system underlying domain $\mathbf{D}$.

The information systems underlying semantics 3.1 are:

$$\mathbf{Val}_{\mathbf{s.1}} = \mathbf{Lisp\_constant} + \mathbf{Boolean} + (\mathbf{Val}_{\mathbf{s.1}} \longrightarrow \mathbf{Val}_{\mathbf{s.1}})$$

$$\langle D_{3.1}, \Delta_{3.1}, Con_{3.1}, \vdash_{3.1} \rangle \bowtie \mathbf{Val}_{\mathbf{s.1}}$$

$$\langle D_{LB}, \Delta_{LB}, Con_{LB}, \vdash_{LB} \rangle \bowtie \mathbf{Lisp\_constant} + \mathbf{Boolean}$$

$$\langle D_{3.1\to3.1}, \Delta_{3.1\to3.1}, Con_{3.1\to3.1}, \vdash_{3.1\to3.1} \rangle \bowtie \mathbf{Val}_{\mathbf{s.1}} \longrightarrow \mathbf{Val}_{\mathbf{s.1}}$$

For some proposition $*$ not in D_{LB} nor in $D_{3.1\to3.1}$, the set $D_{3.1}$ of propositions is defined as: (using definition 2.29)

(1) $$D_{3.1} = \{\langle X, * \rangle \mid X \in D_{LB}\} \ \cup \ \{\langle *, | \rangle X \in D_{3.1\to3.1}\} \ \cup \ \{\langle *, * \rangle\}$$

Simultaneously, $D_{3.1\to3.1}$ is defined as: (using 2.32)

(2) $$D_{3.1\to3.1} = \{\langle u, v \rangle \mid u \in Con_{3.1} \ \wedge \ v \in Con_{3.1}\}$$

We readily see that the information systems $\langle D_{3.1}, \Delta_{3.1}, Con_{3.1}, \vdash_{3.1} \rangle$ and $\langle D_{3.1\to3.1}, \Delta_{3.1\to3.1}, Con_{3.1\to3.1}, \vdash_{3.1\to3.1} \rangle$ are recursively defined in terms of each other as $\mathbf{Val}_{\mathbf{s.1}}$ and $\mathbf{Val}_{\mathbf{s.1}} \longrightarrow \mathbf{Val}_{\mathbf{s.1}}$ are.

Lets compare this with the information system for domain $\mathbf{Val}_{\mathbf{s.s}}$.

$$\mathbf{Val}_{\mathbf{s.s}} = BH(\mathbf{Lisp\_constant} + \mathbf{Boolean} + (\mathbf{Val}_{\mathbf{s.s}} \longrightarrow \mathbf{Val}_{\mathbf{s.s}})) \times (\mathbf{Ex} \longrightarrow \mathbf{BIT})$$

$$\langle D_{LB}, \Delta_{LB}, Con_{LB}, \vdash_{LB}\rangle \bowtie \textbf{Lisp\_constant} + \textbf{Boolean}$$

$$\langle D_{3.8}, \Delta_{3.8}, Con_{3.8}, \vdash_{3.8}\rangle \bowtie \textbf{Vals}_{3.8}$$

$$\langle D_{E\rightarrow B}, \Delta_{E\rightarrow B}, Con_{E\rightarrow B}, \vdash_{E\rightarrow B}\rangle \bowtie \textbf{Ex} \longrightarrow \textbf{BIT}$$

$$\langle D_{3.8\rightarrow 3.8}, \Delta_{3.8\rightarrow 3.8}, Con_{3.8\rightarrow 3.8}, \vdash_{3.8\rightarrow 3.8}\rangle \bowtie \textbf{Vals}_{3.8} \longrightarrow \textbf{Vals}_{3.8}$$

$$\langle D_{BH}, \Delta_{BH}, Con_{BH}, \vdash_{BH}\rangle \bowtie BH(\textbf{Lisp\_constant} + \textbf{Boolean} + (\textbf{Vals}_{3.8} \longrightarrow \textbf{Vals}_{3.8}))$$

For some proposition $*'$ not in D_{LB} nor in $D_{3.8} \rightarrow D_{3.8}$, we can combine definitions 2.25 and 2.29 to obtain:

(3) $$D_{BH} = \{\langle X, *'\rangle \mid X \in D_{LB}\} \cup \{\langle *', X\rangle \mid X \in D_{3.8\rightarrow 3.8}\} \cup \{\langle *', *'\rangle\}$$

Here we use 2.13 to obtain the definition of $D_{3.8}$:

(4) $$D_{3.8} = \{\langle X, \Delta_{E\rightarrow B}\rangle \mid X \in D_{BH}\} \cup \{pair \Delta_{BH} X \mid X \in D_{E\rightarrow B}\}$$

Here we use 2.32 to obtain the definition of $D_{3.8\rightarrow 3.8}$:

(5) $$D_{3.8\rightarrow 3.8} = \{\langle u, v\rangle \mid u \in Con_{3.8} \wedge v \in Con_{3.8}\}$$

This shows how the sets of propositions from the various information systems are defined. We can work out the definitions of the various Δ, Con sets and $\vdash$ relations but we will not. We only want to define the abstraction and concretization functions between semantics 3.1 and 3.8 and it turns out that we need to know what the propositions were.

Definition 3.9

Relations $\Downarrow$ and $\downarrow$ are defined recursively in terms of each other as:

Relation $\downarrow$ is a relation from $D_{3.8}$ to $D_{3.1}$ defined as follow:

i) $\langle\langle *', *'\rangle, s\rangle \downarrow \langle *, *\rangle \quad \forall s \in D_{E \to B}$

ii) $\langle\langle a, *'\rangle, s\rangle \downarrow \langle a, *\rangle \quad \forall s \in D_{E \to B}\, \forall a \in D_{LB}$

iii) $\langle\langle *', x\rangle, s\rangle \downarrow \langle *, y\rangle \quad \forall s \in D_{E \to B}\, \forall x \in D_{3.8 \to 3.8}$

where y is defined as:

$y = \{\langle a, b\rangle \mid \exists\langle u, v\rangle \in x, \quad u \Downarrow a \wedge v \Downarrow b\}$

iv) *$X \downarrow Y$ only as implied by i) through iii) above.*

Relation $\Downarrow$ is defined over sets of propositions in $D_{3.8}$ and $D_{3.1}$ respectively as follow:

let $S \subseteq D_{3.8}$ and $T = Cl(\{x \in D_{3.1} \mid y \in S \wedge y \downarrow x\})$

$S \Downarrow T \iff \forall y \in S\, \exists x \in D_{3.1} (y \downarrow x)$

and all finite subset of T are in $Con_{3.1}$.

Relation $\Uparrow$ is defined between sets of propositions in $D_{3.8}$ and $D_{3.1}$ respectively as the inverse of $\Downarrow$.

$$X \Uparrow Y \iff Y \Downarrow X$$

Let's discuss the meaning of relations $\downarrow$, $\Downarrow$ and $\Uparrow$. We have said that the domain **Val$_{3.8}$** is the conjunction of the information contained in the domain **Val$_{3.1}$**

with the information in $\mathbf{Ex} \longrightarrow \mathbf{BIT}$. This is only an intuitively appealing approximation of the reality. Let's look back again at the definitions:

$$\mathbf{Val}_{\mathbf{s.s}} = BH(\mathbf{Lisp\_constant} + \mathbf{Boolean} + (\mathbf{Val}_{\mathbf{s.s}} \longrightarrow \mathbf{Val}_{\mathbf{s.s}})) \times (\mathbf{Ex} \longrightarrow \mathbf{BIT})$$

$$\mathbf{Val}_{\mathbf{s.1}} = \mathbf{Lisp\_constant} + \mathbf{Boolean} + (\mathbf{Val}_{\mathbf{s.1}} \longrightarrow \mathbf{Val}_{\mathbf{s.1}})$$

Let's ignore for a moment the complications introduced by the presence of the functor BH in the definition of $\mathbf{Val}_{\mathbf{s.s}}$. If we compute the projection $fst(x)$ do we get an element of $\mathbf{Val}_{\mathbf{s.1}} \longrightarrow \mathbf{Val}_{\mathbf{s.1}}$? Not quite. If $fst(x)$ happens to be in the **Lisp_constant** or in the **Boolean** component of the disjoint union, it's not bad. But if $fst(x)$ happens to be in $\mathbf{Val}_{\mathbf{s.s}} \longrightarrow \mathbf{Val}_{\mathbf{s.s}}$, this is not at all the same as having an element of $\mathbf{Val}_{\mathbf{s.1}} \longrightarrow \mathbf{Val}_{\mathbf{s.1}}$. Data flow information is present in $fst(x)(y)$ as well as in x and in y. The domain $\mathbf{Val}_{\mathbf{s.s}}$ is more than the conjunction of $\mathbf{Val}_{\mathbf{s.1}}$ with $\mathbf{Ex} \longrightarrow \mathbf{BIT}$. It also recursively preserves the conjunction of informational content across functional abstractions and applications. This implies $\mathbf{Val}_{\mathbf{s.1}}$ and the first component of $\mathbf{Val}_{\mathbf{s.s}}$ don't have the same type. Our problem is to show that the first component of $\mathbf{Val}_{\mathbf{s.s}}$ includes the informational content of Val despite this change in type.

Relation $\downarrow$ deals with propositions in the information system. The statement $X \downarrow Y$ means that if we forget data flow propagation information enclosed in X, then X has the same meaning as Y. Refer to the relevant definitions of chapter 2 when reading definition 3.9. Clause 3.9.i forgets data flow information when we talk about the least informative proposition of the disjoint union. Clause 3.9.ii forgets this information when we talk about propositions that describes elements of **Lisp_constant** + **Boolean**. Clause 3.9.iii forgets data flow information when we discuss the elements of $\mathbf{Val}_{\mathbf{3.8}} \longrightarrow \mathbf{Val}_{\mathbf{3.8}}$ and $\mathbf{Val}_{\mathbf{3.1}} \longrightarrow \mathbf{Val}_{\mathbf{3.1}}$ respectively. If the reader feels he do not understand 3.9.iii right now, he should read the next two paragraphs and then go back to that tricky clause.

Relation $\Downarrow$ generalizes $\downarrow$ over sets of propositions. By extension, $\Downarrow$ is a relation defined over elements of the domains $\mathbf{Val}_{\mathbf{3.8}}$ and $\mathbf{Val}_{\mathbf{3.1}}$. It states, like $\downarrow$, that $v \Downarrow u$ if and only if v has the same meaning as u provided we forget about data flow analysis information enclosed in v. Relation $\Uparrow$ is a notational convenience for the inverse of $\Downarrow$. We elaborate on the meaning of $\Downarrow$ and $\Uparrow$ when applied to elements in theorem 3.10

Let's give more explanations on clause 3.9.iii. Its intention is to state that a function g: $\mathbf{Val}_{\mathbf{3.8}} \longrightarrow \mathbf{Val}_{\mathbf{3.8}}$ will relate to a function f: $\mathbf{Val}_{\mathbf{3.1}} \longrightarrow \mathbf{Val}_{\mathbf{3.1}}$ if and only if $g(v) \Downarrow f(u)$ for all properly typed u and v such that $v \Downarrow u$. Why is it so? Our problem is we cannot have equality to hold between elements of different domains. This is not meaningful. We have to be happy with a relation that mimics the extensionality law. The principle of extensionality states that $g = f \iff \forall u,\ g(u) = f(u)$. We can't have such a law to hold when g and f have different types. We write instead $\langle g, s\rangle \Downarrow f \iff g(v) \Downarrow f(u)$ for all v and u such that $v \Downarrow u$. Relation $\Downarrow$ becomes a sort of "equality across types". We relate propositions in $D_{3.8}$ and $D_{3.1}$ with $\downarrow$ in such a way this pseudo extensionality will be true. Beware, not every proposition in $D_{3.8}$ has a corresponding proposition in $D_{3.1}$ according to this definition. The BH functor introduces a top element in the disjoint union. This allows some functions to exists in $\mathbf{Val_{3.8}} \longrightarrow \mathbf{Val_{3.8}}$ that have no equivalent in $\mathbf{Val_{3.1}} \longrightarrow \mathbf{Val_{3.1}}$.

Theorem 3.10

Let $\mathbf{Z} = BH(\mathbf{Lisp\_constant} + \mathbf{Boolean} + (\mathbf{Val_{3.8}} \rightarrow \mathbf{Val_{3.8}}))$

All of the following are true

i) For any x, if there is a $y \in \mathbf{Val_{3.1}}$ such that $x \Downarrow y$, then y is unique.

ii) $\langle \bot_{\mathbf{Z}}, s\rangle \Downarrow \bot_{\mathbf{Val_{3.1}}}$ *for every* $s \in \mathbf{Ex} \longrightarrow \mathbf{BIT}$.

iii) $\langle x, s\rangle \Downarrow x$ *for every* $x \in \mathbf{Lisp\_constant} + \mathbf{Boolean}$ *and every* $s \in \mathbf{Ex} \longrightarrow \mathbf{BIT}$

iv) Assume g, s and f are elements of $\mathbf{Val_{3.8}} \longrightarrow \mathbf{Val_{3.8}}$, $\mathbf{Ex} \longrightarrow \mathbf{BIT}$ *and* $\mathbf{Val_{3.1}}$ *respectively. Then* $\langle g, s\rangle \Downarrow f$ *if and only if* $g(v) \Downarrow f(u)$ *for every u and v such that $v \Downarrow u$. Here u and v can be any sets and application is taken in the sense of definition 2.7.*

Proof:

i) Uniqueness of y immediately follows from the definition of $\Downarrow$ because if $x \Downarrow y$ then $y = Cl(s)$ for exactly one s.

ii) $\langle \bot_{\mathbf{Z}}, s\rangle = \{\langle\langle *', *'\rangle, a\rangle \mid s \vdash_{E \to B} a\}$ by definitions of $\mathbf{D_{3.8}}$ and $\bot_{\mathbf{Z}}$

$\langle \bot_{\mathbf{Z}}, s\rangle \Downarrow \{\langle *, *\rangle\}$ By definition of $\Downarrow$

$\langle \bot_{\mathbf{Z}}, s\rangle \Downarrow \bot_{\mathbf{Vals.1}}$ By definition of $\bot_{Vals.1}$.

iii) $\langle x, s\rangle = \{\langle\langle a, *'\rangle, \Delta_{E\to B}\rangle \mid x \vdash_{LB} a\} \cup \{\langle\langle *', *'\rangle, b\rangle \mid s \vdash_{E\to B} b\}$ by definition 5.i

$\langle x, s\rangle \Downarrow \{a \mid x \vdash_{LB} a\} \cup \{\langle *', *'\rangle\}$ by definition of $\Downarrow$

$\langle x, s\rangle \Downarrow x$ by definition of element x in $\mathbf{Vals.1}$

iv) Assume that $\langle g, s\rangle \Downarrow f$. Then by clause iii) of 3.9 we have

$$g = \{\ldots\langle x_i, y_i\rangle\ldots\} \quad f = Cl(\{\ldots\langle X_i, Y_i\rangle\ldots\})$$

where $x_i \Downarrow X_i$ and $y_i \Downarrow Y_i$ for all i

Definition 2.7 implies that

$$g(v) = \bigcup\{y_i \mid \exists x \subseteq v, \quad \langle x, y_i\rangle \in g\}$$

Definition 3.9 together with the identity $Cl(\bigcup\{Cl(x) \mid x \in S\} = Cl(\bigcup\{x \mid x \in S\})$ implies that if Y is such that $g(v) \Downarrow Y$, we have the identities:

$$Y = Cl(\bigcup\{Y_i \mid \langle x, y_i\rangle \in g \text{ for some } x \subseteq v\})$$

$$Y = Cl(\bigcup\{Y_i \mid \langle X_i, Y_i\rangle \in f, \quad x_i \Downarrow X_i \text{ for some } x_i \subseteq v\})$$

Definition 2.32.iv implies that if $X \vdash_{3.1} Z$ and $\langle Z, Z'\rangle \in Y$, and $Z' \vdash_{3.1} Y'$, then $\langle X, Y'\rangle$ will also be in Y. The set Y is also the least set satisfying this condition and including the set $\{Y_i \mid \langle X_i, Y_i\rangle \in f,\ x_i \Downarrow X_i, \text{ for some } x_i \subseteq v\}$.

It also implies that $Y \vdash_{3.1\to3.1} \langle X, Y'\rangle$ in no other cases. The same definition 2.32 iv) implies likewise that if $X \vdash_{3.1} Z$ and $\langle Z, Z'\rangle \in f$ and $Z' \vdash_{3.1} Y'$ then $\langle X, Y'\rangle$ is also in f. Therefore Y is included in f. On the other hand assume that $x_i \subseteq v$, $X \subseteq X_i$, $x_i \Downarrow X_i$ and $\langle X, Y'\rangle \in f$. The fact that $g \Downarrow f$ implies that $\langle X, Y'\rangle$ must be in Y because f must be the least set satisfying the same closure property from the same initial data. This allows us to rewrite the last formula as:

$$Y = Cl(\bigcup\{Y' \mid \langle Z, Y'\rangle \in f, \quad Z \subseteq Cl(\bigcup\{X_i \mid x_i \Downarrow X_i, \quad x_i \subseteq v\})\})$$

Since the hypothesis $v \Downarrow u$ and definition 3.9 implies that $u = Cl(\bigcup\{Z \mid x \Downarrow Z \wedge x \subseteq v\})$, we can further rewrite the formula as:

$$Y = Cl(\bigcup\{Y' \mid \langle Z, Y'\rangle \in f \wedge Z \subseteq u\})$$

Definition 2.7 allow us to write

$$Y = Cl(f(u))$$

and the definition of element implies that $f(u)$ is equal to its closure. Therefore, $Y = f(u)$ as required.

End of proof of 3.10.▮

We want to stress the similarity between the law of extensionality and 3.10.iv. Even readers that had skipped over theorem 3.10 and its preliminaries should look at this.

Law of Extensionality:

$$\forall u, v,\ (u = v) \Rightarrow (g(u) = f(u)) \quad \Longleftrightarrow \quad (g = f)$$

Clause 3.10.iv:

$$\forall u, v,\ (v \Downarrow u)) \Rightarrow (g(v) \Downarrow f(u)) \quad \Longleftrightarrow \quad (\langle g, s\rangle \Downarrow) f$$

We introduce a relation $v \Downarrow u$ whose meaning is "v has the same informational content as u once data flow information is ignored". We want to state that elements of $\mathbf{Val}_{\mathbf{3.8}}$ have the same informational content as elements of $\mathbf{Val}_{\mathbf{3.1}}$ once data flow information has been stripped out. We have to overcome difference of types. For instance, let $\langle g, s\rangle \in \mathbf{Val}_{\mathbf{3.8}}$ and $f \in \mathbf{Val}_{\mathbf{3.1}}$. If g happens to be in the **Lisp_constant** or **Boolean** components, we can just discard the data flow information and state that $\langle g, s\rangle \Downarrow f \iff g = f$. However when g happens to be in $\mathbf{Val}_{\mathbf{3.8}} \rightarrow \mathbf{Val}_{\mathbf{3.8}}$ and f is in $\mathbf{Val}_{\mathbf{3.1}} \rightarrow \mathbf{Val}_{\mathbf{3.1}}$, it is no longer possible to use equality of f and g for this purpose since g do not even belong to the same domain as f. We did the next best thing: we rewrote the extensionality law taking into account the change in types. The result is a kind of "equality relation" across unequal domains.

3.3.2 Equivalence of Semantics

The reader should now be aware that relation $\Downarrow$ actually strips out data flow information to retrieve the value in $\mathbf{Val}_{\mathbf{3.1}}$ from any of its representation in $\mathbf{Val}_{\mathbf{3.8}}$. Remember we have to show that semantics 3.8 is a representation of 3.1 in the sense of 2.42. Relation $\Downarrow$ will be our concretization function. The abstraction will be its reverse. Proving we have a representation means information is handled in harmony in both semantics; that is, these relations are preserved across abstractions and applications.

There is a functor $LISP$ such that, given a domain **Val**, LISP will build a domain $LISP(\mathbf{Val}) = (\mathbf{Ex} \times (\mathbf{Var} \longrightarrow \mathbf{Val}))$ where **Ex** is the domain of Lisp programs and **Var** is the domain of Lisp variables. Let $Uncurry(I)(x, y) = I(x)(y)$ be the definition of the combinator $Uncurry$. In semantics 3.1, the triple $\langle LISP, \mathbf{Val}_{3.1}, Uncurry(I_{3.1})\rangle$ is a Cousot's framework since $\mathbf{Val}_{3.1}$ together with $Uncurry(I_{3.1})$ form a $LISP$-algebra in the sense of 2.40. Similarly the triple $\langle LISP, \mathbf{Val}_{3.8}, Uncurry(I_{3.8})\rangle$ from semantics 3.8 also forms a Cousot framework. According to the definition of representations, we want to find a pair $\langle a, c\rangle$ so that the following categorical diagram commutes:

$$\begin{tikzcd}
LISP(\mathbf{Val}_{3.8}) \arrow[rr, "Uncurry(I_{3.8})"] & & \mathbf{Val}_{3.8} \arrow[d, "c"] \\
LISP(\mathbf{Val}_{3.1}) \arrow[u, "a"] \arrow[r, "Uncurry(I_{3.1})"] & \mathbf{Val}_{3.1} \arrow[r, dashed, "i"] & \mathbf{Val}_{3.1}
\end{tikzcd}$$

The relation i must be the identity over $\mathbf{Val}_{3.1}$. A way to build the required relations a and c is given by the following theorem:

Theorem 3.11

The pair $\langle \Uparrow, \Downarrow \rangle$ *is a representation from* $\langle LISP, \mathbf{Val}_{3.1}, Uncurry(I_{3.1}) \rangle$ *to* $\langle LISP, \mathbf{Val}_{3.8}, Uncurry(I_{3.8}) \rangle$.

Proof:

Looking at the definition of representations, the definition of the relation a and at the definition of the LISP functor, this theorem can be equivalently phrased as:

Assuming that $e\colon \mathbf{Var} \longrightarrow \mathbf{Val}_{3.1}$ and $e'\colon \mathbf{Var} \longrightarrow \mathbf{Val}_{3.8}$ are such that $e'[\mathbf{var}] \Downarrow e[\mathbf{var}]$ for all **var** in **Var**, then for every expression **Exp** in **Ex**, we have $I_{3.8}[\mathbf{Exp}](e') \Downarrow I[\mathbf{Exp}](e)$.

This restatement of the theorem is obtained when we phrase into English the meaning of the categorical diagram defining representations.

We can show that $I_{3.8}[\mathbf{Exp}](e') \Downarrow I[\mathbf{Exp}](e)$ by induction on the complexity of the syntax of the Lisp expression.

Exp $\longrightarrow$ **constant**

This case follows from 3.10.iii

Exp $\longrightarrow$ **var**

This case follows from the hypotheses $e'[\mathbf{var}] \Downarrow e[\mathbf{var}]$ for all **var**.

Exp $\longrightarrow$ **(lambda var Exp)**

We have the pair of equations:

$I_{3.1}[\![\texttt{(lambda var Exp)}]\!](e) = \lambda x.I_{3.1}[\![\texttt{Exp}]\!](e[\texttt{var} := x])$

$I_{3.8}[\![\texttt{(lambda var Exp)}]\!](e') =$

$$\langle \lambda x.I_{3.8}[\![\texttt{Exp}]\!](e'[\texttt{var} := x]), \texttt{(lambda var Exp)}\rangle$$

This induction hypothesis is for all e and e' such that $e'[\![\texttt{var}]\!] \Downarrow e[\![\texttt{var}]\!]$ is true for every variable var, the relation $I_{3.8}[\![\texttt{Exp}]\!](e') \Downarrow I_{3.1}[\![\texttt{Exp}]\!](e)$ is true. If we assume that $x \Downarrow y$, then the induction hypothesis implies that:

$$I_{3.8}[\![\texttt{Exp}]\!](e'[\texttt{var} := x]) \Downarrow I_{3.1}[\![\texttt{Exp}]\!](e[\texttt{var} := y])$$

Looking at the right hand side of the definition of $I_{3.1}$ and $I_{3.8}$, we can perform the reverse of beta reduction to conclude the following relation is true for all x, y such that $x \Downarrow y$.

$$I_{3.8}[\![\texttt{(lambda var Exp)}]\!](e')(x) \Downarrow I_{3.1}[\![\texttt{(lambda var Exp)}]\!](e)(y)$$

The rest follows from 3.10.iv.

Exp $\longrightarrow$ **(Exp1 Exp2)**

We have the pair of equations:

$$I_{3.1}[\![\texttt{(Exp1 Exp2)}]\!](e) = I_{3.1}[\![\texttt{Exp1}]\!](e)(I_{3.1}[\![\texttt{Exp2}]\!](e))$$

$$I_{3.8}[\![\texttt{(Exp1 Exp2)}]\!](e') = v \uplus \texttt{(Exp1 Exp2)}$$

$$\text{where} v = fst(I_{3.8}[\![\texttt{Exp1}]\!](e'))(I_{3.8}[\![\texttt{Exp2}]\!](e'))$$

By induction hypothesis we have both

$I_{3.8}[\![\mathbf{Exp1}]\!](e') \Downarrow I_{3.1}[\![\mathbf{Exp1}]\!](e)$ and

$I_{3.8}[\![\mathbf{Exp2}]\!](e') \Downarrow I_{3.1}[\![\mathbf{Exp2}]\!](e)$

This together with 3.10.iv leads directly to the desired result.

Exp $\longrightarrow$ **(letrec (var Exp) Exp1)**

This case can be ignored since this syntax can be replaced by another one not using letrec. To achieve this we use the explicit definition of the fixed point combinator.

Exp $\longrightarrow$ **(if Exp1 Exp2 Exp3)**

The induction hypothesis assumes that all of $I_{3.8}[\![\mathbf{Exp1}]\!](e') \Downarrow I_{3.1}[\![\mathbf{Exp1}]\!](e)$, $I_{3.8}[\![\mathbf{Exp2}]\!](e') \Downarrow I_{3.1}[\![\mathbf{Exp2}]\!](e)$ and $I_{3.8}[\![\mathbf{Exp3}]\!](e') \Downarrow I_{3.1}[\![\mathbf{Exp3}]\!](e)$ are true. This case follows from theorem 3.10.iii applied to the value of **Exp1** and the definition of **if** .

End of theorem 3.11. ▮

The reader will have noticed how formidable a task it is to show that one semantics is an abstraction or a representation of another according to definition 2.42. He will have also noticed how obvious the result would seem when comparing both definitions from a purely syntactical point of view. We should be able to use some theorem or notation that would make quick and easy the proofs of apparently trivial things. Unfortunately the author is not aware of the required results. Also more time would be required to investigate further such fundamental problems without going beyond the scope of the present work and putting its completion in jeopardy. From now on we will stop writing these tedious proofs and rely on the apparent syntactic similarities in the assumption that we are indeed writing abstractions or representations. This may make our work "unsafe" from a purely formal point of view and it will appear incomplete. On the other hand, we conjecture syntactical similarities as strong as those we encountered are strong indications that we are going in the right direction and our expectations about formalism can be relaxed for a while, until we develop better tools to meet an acceptable level of mathematical rigor. Very few data flow analysts take the trouble to show that their work is indeed a Cousot like approximation with the help of more proofs than such syntactic similarities. A similar attitude exists among those authors who write "continuation passing style semantic" (see [Stoy 1977] for instance). The extra continuation parameter this style of semantics adds makes the resulting function differ in type from the normal semantics. However, they are nevertheless regarded as equivalent with neither proof nor a formal statement of what such equivalence should mean. This is especially flagrant when we consider programs that compute high order functions. Programs such as factorial or append that compute nonfunction results don't cause problems since they happen to return the same value in both semantics. For these reasons, despite the fact we have pinpointed the formal difficulty, we feel justified in leaving it unsolved until future research brings an acceptable solution.

3.4 Approximate Data Flow Analysis of LISP

Although semantics 3.8 effectively computes data flow information, it is inadequate for purposes of compiler writing because it has as much informational contents as semantics 3.1. We do not want to execute the program when we compile it. It is very reasonable to assume that I/O constructs will be included in the language. This implies that some of the information required for execution is not even available at compile time. What we need is a semantics that forgets the usual meaning of Lisp programs and concentrates solely on data flow analysis.

Semantics 3.12 Data Flow Oriented Abstract Semantics of Lexically Scoped Lisp

Domains

Var = *The domain of Lisp variables*

Ex = *The domain of Lisp expressions*

$\mathbf{Val}_{3.12} = (\mathbf{Val}_{3.12} \longrightarrow \mathbf{Val}_{3.12}) \times (\mathbf{Ex} \longrightarrow \mathbf{BIT})$

The new domain of values

$\mathbf{E}_{3.12} = \mathbf{Var} \longrightarrow \mathbf{Val}_{3.12}$

the domain of environments.

Functionality

$I_{3.12}\colon \mathbf{Ex} \longrightarrow \mathbf{E}_{3.12} \longrightarrow \mathbf{Val}_{3.12}$

$I_{3.12}[\![\mathrm{Exp}]\!](e)$ will give the value of the expression Exp in the environment e.

Convention

Again we omit injection into and selection from a disjoint union to improve readability.

Definition

$\uplus : (\mathbf{Val}_{3.12} \times \mathbf{Ex}) \longrightarrow \mathbf{Val}_{3.12}$

$x \uplus y = \langle fst(x),\ snd(x) \sqcup \{\text{'y}\}\rangle$

The function $\uplus$ accepts a value $x \in \mathbf{Val}_{3.12}$ and an expression $y \in \mathbf{Ex}$ and build a new value in $\mathbf{Val}_{3.12}$ that includes y among the expressions that propagates to that value.

Equations

Exp $\longrightarrow$ `constant`

$I_{3.12}[\![\texttt{constant}]\!](e) = \langle \perp_{\mathbf{Val}_{3.12} \rightarrow \mathbf{Val}_{3.12}}, \texttt{constant}\rangle$

Exp $\longrightarrow$ `var`

$I_{3.12}[\![\texttt{var}]\!](e) = e[\![\texttt{var}]\!] \uplus \texttt{var}$

Exp $\longrightarrow$ **(lambda var Exp)**

$I_{3.12}[\![\textbf{(lambda var Exp)}]\!](e) =$

$$\langle \lambda x. I_{3.12}[\![\mathbf{Exp}]\!](e[\texttt{var} := x]), \textbf{(lambda var Exp)}\rangle$$

Exp $\longrightarrow$ **(Exp1 Exp2)**

$I_{3.12}[\![\textbf{(Exp1 Exp2)}]\!](e) = v \uplus \textbf{(Exp1 Exp2)}$

where $v = fst(I_{3.12}[\![\mathbf{Exp1}]\!](e)(I_{3.12}[\![\mathbf{Exp2}]\!](e))$

Exp $\longrightarrow$ **(letrec (var Exp) Exp1)**

$I_{3.12}[\![$**(letrec (var Exp2) Exp1)**$]\!](e) = u \uplus$ **(letrec (var Exp2) Exp1)**

where $u = I_{3.12}[\![\mathbf{Exp1}]\!](e[\mathbf{var} := v \uplus \mathbf{Exp}])$

and $v = \mathbf{Y}(\lambda x. I_{3.12}[\![\mathbf{Exp}]\!](e[\mathbf{var} := x \uplus \mathbf{Exp}]))$

Exp $\longrightarrow$ **(if Exp1 Exp2 Exp3)**

$I_{3.12}[\![$**(if Exp1 Exp2 Exp3)**$]\!](e) = v \uplus$ **(if Exp1 Exp2 Exp3)**

where $v = I_{3.12}[\![\mathbf{Exp2}]\!](e) \sqcup I_{3.12}[\![\mathbf{Exp3}]\!](e)$

End of semantics 3.12. ▌

This semantics is best understood when compared to semantics 3.8. Let's take a look at the domain equations.

$$\mathbf{Val_{3.12}} = (\mathbf{Val_{3.12}} \longrightarrow \mathbf{Val_{3.12}}) \times (\mathbf{Ex} \longrightarrow \mathbf{BIT})$$

$$\mathbf{Val_{3.8}} = BH(\mathbf{Lisp\_constant} + \mathbf{Boolean} + (\mathbf{Val_{3.8}} \longrightarrow \mathbf{Val_{3.8}})) \times (\mathbf{Ex} \longrightarrow \mathbf{BIT})$$

We immediately see that in $\mathbf{Val_{3.12}}$ we have omitted the **Lisp_constant** and the **Boolean** components from the disjoint union. This is because we discard any information relevant only to the normal semantics to keep only data flow information in $\mathbf{Ex} \longrightarrow \mathbf{BIT}$. We keep the component $\mathbf{Val_{3.12}} \longrightarrow \mathbf{Val_{3.12}}$ because we want high order data flow information to evaluate how applications and abstractions propagate data and not just the flat information in $\mathbf{Ex} \longrightarrow \mathbf{BIT}$.

3.12 $\uplus: (\mathbf{Val}_{3.12} \times \mathbf{Ex}) \longrightarrow \mathbf{Val}_{3.12}$

$x \uplus y = \langle fst(x),\ snd(x) \sqcup \{'y\} \rangle$

3.8 $\uplus: (\mathbf{Val}_{3.8} \times \mathbf{Ex}) \longrightarrow \mathbf{Val}_{3.8}$

3.8 $x \uplus y = \langle fst(x), snd(x) \sqcup \{'y\} \rangle$

The auxiliary function $\uplus$ introduced in 3.8 is replaced by its equivalent $\uplus$. The latter is identical to $\uplus$ but for its type. Both adds data flow information y to the **Ex** $\longrightarrow$ **BIT** component of x without affecting its other component.

From now on we will start to compare each clause in the definition of $I_{3.12}$ with the corresponding clause in the definition of $I_{3.8}$. We will show how the information relevant to the normal semantics is discarded without loss of data flow information.

Exp $\longrightarrow$ **constant**

$I_{3.12}[\![\mathbf{constant}]\!](e) = \langle \bot_{\mathbf{Val}_{3.12} \rightarrow \mathbf{Val}_{3.12}}, \mathbf{constant} \rangle$

$I_{3.8}[\![\mathbf{constant}]\!](e) = \langle \text{value of constant}, \mathbf{constant} \rangle$

A constant does not correspond to any information in $\mathbf{Val}_{3.12} \longrightarrow \mathbf{Val}_{3.12}$, so $I_{3.12}$ registers the least possible amount of information in that domain.

Exp $\longrightarrow$ **var**

$I_{3.12}[\![\mathbf{var}]\!](e) = e[\![\mathbf{var}]\!] \uplus \mathbf{var}$

$I_{3.8}[\![\mathbf{var}]\!](e) = e[\![\mathbf{var}]\!] \uplus \mathbf{var}$

The treatment of a variable does not depend on the semantic domain.

Exp $\longrightarrow$ **(lambda var Exp)**

$I_{3.12}[\![\textbf{(lambda var Exp)}]\!](e) =$

$$\langle \lambda x. I_{3.12}[\![\textbf{Exp}]\!](e[\textbf{var} := x]), \textbf{(lambda var Exp)} \rangle$$

$I_{3.8}[\![\textbf{(lambda var Exp)}]\!](e) =$

$$\langle \lambda x. I_{3.8}[\![\textbf{Exp}]\!](e[\textbf{var} := x]), \textbf{(lambda var Exp)} \rangle$$

Abstractions are handled the same way in both domains.

Exp $\longrightarrow$ **(Exp1 Exp2)**

$I_{3.12}[\![\textbf{(Exp1 Exp2)}]\!](e) = v \uplus \textbf{(Exp1 Exp2)}$

wherev $= fst(I_{3.12}[\![\textbf{Exp1}]\!](e)\big(I_{3.12}[\![\textbf{Exp2}]\!](e)\big)$

$I_{3.8}[\![\textbf{(Exp1 Exp2)}]\!](e) = v \uplus \textbf{(Exp1 Exp2)}$

Where $v = fst(I_{3.8}[\![\textbf{Exp1}]\!](e))(I_{3.8}[\![\textbf{Exp2}]\!](e))$

Applications are handled the same way in both domains.

Exp $\longrightarrow$ **(letrec (var Exp) Exp1)**

$I_{3.12}[\![\textbf{(letrec (var Exp2) Exp1)}]\!](e) = u \uplus \textbf{(letrec (var Exp2) Exp1)}$

where $u = I_{3.12}[\![\textbf{Exp1}]\!](e[\textbf{var} := v \uplus \textbf{Exp}])$

and $v = \mathbf{Y}(\lambda x. I_{3.12}[\![\textbf{Exp}]\!](e[\textbf{var} := x \uplus \textbf{Exp}]))$

$I_{3.8}[\![\textbf{(letrec (var Exp2) Exp1)}]\!](e) = u \uplus \textbf{(letrec (var Exp2) Exp1)}$

Where $v = \mathbf{Y}(\lambda x. I_{3.8}[\![\textbf{Exp}]\!](e[\textbf{var} := x \uplus \textbf{Exp}]))$

and $u = I_{3.8}[\![\textbf{Exp1}]\!](e[\textbf{var} := v \uplus \textbf{Exp}])$

Recursion is the same in both domains.

Exp $\longrightarrow$ **(if Exp1 Exp2 Exp3)**

$I_{3.12}[\![(\textbf{if Exp1 Exp2 Exp3})]\!](e) = v \uplus (\textbf{if Exp1 Exp2 Exp3})$

where $v = I_{3.12}[\![\textbf{Exp2}]\!](e) \sqcup I_{3.12}[\![\textbf{Exp3}]\!](e)$

$I_{3.8}[\![(\textbf{if Exp1 Exp2 Exp3})]\!](e) = v \uplus (\textbf{if Exp1 Exp2 Exp3})$

Where $v =$ If $fst(I_{3.8}[\![\textbf{Exp1}]\!](e)$ then $I_{3.8}[\![\textbf{Exp2}]\!](e)$ else $I_{3.8}[\![\textbf{Exp3}]\!](e)$

Conditionals are handled differently. Information of boolean type is lost in $I_{3.12}$. There is no way to select any branch of the alternative. The only way to avoid losing any data flow information is to join the results of the evaluation of both branches.

Semantics 3.12 appears to be to the one we seek. Close inspection of the clauses shows that semantics 3.12 gathers at least as much information as 3.8. This is the intuitively correct condition for using 3.12 as an approximation of 3.8. It would require tedious formalism like theorems 3.10 and 3.11 to define the detailed abstraction and concretization relations. We conjecture that 3.12 is an approximation of 3.1, that the approximations involved are monotonic, and therefore that theorem 2.51 enables us to compose them. More research is required on this point.

CHAPTER 4

THE TERMINATION PROBLEM

It should be obvious from the preceding chapters that abstract semantics are coarse homomorphic images of the related normal semantics. This enables abstract semantics to mock the operational and denotational properties of the program and therefore allows a compiler to get information about normal program behavior. The question arises whether undesirable properties of a program such as nontermination fall in its abstract semantics together with the desirable ones.

So we now ask when an abstract semantics terminates. This is a different question from when a normal semantics terminates for the following reason. Under a normal semantics it is the programmer's responsibility to ensure the code he writes terminates. If the code doesn't, this is the programmer's error and he is the one who will have to correct it. Under an abstract semantics, the situation is different. Data flow analysis is a compilation phase transparent to the programmer. If we want the compiler to be convenient to use, that phase should terminate for every program anybody could ever write, even in presence of the most awkward conditions. The programmer using the compiler is not aware of the existence of data flow analysis and can obviously not correct a condition he does not know exists. Therefore an abstract semantics is useful in data flow analysis only if we can prove that every program anybody could write will terminate when interpreted according to that semantics.

The sample data flow semantics in figure 3.12 shows how hard to obtain this last requirement is. One obvious way to enforce termination is to select approximations that will result in a provably terminating semantics. Unfortunately there are useful approximations that fundamentally introduce nontermination. For example consider the lambda–expression computing the factorial function:

$$fact(x) = \textbf{ if } is\_zero(x) \textbf{ then } 1 \textbf{ else } x \times fact(x-1)$$

Suppose we translate this definition into a LISP program, assuming we expand our LISP to include integers and an explicit join operator. According to 3.12 the above expression is semantically equivalent to the following:

$$fact(x) = 1 \sqcup (x \times fact(x-1))$$

This is a straightforward use of the semantics rule for **if** expressions given in figure 3.12. Notice how the recursion occurs unconditionally because the essence of the approximation is to remove every decision and replace it by the join of both alternatives. As a side–effect, we are no longer able to give any loop a termination condition, so a straightforward implementation of the semantics will lead to obviously nonterminating programs.

We cannot blame the semantics for this awkward property, because the approximation we chose is a very reasonable one. If we look for a coarser but terminating semantics, we find we can use top as an approximation of such ill-recursive expressions. This would be assuming that every subexpression of a program could propagate out of any recursion. The result is a correct but not very satisfactory solution to the problem because it amounts to performing no data flow analysis within recursive functions. It is very hard to imagine how an approximation coarser than the one in figure 3.12 but less drastic than top could introduce termination. Recall the transformation process from which we derived that semantics. Termination conditions were removed because we wanted the abstract semantics to be independent (in the sense just described) from the computations implied by the normal semantics. The reader can check this by comparing the semantics for if expressions in figure 3.8 and 3.12. This independence requirement implies the abstract semantics must have no explicit knowledge of the normal values taken by subexpressions in the program. From a compiler user point of view this means a data flow analyzer must be transparent, i.e. must behave correctly independently of actual run-time execution behavior. Termination conditions are removed because we cannot decide at compile time whether a loop will terminate according to run-time data. The information required to do so is not available. We only know that either the program stops, or it continues and we have to play safe, considering both alternatives at once. Approximations written according to this principle must depend on both alternatives at the same time. This implies they are very likely to generate an infinite loop, any branch of the conditional involving a recursive call being considered as providing relevant information at every iteration.

4.1 The Termination Properties of Tarski Least Fixed Point

Trying to find a better abstract semantics is not likely to yield termination. We have to look for a tool that comes from outside the formalism used to define the semantics. We propose to take advantage of some properties that are provable from information systems, but are unprovable within lambda–calculus itself. In particular the least fixed point theorem (proposition 2.35) will prove extremely powerful:

Proposition 2.35 (restated) The Least Fixed Point Theorem

Curry's fixed point combinator $\mathbf{Y} = \lambda f(\lambda x.f(xx))(\lambda x.f(xx))$ *is identical to Tarski's least fixed point operator* fix*, defined as follow:*

$$fix(f) = \bigsqcup_{i=0}^{\infty}\{f^i(\perp)\}$$

The least fixed point theorem states that $\mathbf{Y}$ and fix are equal from a denotational point of view. Operationally they are very different. We all know the behavior of $\mathbf{Y}$. On the other hand, fix works as a Pascal while loop:

```
$x := \bot$;
while $x \neq (f(x) \sqcup x)$ do $x := (f(x) \sqcup x)$;
```

This illustrates how fix will compute the fixed point of a function f. We start with $\bot$ stored in variable x. The function f is then repeatedly applied until the value converges. The requested fixed point is left in variable x. Of course, this may be an infinite loop. For instance if $\mathbf{Y}(f)$ is a nonfinitary element of the domain, it can be the join of an infinite increasing sequence without being itself a member of the sequence. However, there are other situations where it will terminate adequately. For instance, consider the fixed point of I, the identity function. If we compute with the help of Curry's $\mathbf{Y}$, we get $(\lambda x.xx)(\lambda x.xx)$, and then most lambda-calculus evaluators will get into an infinite loop. On the other hand if we run the above Pascal program we get the following steps:

1) $x := \bot$

2) test: $\bot = I(\bot) \sqcup \bot$?

3) yes, terminate the loop, leaving $\bot$ in variable x.

Although this is impressive, some readers will raise two objections. The first one is we assume we have an explicit use of $\bot$, an explicit use of the lattice theoretical join and a decidable universal equality over the lattice. All these features do not mix well with computable functions. In particular, the combination of having an explicit bottom together with a decidable equality implies we are able to solve the halting problem. The second objection is that it is not clear how we handle the form of nontermination that occurs with fix.

Let's take care of the first objection. We do not need the full power of all features required to code fix. Assume the function $c: B \longrightarrow A$ is an homomorphism from domain B to domain A. Assume there is an element $0 \in B$, different from $\bot_B$ such that $c(0) = \bot_A$. Assume also there are computable functions $eq: B \times B \longrightarrow$ **Boolean** and $join: B \times B \longrightarrow B$ such that:

$$
\begin{aligned}
eq(x,y) = \text{true} &\quad\Longrightarrow\quad c(x) = c(y) \\
join(x,y) = z &\quad\Longrightarrow\quad c(z) = c(x) \sqcup c(y)
\end{aligned}
$$

We can then code a function fix' that behaves as fix as follow:

$$fix'(f) = g(f, 0)$$

$$\text{wherec } g = \lambda f \lambda x.\ \text{if } eq(x, join(x, f(x))) \text{ then } x \text{ else } g(f, join(x, f(x)))$$

Notice eq need not correctly represent universal equality in every case. We only require that if $eq(x,y)$ is positive, then $c(x)$ must be equal to $c(y)$ without imposing any constraints on what happens when $eq(x,y)$ is false. As a result, if fix' terminates, then we had computed the representation in $B \longrightarrow B$ of the fixed point of a function in $A \longrightarrow A$. Our plan is if we have to perform data flow analysis using lattice A, we do not implement A in our algorithm. We find a suitable representation B for A where the abstractions of interesting functions in A happens to terminate in B. The art of data flow analysis lies in the correct selection of B, 0, *eq* and *join*. The above discussion is formalized into the next theorem, having at the same time a small optimization introduced into the code for fix'.

Theorem 4.1

Let A and B be domains. Let $\langle a, c\rangle$ be an abstraction from A to B, that is $\langle a, c\rangle$ makes the following graph commute:

$$\begin{tikzcd}
Lamb(B) \arrow[rr, "I_B"] & & B \arrow[d, "c"] \\
Lamb(A) \arrow[u, "Lamb(a)"] \arrow[r, "I_A"] & A \arrow[r, dashed, "r"] & A
\end{tikzcd}$$

Lamb is the functor inducing lambda-calculus syntax. The function I_A gives its semantics within domain A and I_B gives the semantics within domain B. Let 0 be an element of B such that $0 \neq \perp_B$ and $c(0) = \perp_A$. Let $eq: B \times B \longrightarrow$ **Boolean** *be such as $eq(x, y) = true \Longrightarrow c(x) = c(y)$. Let also fix' be defined as:*

$$fix'(f) = g(f, 0) \text{ wherec } g = \lambda f \lambda x.\ \text{if } eq(x, f(x)) \text{ then } x \text{ else } g(j, f(x))$$

We state that if $fix'(f)$ terminates, then all three conditions hold:

i) if $\langle a, c\rangle$ is a representation, then $c(fix'(f)) = fix(c(f))$

ii) if $\langle a, c\rangle$ is an increasing approximation, then $fix(c(f)) \sqsubseteq c(fix'(f))$

iii) if $\langle a, c\rangle$ is an increasing approximation, then $c(fix'(f)) \sqsubseteq fix(c(f))$

Clause i) is the way to transform exactly a nonterminating program into a terminating one using fix. Clauses ii) and iii) are used to create terminating approximations of programs, an option that is often convenient in data flow analysis.

Proof:

We shall first prove a few lemmas:

Lemma 4.2

For every integer i, we have:

$$f^i(\bot) \sqsubseteq f^{i+1}(\bot)$$

This is shown by induction on i.

Basis:

$\bot \sqsubseteq f(\bot)$ always true by definition of $\bot$.

Induction:

assume $f^{i-1}(\bot) \sqsubseteq f^i(\bot)$, by hypothesis f is generated from an approximable mapping and therefore is monotonic, (proposition 2.8 iv). Then we have

$$f(f^{i-1}(\bot)) \sqsubseteq f(f^i(\bot))$$

$$f^i(\bot) \sqsubseteq f^{i+1}(\bot)$$

This completes the induction and the proof of lemma 4.2

This lemma shows a very important fact about the behavior of fix. The sequence of values built from iteration of f over bottom is monotonicaly nondecreasing. Please keep this fact in mind for we will use it several times hereafter.

Corollary 4.3

$$\bigsqcup\{f^i(\perp) \mid i \leq j\} = f^j(\perp)$$

This follow from 4.2 by the law $x = x \sqcup y$ whenever $y \sqsubseteq x$.

Lemma 4.4

Assume that $f^j(\perp) = f^{j+1}(\perp)$ *then* $fix(f) = f^j(\perp)$.

Proof:

By induction on k, we show that for every $k \geq 0$ we have:

$$f^j(\perp) = f^{j+k}(\perp)$$

$k = 0$: trivial since $j + 0 = j$

$k > 0$: Assume $f^j(\perp) = f^{j+k-1}(\perp)$ then $f^{j+k}(\perp) = f^{j+1}(\perp) = f^j(\perp)$. It follow from idempotency of join that:

$$fix(f) = \bigsqcup_{i=0}^{\infty}\{f^i(\perp)\} = \bigsqcup\{f^i(\perp) \mid i \leq j\}$$

and corollary 4.3 implies that

$$fix(f) = \bigsqcup_{i=0}^{\infty}\{f^i(\perp)\} = f(\perp)$$

Now we return to the proof of 4.1. Remark that $fix'(f)$ can only terminate if convergence is obtained after finitely many iterations of f, that is if the following proposition holds:

$$eq(f^{i+1}(0), f^j(0)) = \text{ true for some } j$$

by defiition of eq, this implies:

(a) $$(c(f))^{j+1}(0) = (c(f))^j(0)$$

on the other hand we have by definitions of j and fix': $fix'(f) = f^j(0)$. if we apply c on both sides, we get:

(b) $$c(fix'(f)) = c(f^j(0))$$

Now we consider conclusions 4.1.i through 4.1.iii one after the other, starting with conclusion 4.1.i.

4.1.i we assume $\langle a, c\rangle$ is a representation. This allows us to write from (b):

$$c(fix'(f)) = (c(f))^j(0)$$

$$c(fix'(f)) = (c(f))^j(\perp_A) \qquad \text{by definition of } 0$$

$$c(fix'(f)) = fix(c(f)) \qquad \text{by (a) and 4.4}$$

4.1ii we assume that $\langle a, c\rangle$ is an increasing abstraction. This allows us to write from (b):

$$c(fix'(f)) \sqsubseteq (c(f))^j(c(0))$$

$$c(fix'(f)) \sqsubseteq (c(f))^j(\bot_A) \qquad \text{by definition of } 0$$

$$c(fix'(f)) \sqsubseteq fix(c(f)) \qquad \text{by (a) and 4.4}$$

4.1.iii We assume that $\langle a, c\rangle$ is a decreasing approximation. We proceed as in ii) with $\sqsupseteq$ substituted for $\sqsubseteq$ in the proof.

End of proof of 4.1. ▮

If we observe that $x = y \iff x \sqsubseteq y \wedge y \sqsubseteq x$, we can find another useful computable representation of fix. We need a computable function $lt: B \times B \longrightarrow$ Boolean such that $lt(x, y) = true$ always imply $c(x) \sqsubseteq c(y)$. Function eq in 4.1 can be defined as the conjunction of $lt(x, y)$ and $lt(y, x)$. However, lemma 4.2 implies one of the partial order is always true in the cases we are interested and need not be checked, this allows us to write:

Theorem 4.5

Let A and B and $\langle a, c\rangle$ satisfy the hypotheses of 4.1.

Let 0 be an element of B such that $0 \neq \bot_B$ and $c(0) = \bot_A$. Let $lt: B \times B \longrightarrow$ **Boolean** *be such as $lt(x, y) = true \Longrightarrow c(x) \sqsubseteq c(y)$. Let also fix' be defined as:*

$$fix'(f) = g(f,0) \text{ wherec } g = \lambda f \lambda x. \text{ if } lt(f(x),x) \text{ then } x \text{ else } g(j, f(x))$$

We state that if $f: B \longrightarrow B$ is such that $f^{i+1}(0) \sqsubseteq f^i(0)$ and such that $fix'(f)$ terminates, then all three conditions hold:

i) if $\langle a,c \rangle$ is a representation, then $c(fix'(f)) = fix(c(f))$

ii) if $\langle a,c \rangle$ is an increasing approximation, then $fix(c(f)) \sqsubseteq c(fix'(f))$

iii) if $\langle a,c \rangle$ is an increasing approximation, then $c(fix'(f)) \sqsubseteq fix(c(f))$

This is a restatement of 4.1 substituting $lt(f(x),x)$ for $eq(x, f(x))$ as the terminating condition of fix'.

Proof: Very similar to the proof of 4.1. Note that $fix'(f)$ can only terminate if convergence is obtained after finitely many iterations of f, that is if the following proposition holds for some j:

$$lt(f^{j+1}(0), f^j(0)) = \text{true for some } j.$$

This implies $c(f^{j+1}(0)) \sqsubseteq c(f^j(0))$ by definition of lt. On the other hand, the hypothesis that $lt(f^i(0), f^{i+1}(0))$ implies that $c(f^j(0)) \sqsubseteq c(f^{j+1}(0))$. We can now state that

(c) $$c(f^{j+1}(0)) = c(f^j(0))$$

on the other hand we have by definition of j:

(d) $$fix'(f) = f^j(0)$$

equation (c) is identical to (a) and equation (d) is identical to (b), therefore conclusions 4.5.i through 4.5.iii are shown as the corresponding conclusions of 4.1. End of proof of 4.5.▮

The inconvenient of 4.5 over 4.1 is we need to have c monotonic and f to satisfy $f^j(0) \sqsubseteq f^{j+1}(0)$ to be able to use it. Fortunately these restictions are reasonable. The principal advantage of 4.5 is we can more easily determine when fix' will terminate with the help of the following definition:

Definition 4.6 The Ascending Chain Condition

The domain A satisfies the ascending chain condition if any monotonicaly nondecreasing chain of elements $\{a_i\}$ *in A converges after some integer value* n*, that is* $a_m = a_n$ *whenever* $m \leq n$*. Assume also there is a domain B and a function* $lt: B \times B \longrightarrow$ **Boolean** *that satisfy the hypotheses of 4.5. We will say that lt preserves the ascending chain condition with respect to a class of functions C if and only if for every function* $f: B \longrightarrow B$ *in class C the following proposition holds:*

$$\forall i, \quad lt(f^i(0), f^{i+1}(0))$$

The significance of this condition follows from lemma 4.2 where we show that the iteration of f on bottom involved in fixed pointing builds a nondecreasing sequence of values. If the domain of this sequence satisfies the ascending chain condition, sooner or later we will have performed enough iterations to be able to apply lemma 4.4 and terminate the search for a fixed point. This will be true for every f since every sequence converges. If in addition domain B and function lt preserve the ascending chain condition, then theorem 4.5 will apply to prove termination of every fixed point expression. Of course not all useful semantics will be proven to terminate in this way. Part of the art of data flow analysis is to select domains A and B and function lt in such a way that the class C of functions generated by the semantics preserves the ascending chain condition in order to have the proper termination properties.

4.2 The Semantics of Dynamically Scoped Lambda–Calculus

Many useful domains satisfies the ascending chain condition. Every flat domain such as **BIT**, integers and boolean, and every domain with finitely many elements satisfies this condition. This is also true of cartesian products and disjoint unions of domains satisfying the ascending chain condition. Together, theorem 4.5 and definition 4.6 form a very powerful tool to force termination of semantics. We still have to verify how useful this power is. The best way is to provide an example. We plan to show that dynamically scoped lambda–calculus semantics always terminates if fixed points are implemented as in 4.5. This is a significant example because there are many dynamically scoped languages on the market.

Semantics 4.7 Dynamically Scoped Lambda–Calculus

Domains

Var : *The syntactic domain of variables*

Ex : *The syntactic domain of expressions*

$\mathbf{Val}_{4.7}$: *The domain of values*

$$\mathbf{Val}_{4.7} = \mathbf{E}_{4.7} \longrightarrow \mathbf{Val}_{4.7} \longrightarrow \mathbf{Val}_{4.7}$$

$\mathbf{E}_{4.7}$: *The domain of environments*

$$\mathbf{E}_{4.7} = \mathbf{Var} \longrightarrow \mathbf{Val}_{4.7}$$

Functionality

$I_{4.7}\colon \mathbf{Ex} \longrightarrow \mathbf{E}_{4.7} \longrightarrow \mathbf{Val}_{4.7}$

Semantics

Exp $\longrightarrow$ **var**

$$I_{4.7}[\![\mathtt{var}]\!](e) = e[\![\mathtt{var}]\!]$$

Exp $\longrightarrow$ **λvar.Exp1**

$$I_{4.7}[\![\lambda\mathtt{var}.\mathtt{f}]\!](e) = \lambda d\lambda x.I_{4.7}[\![\mathtt{f}]\!](d[\mathtt{var}\ :=\ x])$$

Exp $\longrightarrow$ **Exp1(Exp2)**

$$I_{4.7}[\![\mathtt{g(f)}]\!](e) = I_{4.7}[\![\mathtt{g}]\!](e)(e)(I_{4.7}[\![\mathtt{f}]\!](e))$$

End of semantics 4.7. ▮

This requires some explanation. The characteristic of dynamically scoped languages is their mechanism for handling free variables. This is expressed by the domain equation $\mathbf{Val}_{4.7} = \mathbf{E}_{4.7} \longrightarrow \mathbf{Val}_{4.7} \longrightarrow \mathbf{Val}_{4.7}$. Functional values do not bind their free variables when they are declared, free variables are bound to the environment of the caller when the function using them is called. For this reason functional values require as an extra implicit parameter: the environment that will provide the call-time free variable binding. This is expressed by the equation pair:

$$I_{4.7}[\![\lambda \mathtt{var}.\mathtt{f}]\!](e) = \lambda d \lambda x. I_{4.7}[\![\mathtt{f}]\!](d[\mathtt{var} := x])$$

$$I_{4.7}[\![\mathtt{g}(\mathtt{f})]\!](e) = I_{4.7}[\![\mathtt{g}]\!](e)(e)(I_{4.7}[\![\mathtt{f}]\!](e))$$

The value of an abstraction requires two parameters: the first parameter is the environment d that provides the values of free variables, and the second parameter is the actual value x for the formal parameter $\mathtt{var}$ bound by the abstraction. The rule for application has to pass the environment e twice to evaluate the function component. The first time e is passed to get is required to evaluate the semantic value of g, then e is passed again to bind the free variables that may occur in the abstraction that happens to be the value of g.

4.2.1 The Tree Traversal Semantics

The interesting aspect of this semantics is the function $I_{4.7}$ itself is defined with an equation system we want to fixed point. It follows from this observation that the termination properties of lambda-expressions depend on which fixed point operator we chose, **Y** or fix. If we chose **Y**, we have the usual termination properties of applicative or normal order evaluation strategies. We should ask what happens if we use fix instead. The answer is not trivial. Some preliminary results require us to rewrite the interpretation function $I_{4.7}$ as a tree traversal function. The tree will be a data structure representing the "computation", that is the operations required to evaluate a program.

Semantics 4.8 Evaluation as Tree Traversal

Domains

T : *the domain of computation trees*

$$\mathbf{T} = (\mathbf{L} \times \mathit{Leaf}) + (\mathbf{L} \times \mathbf{T} \times \mathbf{T} \times \mathbf{T})$$

L : *the domain of labels of computation trees*

$$\mathbf{L} = \mathbf{E}_{4.8} \longrightarrow \mathbf{L} \longrightarrow \mathbf{T}$$

Ex : *the domain of lambda-expressions*

Var : *the domain of names of variables*

$\mathbf{E}_{4.8}$: *the domain of environments*

$\mathbf{E}_{4.8} = \mathbf{Var} \longrightarrow \mathbf{L}$

Functionality

Tree : *the interpretation function of lambda-expressions*

$$\mathit{Tree} :: \mathbf{Ex} \longrightarrow \mathbf{E}_{4.8} \longrightarrow \mathbf{T}$$

Semantics

Exp $\longrightarrow$ **var**

$$\mathit{Tree}[\![\mathtt{var}]\!](e) = \langle e[\![\mathtt{var}]\!], \mathit{Leaf}\rangle$$

Exp $\longrightarrow$ **λvar.Exp1**

$$\mathit{Tree}[\![\lambda\mathtt{var}.\mathtt{f}]\!](e) = \langle \lambda d\lambda x.(\mathit{Tree}[\![\mathtt{f}]\!](d[\mathtt{var} := x])), \mathit{Leaf}\rangle$$

Exp $\longrightarrow$ **Exp1(Exp2)**

$$\mathit{Tree}[\![\mathtt{g(f)}]\!](e) = \langle fst(z), x, y, z\rangle$$

$$\begin{aligned} \textit{Where } x &= \mathit{Tree}[\![\mathtt{g}]\!](e) \\ y &= \mathit{Tree}[\![\mathtt{f}]\!](e) \\ z &= fst(x)(e)(fst(y)) \end{aligned}$$

End of semantics 4.8. ▮

This requires some explanations. The label of the trees should be understood as the "real" values of the lambda–expressions. Therefore domain **L** should be understood as a representation of $\mathbf{Val}_{4.7}$ as defined in semantics 4.7. The trees are the conjunction of the value of an expression and the computational process of evaluation as implemented in most dynamically scoped LISP compilers, the evaluation process being similar to the traversal of that tree in postfix order. Let's explain this idea in more detail. If we are at a leaf node, the expression value is considered known and the label of the leaf is the value. For instance when the expression is an abstraction, we assume we know its value and a compiler will consider it to be the entry point of the code that implements it. The tree is a leaf node labeled with the value. Similarly, if the expression is a variable, its value is known from the environment and the single leaf tree is immediately constructed. On the other hand if we evaluate an application $g(f)$, then we require a more elaborate evaluation process. The tree has three branches: one to evaluate f, one to evaluate g and one to evaluate the body of the value of f when g is passed to it as a parameter. The label of that third child is the "real value" of the application and is taken as the label of the whole tree. The tree walking strategy is therefore to evaluate f first, then to evaluate g and at last to pass the value of g to the value of f as a parameter.

The claim that semantics 4.8 is a representation of 4.7 is left unproven. As observed in chapter 3, such a proof is enormous and tedious and would lead us too far from dat? flow analysis. Most readers should nevertheless agree with the statement on the basis of a pair wise comparison of the equations involved. The following categorical diagram shows what kind of representation we have in mind:

$$\begin{tikzcd}
Lamb(\mathbf{L}) \arrow[rr, "\lambda x \lambda e.\, Tree[\![x]\!](e)"] & & \mathbf{L} \arrow[d, "c"] \\
Lamb(\mathbf{Val}_{4.7}) \arrow[u, "a"] \arrow[rr, "I_{4.7}"] & & \mathbf{Val}_4.7
\end{tikzcd}$$

In this diagram, *Tree* and $\mathbf{L}$ are as defined in 4.8 and $I_{4.7}$ and $\mathbf{Val}_{4.7}$ are as defined in 4.7. The functor *Lamb* map the domain $\mathbf{D}$ to the domain $\mathbf{Ex} \times (\mathbf{Var} \longrightarrow \mathbf{D})$. Some uncurrying operations are left implicit in the diagram. Relations a and c are the abstractions and concretizations as suggested in semantics 4.7 and 4.8.

4.2.2 The Termination Properties of Dynamically Scoped Lambda–Calculus

The reader should observe that the label generated by the abstraction rule does not depend on the environment parameter nor on anything other than the abstraction body. No other semantic rule will spontaneously produce labels different from the ones generated in this way. They only propagate labels already present in the semantics. This establishes a one–to–one correspondence between abstractions and labels occurring as values. Therefore, within the context of a given program, the set of all elements of **L** that are actually used in the evaluation is a finite set since there are only finitely many abstractions. If we complete that set with a top and bottom element, we obtain a finite domain **D** such that the value of the program must be an element of **D**. This is formalized in the theorem below:

Theorem 4.9

Let P be a program and e an environment, let **D** *be the domain constructed from the set* $\{fst(Tree[\![\mathbf{a}]\!](e)) \mid \mathbf{a}$ *is a lambda–abstraction,* $\mathbf{a}$ *occurs in* $P\}$ *by adding* $\bot$ *and* $\top$, *and let* **N** *be the domain of all variables that actually occur in* P. *Then for all subexpressions* $\mathbf{f}$ *occurring in* P, *if* d *is an environment such that* $d(v) \in \mathbf{D}$, *then* $fst(Tree[\![\mathbf{f}]\!](d))$ *belongs to* **D**.

Proof:

We use induction on the syntax of $\mathbf{f}$.

Basis:

Case a: $\mathtt{f}$ is the variable var.

We have te equality $\mathit{Tree}[\![\mathtt{var}]\!](d) = \langle d[\mathtt{v}], \mathit{Leaf}\rangle$ and $d[\mathtt{v}]$ has type $\mathbf{D}$ by hypothesis on d.

Case b: $\mathtt{f}$ is the abstraction $\lambda\mathtt{var.g}$.

This implies that $\lambda\mathtt{var.g}$ occurs in P and $\mathit{fst}(\mathit{Tree}[\![\lambda\mathtt{var.g}]\!](d)) \in \mathbf{D}$ by definition of $\mathbf{D}$.

Induction : $\mathtt{f}$ is the application $\mathtt{g(h)}$

By the syntactic induction hypothesis both labels of $\mathit{Tree}[\![\mathtt{g}]\!](d)$ and of $\mathit{Tree}[\![\mathtt{h}]\!](d)$ are in $\mathbf{D}$.

We need to prove the third child $\mathit{fst}(\mathit{Tree}[\![\mathtt{g}]\!](d))(d)\mathit{fst}(\mathit{Tree}[\![\mathtt{h}]\!](d))$ has a label in $\mathbf{D}$. There are three cases depending on the value of $\mathit{fst}(\mathit{Tree}[\![\mathtt{g}]\!](d))$.

Case 1—$\mathit{fst}(\mathit{Tree}[\![\mathtt{g}]\!](d)) = \perp$.

Recall $\perp = \lambda x.\perp$ by 2.34.i and $\perp = \langle\perp, \perp\rangle$ by 2.28.i. Therefore the label of the third child will be $\perp$, a value that belongs to $\mathbf{D}$.

Case 2—$\mathit{fst}(\mathit{Tree}[\![\mathtt{g}]\!](d)) = \top$.

Recall $\top = \lambda x.\top$ by 2.34.ii and $\top = \langle\top, \top\rangle$ by 2.28.ii. Therefore the label of the third child will be $\top$, a value that belongs to $\mathbf{D}$.

Case 3—$fst(\mathit{Tree}[\![\mathbf{g}]\!](d)) = \lambda c\lambda x.\mathit{Tree}[\![\mathbf{u}]\!](c[\mathbf{v} \;:=\; x])$ for some expression $\mathbf{u}$ and variable $\mathbf{v}$.

Let $k = fst(\mathit{Tree}[\![\mathbf{h}]\!](d))$. Recall the rule of beta substitution. It implies that $\mathit{Tree}[\![\mathbf{u}]\!](d[\mathbf{v} \;:=\; k])$ is the third child of $\mathit{Tree}[\![\mathbf{g}\,(\mathbf{h})]\!](d)$. It is a tree generated from a subexpression occurring in program P and an environment of type $\mathbf{N} \longrightarrow \mathbf{D}$. We want to show its label is in $\mathbf{D}$. We have to consider whether the tree is finite or infinite. If it is finite, the proof will take the form of an induction on the length of the starting from the root of the tree and descending along the third child down to the leaf. If the tree is infinite, a special treatment is given. This gives us three subcases:

Case 3.i—Basis of the induction. We are at a leaf.

Immediate from an argument identical to the one given for the basis (cases a and b) of the induction on expression syntax above.

Case 3.ii—Induction step. We are at a parent node, the path is finite.

The induction hypothesis is the label of the third child must be in $\mathbf{D}$. The definition of *Tree* implies the label of the parent is the label of that third child. This completes the induction on the length of the right-most path in the tree.

Case 3.iii—Special treatment. We are at a parent node, the path is infinite.

Descending one node down the tree along that path corresponds to parameter passing. Therefore descending one node down corresponds to performing some leftmost beta contraction in the lexically scoped metalanguage that is used to define the *Tree* function. Repeated beta contraction is defined as finding the fixed point of some function defined over the domain of lambda-expression. If we work out the increasing sequence of trees that compute the required Tarski fixed point we get:

(0) $$\perp$$

(1) $$\begin{tikzcd} \langle l_1, f_1, a_1, - \rangle \arrow[d, dash] \\ \perp \end{tikzcd}$$

(2) $$\begin{tikzcd} \langle l_1, f_1, a_1, - \rangle \arrow[d, dash] \\ \langle l_1, f_2, a_2, - \rangle \arrow[d, dash] \\ \perp \end{tikzcd}$$

(i) $$\begin{tikzcd} \langle l_1, f_1, a_1, - \rangle \arrow[d, dash] \\ \langle l_1, f_2, a_2, - \rangle \arrow[d, dash] \\ \cdots\cdots\cdots \arrow[d, dash] \\ \langle l_1, f_i, a_i, - \rangle \arrow[d, dash] \\ \perp \end{tikzcd}$$

In the above sequence every f_i is the tree for the evaluation of a function part, every a_i is the tree for the evaluation of an argument part. For every pair f_i, a_1, the beta contraction of "$f_i(a_i)$" results in the application "$f_{i+1}(a_{i+1})$". The sequence of trees is indeed the infinite sequence of beta contractions we are looking for. In every tree, we have by definition of the function *Tree* the relation $l_1 = fst(\perp)$ and by proposition 2.28.i $l_1 = \perp$. This is an element of **D** as required. If t is the limit tree resulting of the join of all those trees, proposition 2.28.iv together with idempotence of join shows $fst(t) = l_1 = \perp$. This completes the proof of the case of the infinite tree. ▮

Corollary 4.10

Theorem 4.9 will stay true if we substitute the function $I_{4.7}$ for Tree in its statement.

Proof:

Follows from the above theorem by homomorphism of the semantics. ▮

In other words, the fact there are only finitely many labels used in the tree traversal semantics for a given program implies there are finitely many functional values used in the normal semantics for the same program. Refer now to semantics 4.7. We can show that using fix as the fixed point operator in the definition of $I_{4.7}$ will enforce termination of $I_{4.7}$ for every program. This follow from the type

$\mathbf{Ex} \longrightarrow \mathbf{E}_{4.7} \longrightarrow \mathbf{Val}_{4.7}$ of $I_{4.7}$. The domain $\mathbf{E}_{4.7}$ is $\mathbf{Var} \longrightarrow \mathbf{Val}_{4.7}$. Within a given program context, we can substitute the domain $\mathbf{N}$ of variables that effectively occur in the program for the domain **Var** and the domain $\mathbf{D}$ as defined in 4.9 and 4.10 for $\mathbf{Val}_{4.7}$. Therefore $\mathbf{E}_{4.7}$ become $\mathbf{N} \longrightarrow \mathbf{D}$, a finite domain since there are only finitely many functions from a finite domain to another. Similarly domain **Ex** can be considered a finite domain if we consider only expressions that actually occur in the program. The type of the semantic function $I_{4.7}$ can be considered to be $\mathbf{Ex} \longrightarrow \mathbf{E}_{4.7} \longrightarrow \mathbf{D}$, a finite domain. Since all finite domains satisfy the ascending chain condition, the use of *fix* in the computation of $I_{4.7}$ will insure termination.

Let's make one remark on this astonishing result. In semantics 4.7 we do not introduce integers, lists or any other data type than lambda-definable dynamically scoped functions. Will the introduction of such data types affect the termination properties? Obviously yes. We had shown that the domain $\mathbf{Val}_{4.7}$ defined by the equations:

$$\mathbf{Val}_{4.7} = \mathbf{E}_{4.7} \longrightarrow \mathbf{Val}_{4.7} \longrightarrow \mathbf{Val}_{4.7}$$

$$\mathbf{E}_{4.7} = \mathbf{Var} \longrightarrow \mathbf{Val}_{4.7}$$

can be replaced by a finite domain because in the context of any given program we use only a finite portion of it. Of course the finite portion in question changes from program to program so we need the full domain to define the general semantics, but as far as we are concerned with a specific program, we can always find a terminating implementation for it. If we introduce an additional data type in the semantics the equations become:

$$\mathbf{V} = \mathbf{Dt} + (\mathbf{E} \longrightarrow \mathbf{V} \longrightarrow \mathbf{V})$$

$$\mathbf{E} = \mathbf{Var} \longrightarrow \mathbf{V}$$

where **Dt** stands for the new data type. Theorems 4.9 and 4.10 become proofs so that we use only a finite portion of $\mathbf{E} \longrightarrow \mathbf{V} \longrightarrow \mathbf{V}$ within a given program context but say nothing about what portion of **Dt** is used. If **Dt** happens to be infinite, we are no longer able to claim that we are using a finite portion of **V** and our termination result no longer holds. This shows that data types added to dynamically scoped lambda-calculus are not just syntactic sugar but really introduce new features to the language; they cannot be emulated with already available functions. On the other hand, data flow analysts will be pleased to notice they don't need to introduce an infinite data type to the language. They are usually concerned with sets of variables and expressions occurring in the program. Those domains are finite and for this reason do not disturb the termination properties of the language.

4.3 The Computation of Tarski Least Fixed Points

Now we look at some implementation guidelines for *fix*. We already know *fix* constructs an ascending chain. Each element of the chain has type $\mathbf{Ex} \longrightarrow \mathbf{E}_{4.7} \longrightarrow \mathbf{Val}_{4.7}$ because this is the type of $I_{4.7}$. Remember the whole chain converges to $I_{4.7}$ and all elements in the chain have the same type as $I_{4.7}$. Such elements can be represented by a table of entries, each entry containing three fields.

1— The first field is a representation of a subexpression in the program, say a pointer to a node in the parse tree.

2— The second field is an environment. In practice, a pointer to a table data structure associating a value to each variable may be used.

3— The third field is a representation of an element of $\mathbf{Val}_{4.7}$, usually the address of the entry point to a routine.

Such table represents an object in $\mathbf{Ex} \longrightarrow \mathbf{E}_{4.7} \longrightarrow \mathbf{Val}_{4.-11}$ because it contains a triple $\langle c, e, v\rangle$ such that c represents an object in $\mathbf{Ex}$, e represents an object in $\mathbf{E}_{4.7}$, and v is in $\mathbf{Val}_{4.7}$. This represents the required mapping because we will not allow the same pair e, c to be duplicated in two distinct entries $\langle e, c, v\rangle$ and $\langle e, c, v'\rangle$ in the table.

The fixed point evaluation would proceed as follows:

a) We create an initial table T_0 mapping every pair $\langle e, c\rangle$ of environment and subexpression pair to the value $\perp$.

b) If we rewrite the definition of $I_{4.7}$ to make the fixed point operator explicit, we will obtain an iteration function that computes a new table T_1 from table T_0. This iteration function will construct the next element in the ascending chain.

c) We repeat application of the iteration function to obtain tables T_2, T_3 etc. until the relation $T_{n+1} = T_n$ becomes true for some n. When this relation is true, $T(n)$ is a table representing $I_{4.7}$.

Unfortunately the plan we had just outlined is not applicable in practice. Tables implementing objects in $\mathbf{Ex} \longrightarrow \mathbf{E}_{4.7} \longrightarrow \mathbf{Val}_{4.7}$ are much too large. If n is the number of variables in the program, m the number of subexpressions and k the number of abstractions, the size of $\mathbf{Val}_{4.7}$ is $k + 2$, because there are k abstractions plus $\top$ and $\bot$ in the actually used subset of $\mathbf{Val}_{4.7}$. The size of $\mathbf{E}_{4.7} = \mathbf{N} \longrightarrow \mathbf{Val}_{4.7}$ is $(k + 2)^n$ and the number of triples $\langle c, e, v\rangle$ in a table representing an element of $\mathbf{Ex} \longrightarrow \mathbf{E}_{4.7} \longrightarrow \mathbf{Val}_{4.7}$ is the number of possible pairs in $(\mathbf{Ex} \times \mathbf{E}_{4.7})$: $(m \times (k+2)^n)$. Therefore the number of entries in a table tends to grow exponentially in the number of variables occurring in the program. We will devote the rest of this chapter to the search of a more acceptable solution.

This procedure for fix uses a complete representation of a function, that is, there must be an entry $\langle x, f(x)\rangle$ in the table for every argument x. If we use the terminology of Scott's domains, the operator fix operates explicitly on all the information required to describe an element in the domain. We are not using a partial element that describes only a portion of it. On the other hand, $\mathbf{Y}$ does not have such a requirement. For example if we recursively evaluate $fact(2)$ where $fact$ is the factorial function, we get $fact(2) = 2 \times fact(1)$, $fact(1) = 1 \times fact(0)$ and $fact(0) = 1$. That use of $\mathbf{Y}$ will identify that we only need to know $fact(0)$ and $fact(1)$ in order to evaluate $fact(3)$. For a given call $f(x)$ the usual mechanisms of recursion will identify the set of all values y such that the value $f(x)$ depends on $f(y)$. Using the terminology of Scott, we only need to know a partial element approximating f in order to know $f(x)$ for a given x. We want an algorithm that will compute a fixed point while keeping both the ability of $\mathbf{Y}$ to use only the minimum required amount of information and the termination properties of fix.

Theorem 4.11

Assume that domain $\mathbf{A}$ *is finite and domain* $\mathbf{V}$ *satisfies the ascending chain condition. Let* $j: (\mathbf{A} \longrightarrow \mathbf{V}) \longrightarrow (\mathbf{A} \longrightarrow \mathbf{V})$ *be defined as:*

$$j = \lambda f \lambda x. g(f(h_1 x)) \ldots (f(h_n x))(x)$$

meaning that the function $f = fix(j)$ *will contain* n *recursive calls of the form* $f(h_1 x) \ldots f(h_n x)$.

Let function fix'' *be defined as the following Pascal-like program:*

```
Function  $fix''(t: \mathbf{A} \longrightarrow (\mathbf{V} + \{not\_found\}),$
                    $j: (\mathbf{A} \longrightarrow \mathbf{V}) \longrightarrow (\mathbf{A} \longrightarrow \mathbf{V}),\quad x: \mathbf{A}): \mathbf{V};$
Begin
    If $t(x) \neq not\_found$  then return $t(x)$
        else
        Begin
            $temp2 := \bot_{\mathbf{V}}$;
            Repeat
                $temp1 := temp2$;
                $temp2 = j(fix''(t[x := temp2])(j))(x)$;
            until $temp2 = temp1$;
            return $temp2$;
        end;
end.
```

Then for every x we have:

i) If the evaluation of x, $x_1 \ldots x_n$, $t(x)$ and $g(x_1)\ldots(x_n)(x)$ terminate, then the evaluation of $fix''(t)(j)(x)$ also terminate.

ii) If t is such that we have either $t(x) \sqsubseteq fix(j)(x)$ or $t(x) = not\_found$ then $fix''(t)(j)(x) \sqsubseteq fix(j)(x)$.

iii) If t is such that we have either $t(x) = fix(j)(x)$ or $t(x) = not\_found$ then $fix''(t)(j)(x) = fix(j)(x)$.

iv) $fix''(\lambda x.not\_found)(j) = fix(j)$.

These statements require some explanations before we attempt proving them. Function j is the iteration function used to define some fixed point $f = fix(j)$. The function fix'' gives a method for evaluating $fix(j)$ when j is of the specified form. It accepts a parameter t that is a table similar to the one usually used when fix is the fixed point operator except that we allow the option of having no entry $\langle x, t(x)\rangle$ in the table for a given x. The element not_found represents this possibility.

Function fix'' calls itself recursively in the fashion of **Y**. If there is no entry in the table for x, then fix'' calls itself recursively as **Y** would do except that an entry for x is created in the table, estimating $f(x)$ as $\perp$. This is done in a loop. When the call returns, we match the resulting value with the value stored in the table to see if we had reached the fixed point. If this occurs, we are done. Otherwise, the new value is stored in the table, superseding any entry already there, and the recursive call is tried again. This is similar to the behavior of fix because if deeper in the call chain we discover a value is already stored for x in the table, we return that value as a basis for computing the next approximation. This builds an increasing sequence, starting at bottom and going up until convergence.

Because the recursive calls are generated as in $\mathbf{Y}$, all the entries in the table are relevant for the computation, keeping the table small. On the other hand if we detect that $\mathbf{Y}$ is going into a loop because the value of $f(x)$ depends on itself, we revert to the behavior of fix to insure termination. We keep the best of the two worlds.

A clever reader will remark that more optimizations can be done to that program. When we detect the value returned is independent from the values stored in the table, for example when the termination condition of a loop is met, then the second iteration of the repeat loop will bring the same value as the first one. We can flag the returned value as final to avoid useless iterations of the repeat fix-like loop. Similarly if we store a final value in the table, or if we reach a fixed point and store its value, we can mark it as final in the table to avoid useless recomputations. Those optimizations are unessential to the proof of the theorem, leading us too far away from foundational work. We simply give a hint in that direction, leaving the exact formulation of the resulting program to the reader.

Proof of theorem 4.11

We will restate all clauses 4.11.i trough 4.11.iv and write their proof immediately after their statement.

4.11.i— If the evaluation of x, $x_1 \ldots x_n$, $t(x)$ and $g(x_1)\ldots(x_n)(x)$ terminates, then the evaluation of $fix''(t)(j)(x)$ also terminates.

We want to evaluate $fix''(t)(j)(x)$ for some t. If a recursive call to fix'' occurs, then $t(x)$ must be *not_found* because recursive calls only occur in that branch of the if statement. But the recursive call is made using the table $t[x := temp2]$ and *temp*2 is different from *not_found*. Therefore the height of the tree of nested recursive calls is bounded by the number of elements in the set $S(t) = \{x \mid t(x) = not\_found\}$ because at each call that number is reduced by one. The set $S(t)$ must be finite for this is a subset of domain **A** that is finite. This allows us to make an induction on the height of the tree of nested recursive calls, excluding infinite trees.

Basis: there are no recursive calls.

This can occur only if $t(x) \neq not\_found$, for otherwise we take the "reapeat loop" branch of the **if** statement and that loop comtains a recursive call that is going to be executed at least once. We have $fix''(t)(j)(x) = t(x)$. The evaluation of $t(x)$ terminates by hypothesis on $t(x)$.

Induction: There is at least one recursive call.

We must have $t(x) = not\_found$ because otherwise we take the branch of the **if** statement that contains no recursive call. The termination condition of the repeat loop implies we have the equality:

$$fix''(t)(j)(x) = z = j(fix''(t[x := z])(j))(x)$$

where z is the value contained in *temp*2 after the last iteration of the loop. This z is the limit of the sequence z_n defined as follow:

$$z_0 = \bot$$

$$z_{m+1} = j(fix''(t[x := z_m])(j))(x)$$

The induction hypothesis assumes that each individual recursive calls in the evaluation of the sequence z_m terminates. The length of that sequence is finite because an argument similar to the proof of 4.2 shows this is an increasing sequence and domain **V** satisfies the ascending chain condition. This is enough to ensure termination and to complete the proof of 4.11.i.

4.11.ii— If t is such that we have either $t(x) \sqsubseteq fix(j)(x)$ or $t(x) = \mathit{not\_found}$ then $fix''(t)(j)(x) \sqsubseteq fix(j)(x)$.

We use an induction on the height of the tree of recursive calls as in 4.11.i.

Basis: there are no recursive calls.

Again we can assume that $t(x) \neq \mathit{not\_found}$ and $fix''(t)(j)(x) = t(x)$. The hypothesis imposed on t implies trivially $fix''(t)(j)(x) \sqsubseteq fix(j)(x)$.

Induction: there is at least one recursive call.

We must have $t(x) = not\_found$, the termination condition of the repeat loop implies we have the equality:

$$fix''(t)(j)(x) = z = j(fix''(t[x := z])(j))(x)$$

where z is the value contained in *temp2* after the last iteration of the loop. This z is the limit of the sequence z_n defined as follow:

$$z_0 = \perp$$
$$z_{m+1} = j(fix''(t[x := z_m])(j))(x)$$

We use an induction on the number m of iterations of the repeat loop.

Basis: $m = 0$.

We have $z_0 = \perp \sqsubseteq fix(j)(x)$ trivialy.

Induction: we consider z_{m+1}.

Let $s = t[x := z_m]$. The induction on m assumes that $z_m \sqsubseteq fix(j)(x)$. This implies we have:

$$(*) \qquad s(x) \sqsubseteq fix(j)(x) \quad \vee \quad s(x) = not\_found$$

because of the hypothesis imposed on t. The definition of z_{m+1} is rewritten as $z_{m+1} = j(fix''(s)(j))(x)$. We expand the definition of j:

$$(**) \qquad z_{m+1} = g(fix''(s)(j)(h_1 x)) \ldots (fix''(s)(j)(h_n x))(x)$$

By $(*)$ above, we have either $s(x) \sqsubseteq fix(j)(x)$ or $s(x) = not\_found$. The induction on the height of the tree of recursive calls assumes that:

$$(***) \qquad \forall i, \quad fix''(s)(j)(h_i x) \sqsubseteq fix(j)(h_i x)$$

because $fix''(s)(j)(h_i x)$ is a call to fix'' recursively generated from $fix''(t)(j)(x)$. Using monotonicity of lambda–definable functions we can conclude from relations $(**)$ and $(***)$ the inequality:

$$z_{m+1} \sqsubseteq g(fix(j)(h_1 x)) \ldots (fix''(j)(h_n x))(x)$$

Folding back the definition of j, we get:

$$z_{m+1} \sqsubseteq j(fix(j))(x) = fix(j)(x)$$

This completes the induction on m and implies for all m, $z(m) \sqsubseteq fix(j)(x)$. If we use an argument similar to the proof of 4.2, we can show the sequence $z(m)$ is monotonicaly increasing. Therefore it must converge after some finite number of iterations because the domain $\mathbf{V}$ satisfies the ascending chain condition. This gives the result:

$$fix''(t)(j)(x) = z_m \sqsubseteq fix(j)(x)$$

completing the proof of 4.11.ii.

4.11.iii— If t is such that either $t(x) = fix(j)(x)$ or $t(x) = not\_found$ then $fix''(t)(j)(x) = fix(j)(x)$.

There are two cases

Case 1—$t(x) \neq not\_found$.

Then $fix''(t)(j)(x) = t(x) = fix(j)(x)$ by the hypothesis on imposed on t.

Case 2—$t(x) = not\_found$.

The termination condition of the repeat loop implies we have the equality

$$(*) \qquad fix''(t)(j)(x) = z(x) = j(fix''(t[x := z(x)])(j))(x)$$

where z can be thought as a combinator depending on j and t. We omit to write the dependencies on t and j as $z(t)(j)(x)$ to avoid unduly complicated syntax. We also have $z(x) \neq not\_found$, therefore we have:

$$fix''(t[x := z(x)])(j)(x) = t[x := z(x)](x) = z(x)$$

This implies by transitivity of equality:

$$(**) \qquad fix''(t[x := z(x)])(j)(x) = j(fix''(t[x := z(x)])(j))(x)$$

Let's define the diagonal set of pairs $D(j)(t)$ as follows:

$$D(t)(j) = \{(x,y) \mid y = fix''(t[x := z(x)])(j)(x)\}$$

The set $D(t)(j)$ is called the diagonal set because it diagonalizes the function $\lambda y \lambda x.fix''(t[x := z(x)](j)(y)$. Taking D as a function, (*) and (**) will imply the relation:

$$D(t)(j)(x) = j(D(t)(j))(x) = fix''(t)(j)(x)$$

that is $D(t)(j) = fix''(t)(j)$ is a fixed point of j. Considering the hypothesis of 4.11.iii implies the hypothesis of 4.11.ii, we can write:

$$fix''(t)(j)(x) \sqsubseteq fix(j)(x)$$

On the other hand, fix is the least fixed point operator, therefore the inequality can be reversed. This shows

$$fix''(t)(j)(x) = fix(j)(x)$$

and this completes the proof of 4.11.iii.

4.11.iv— $fix''(\lambda x.not\_found)(j) = fix(j)$.

This follows immediately from 4.11.iii, taking $\lambda x.not\_found$ as t.

End of proof of 4.11 ∎

CHAPTER 5
ABSTRACT SEMANTICS OF LEXICALLY SCOPED LAMBDA–CALCULUS

This chapter is devoted to the problem of enforcing termination of lexically scoped lambda–calculus. In the preceding chapter we did show how to enforce termination in the dynamically scoped case. Unfortunately it is not possible to carry the same result in a straightforward way to the lexically scoped case. This would amount saying that every partial recursive function would terminate. Termination occurs in the dynamic case because we can prove for any program only a finite portion of its semantic domain is actually used. The proof is based on the existence of a one–to–one correspondence between the functions that are used in the program execution and the abstractions actually occurring in the program text.

Such one–to–one correspondence doesn't exist in the lexical case because the same abstraction may denote many functions depending on the environment where the free variables are bound. If in the program execution the same abstraction is evaluated infinitely many times with infinitely many different bindings for its free variables, it may return infinitely many different values. The case will occur with a function f that recursively calls $f(x+1)$ when called with a Church numeral x as an argument. This involves the Church successor function $\lambda x \lambda y \lambda z . y(xyz)$.

The successor $x + 1$ is an abstraction $\lambda y \lambda z.y(xyz)$ that has infinitely many values depending on the Church numeral value of the free variable x. We cannot enforce termination in the lexical case with the method used in the dynamic case.

5.1 The Normal Semantics of Lambda-Calculus

We should restate at this point we are looking for a semantics suitable for data flow analysis. For that purpose we do not need an exact representation of lambda-calculus. It is sufficient to write a terminating approximation of its semantics. However, the usual semantics does not immediately support an obvious approximation. The one we have in mind uses side-effect like constructs. We will have to bring a representation of that semantics that is close to the functional definition of the run-time system of an actual compiler and build the approximation from it. This forces us to write this chapter as a series of semantics. Each semantics in the series is a representation of the previous one but is closer to a compiler run-time system. Eventually we will be ready to write the terminating approximation.

Semantics 5.1 The Normal Semantics of Lambda-Calculus

Domains

Var : *The domains of variables*

Ex : *The domain of expressions*

$\mathbf{E} = \mathbf{Var} \longrightarrow \mathbf{V}$

the domain of environments

$$\mathbf{V} = \mathbf{V} \longrightarrow \mathbf{V}$$

the domain of values

Functionality

$$I\colon \mathbf{Ex} \longrightarrow \mathbf{E} \longrightarrow \mathbf{V}$$

Semantics

Exp $\longrightarrow$ `var`

$$I[\![\mathrm{x}]\!](e) = e[\![\mathrm{x}]\!]$$

Exp $\longrightarrow$ `λvar.Exp1`

$$I[\![\lambda\ \mathtt{x.e1}]\!](e) = \lambda v. I[\![\mathtt{e1}]\!](e[\mathtt{x} := v])$$

Exp $\longrightarrow$ `Exp1(Exp2)`

$$I[\![\mathtt{e1(e2)}]\!](e) = I[\![\mathtt{e1}]\!](e)(I[\![\mathtt{e2}]\!](e))$$

End of semantics 5.1. ▮

The meaning of this semantics is obvious, especially if the reader has looked at our explanation of the semantics of the corresponding LISP constructs in chapter 3.

5.2 Closures

Our first transformation will be the introduction of closures. In an actual run-time system, a functional value is represented by a pair, called a closure, consisting of a two pointers, one to an environment and one to the entry point of a routine. This gives the domain equation $\mathbf{Cl} = \mathbf{E} \times (\mathbf{E} \longrightarrow \mathbf{V})$. When the time comes to call a c'osure $f \in \mathbf{Cl}$, we first reconstitute the corresponding functional value in $\mathbf{V}$, passing the environment to the code with the help of function $Bind(f) = snd(f)(fst(f))$. This yields the following semantics.

Semantics 5.2 A Semantics with Explicit Closures

Domains

Var : *The domains of variables*

Ex : *The domain of expressions*

$\mathbf{E} = \mathbf{Var} \longrightarrow \mathbf{Cl}$

the domain of environments

$\mathbf{V} = \mathbf{Cl} \longrightarrow \mathbf{Cl}$

the domain of values

$\mathbf{Cl} = \mathbf{E} \times (\mathbf{E} \longrightarrow \mathbf{V})$

the domain of closures

Functionality

$Bind: \mathbf{Cl} \longrightarrow \mathbf{V}$

$Bind(f) = snd(f)(fst(f))$

$I: \mathbf{Ex} \longrightarrow \mathbf{E} \longrightarrow \mathbf{Cl}$

Semantics

Exp $\longrightarrow$ **var**

$I[\![\mathtt{x}]\!](e) = e[\![\mathtt{x}]\!]$

Exp $\longrightarrow$ λ**var.Exp1**

$I[\![\lambda\ \mathtt{x.e1}]\!](e) = \langle e, \lambda e\lambda v.I[\![\mathtt{e1}]\!](e[\mathtt{x}\ := \ v])\rangle$

Exp $\longrightarrow$ **Exp1(Exp2)**

$I[\![\mathtt{e1(e2)}]\!](e) = Bind(I[\![\mathtt{e1}]\!](e))(I[\![\mathtt{e2}]\!](e))$

End of semantics 5.2. ▮

Let's convince ourself that 5.2 is indeed a representation of 5.1. Here we compare pair wise the clauses for each syntactic constructs.

5.1— $I[\![\mathtt{x}]\!](e) = e[\![\mathtt{x}]\!]$

5.2— $I[\![\mathtt{x}]\!](e) = e[\![\mathtt{x}]\!]$

5.1— $I[\![\lambda \mathtt{x.e1}]\!](e) = \quad \lambda v.I[\![\mathtt{e1}]\!](e[\mathtt{x}\ := \ v])$

5.2— $I[\![\lambda\ \mathtt{x.e1}]\!](e) = \langle e, \lambda e\lambda v.I[\![\mathtt{e1}]\!](e[\mathtt{x}\ := \ v])\rangle$

5.1 – $I[\![\mathtt{e1(e2)}]\!](e) = \qquad I[\![\mathtt{e1}]\!](e)\ \ (I[\![\mathtt{e2}]\!](e))$

5.2— $I[\![\mathtt{e1(e2)}]\!](e) = Bind(I[\![\mathtt{e1}]\!](e))(I[\![\mathtt{e2}]\!](e))$

The key difference is in the rule for abstraction where semantics 5.2 abstracts out the environment e and adjoins it as separate pair component. This difference in representation is compensated in the rule for application where the *Bind* function replaces things as they should work.

5.3 Call Frames

Let's introduce undelying ides of the next semantics. In an actual compiler, when a function is called, we create a small data structure called a "call frame" (or "activation record") that maps the formal parameters to the actual values they take within that particular call. The domain $\mathbf{F}$ of call frames is given by the equation $\mathbf{F} = \mathbf{Var} \longrightarrow (\mathbf{Cl} + \{not\_found\})$. A variable is mapped to its value or it is not found in the frame. The list of all relevant call frames is the representation of some environment. This gives the equation $\mathbf{E} = nil + (\mathbf{F} \times \mathbf{E})$. The value of some variable is found by scanning the list of frames until the proper frame is found. This is done with the function *Apply* defined as:

$$Apply\colon \mathbf{E} \longrightarrow \mathbf{Var} \longrightarrow \mathbf{Cl}$$

$$Apply(nil)(x) = \bot_{\mathbf{Cl}}$$

$$Apply(\langle e, r\rangle)(x) = \textbf{ if } e(x) \neq not\_found \textbf{ then } e(x) \textbf{ else } Apply(r)(x)$$

Note how $Apply(f)$ maps an environment in list form to the same environment in functional form. This is expressed by the type of $Apply\colon \mathbf{E} \longrightarrow \mathbf{Var} \longrightarrow \mathbf{Cl}$. The creation of a new environment with $e[\mathrm{x} := v]$ is redefined as:

$$e[\mathrm{x} := v] = \langle(\lambda u.\ \textbf{if } u = x \textbf{ then } v \textbf{ else } not\_found), e\rangle$$

In other words, we simply build a frame for $[x := v]$ and push it on top of e. We are now ready to write semantics 5.3.

Semantics 5.3 A Semantics with Explicit Call Frames

Domains

Var : *The domains of variables*

Ex : *The domain of expressions*

$\mathbf{F} = \mathbf{Var} \longrightarrow (\mathbf{Cl} + \{not\_found\})$

the domain of frames

$\mathbf{E} = \{nil\} + (\mathbf{F} \times \mathbf{E})$

the domain of environments

$\mathbf{V} = \mathbf{Cl} \longrightarrow \mathbf{Cl}$

the domain of values

$\mathbf{Cl} = \mathbf{E} \times (\mathbf{E} \longrightarrow \mathbf{V})$

the domain of closures

Functionality

Bind: $\mathbf{Cl} \longrightarrow \mathbf{V}$

$Bind(f) = snd(f)(fst(f))$

Apply: $\mathbf{E} \longrightarrow \mathbf{Var} \longrightarrow \mathbf{Cl}$

$Apply(nil)(x) = \perp_{\mathbf{Cl}}$

$Apply(\langle e, r\rangle)(x) =$ **if** $e(x) \neq not\_found$ **then** $e(x)$ **else** $Apply(r)(x)$

$e[\mathtt{x} := v] = \langle(\lambda u.\ \textbf{if}\ u = x\ \textbf{then}\ v\ \textbf{else}\ \mathit{not\_found}), e\rangle$

$I\colon \mathbf{Ex} \longrightarrow \mathbf{E} \longrightarrow \mathbf{Cl}$

Semantics

$\mathbf{Exp} \quad \longrightarrow \quad \mathtt{var}$

$I[\![\mathtt{x}]\!](e) = \mathit{Apply}(e)[\![\mathtt{x}]\!]$

$\mathbf{Exp} \quad \longrightarrow \quad \lambda\mathtt{var}.\mathbf{Exp1}$

$I[\![\lambda\mathtt{x}.\mathtt{e1}]\!](e) = \langle e, \lambda e\lambda v.I[\![\mathtt{e1}]\!](e[\mathtt{x} := v])\rangle$

$\mathbf{Exp} \quad \longrightarrow \quad \mathbf{Exp1(Exp2)}$

$I[\![\mathtt{e1(e2)}]\!](e) = \mathit{Bind}(I[\![\mathtt{e1}]\!](e))(I[\![\mathtt{e2}]\!](e))$

End of semantics 5.3. ∎

The key to this semantics is the definition of *Apply* and $e[\mathtt{x} := v]$. If it is clear to the reader that $\mathit{Apply}(e)[\![\mathtt{x}]\!]$ indeed retrieves the value of x in the environment e and that the equality $\mathit{Apply}(e[\mathtt{x} := v]) = \lambda u.\ \textbf{if}\ u = \mathtt{x}\ \textbf{then}\ v\ \textbf{else}\ \mathit{Apply}(e)(u)$ holds, then the correctness of the semantics follow from a pair wise comparison of the corresponding semantic rules.

5.2— $I[\![\mathtt{x}]\!](e) = \qquad e[\![\mathtt{x}]\!]$

5.3— $I[\![\mathtt{x}]\!](e) = \mathit{Apply}(e)[\![\mathtt{x}]\!]$

5.2— $I[\![\lambda\ \mathtt{x}.\mathtt{e1}]\!](e) = \langle e, \lambda e\lambda v.I[\![\mathtt{e1}]\!](e[\mathtt{x} := v])\rangle$

5.3— $I[\![\lambda\ \mathtt{x}.\mathtt{e1}]\!](e) = \langle e, \lambda e\lambda v.I[\![\mathtt{e1}]\!](e[\mathtt{x} := v])\rangle$

5.2— $I[\![\mathbf{e1(e2)}]\!](e) = \mathit{Bind}(I[\![\mathbf{e1}]\!](e))(I[\![\mathbf{e2}]\!](e))$

5.3— $I[\![\mathbf{e1(e2)}]\!](e) = \mathit{Bind}(I[\![\mathbf{e1}]\!](e))(I[\![\mathbf{e2}]\!](e))$

5.4 Lexical Scopes and Return Addresses

The semantics is now much closer to the form required to support a terminating approximation. We are still lacking some information that is usually associated with call frames. One useful piece of information is the lexical scope where the variables are defined. This is in one–to–one correspondence with the abstraction expression that is called. The other piece of information is the return address, information that is in one–to–one correspondence with the application expression where the call occurred. A frame labeled with this information is defined by the domain equation $\mathbf{F} = (\mathbf{Var} \longrightarrow (\mathbf{Cl} + \{\mathit{not\_found}\})) \times (\mathbf{Ex} \times \mathbf{Ex})$. The first component of domain **F** is a frame as was defined in 5.3. The second component is a pair $(\mathbf{Ex} \times \mathbf{Ex})$ where the first expression is an abstraction expression representing the lexical scope and the second expression is an application representing the return address. A complete environment is a list of frames as in 5.3. Of course, function *Apply* has to be redefined to ignore the second component of each frame:

$$\mathbf{E} = \{\mathit{nil}\} + (\mathbf{F} \times \mathbf{E})$$

$$\mathit{Apply}\colon \mathbf{E} \longrightarrow \mathbf{Var} \longrightarrow \mathbf{Cl}$$

$$\mathit{Apply}(\mathit{nil})(x) = \bot_{\mathbf{Cl}}$$

$$\mathit{Apply}(\langle\langle e, l\rangle, r\rangle)(x) = \textbf{if } e(x) \neq \mathit{not\_found} \textbf{ then } e(x)$$

else $Apply(r)(x)$

Similarly, new parameters $\langle a, b\rangle$ have to be introduced in $e[\mathrm{x} := v]$ to handle the extra information:

$$e[\mathrm{x} := v]/\langle a, b\rangle = \langle\langle f, \langle a, b\rangle\rangle, e\rangle$$

Where $f = \lambda u.$ **if** $u = x$ **then** v **else** not_found

Another change occurs to the semantics as well. The lexical scope is available when the abstraction is evaluated but the return address is not until an actual call is made. This changes the closure into a triple $\mathbf{Cl} = (\mathbf{E}\times(\mathbf{Ex}\times(\mathbf{Ex} \longrightarrow \mathbf{Ex} \longrightarrow \mathbf{E} \longrightarrow V)))$ that contains an environment, a lexical scope and a function that accepts a label for a call frame and an environment to return a value. The $Bind$ function is affected as follows:

$$scnd(x) = fst(snd(x))$$

$$trd(x) = snd(snd(x))$$

$$\langle x, y. z\rangle = \langle x. \langle y. z. \rangle\rangle$$

$$Bind\colon \mathbf{Ex} \longrightarrow \mathbf{Cl} \longrightarrow V$$

$$Bind(ex)(x) = (\ trd(x)\ )(\ scnd(x)\ )[\![(]\!] ex)(\ fst(x);\ )$$

The function identified as $trd(x)$ expects both expressions required to compose a label and an environment to return a value in $\mathbf{V}$. $Bind(ex)(x)$ simply extracts all the required information and obtains the value in $\mathbf{V}$ from the closure as should be expected. This yields the semantics:

Semantics 5.4 A Semantics with Labeled Call Frames

Domains

Var : *The domains of variables*

Ex : *The domain of expressions*

$\mathbf{RetAdd} = \mathbf{Ex}$

The domain of return addresses

$\mathbf{LexScope} = \mathbf{Ex}$

The domain of lexical scopes

$\mathbf{F} = (\mathbf{Var} \longrightarrow (\mathbf{Cl} + \{\mathit{not\_found}\})) \times (\mathbf{LexScope} \times \mathbf{RetAdd})$

the domain of frames

$\mathbf{E} = \{\mathit{nil}\} + (\mathbf{F} \times \mathbf{E})$

the domain of environments

$\mathbf{V} = \mathbf{Cl} \longrightarrow \mathbf{Cl}$

the domain of values

$\mathbf{Cl} = (\mathbf{E} \times (\mathbf{Ex} \times (\mathbf{LexScope} \longrightarrow \mathbf{RetAdd} \longrightarrow \mathbf{E} \longrightarrow \mathbf{V})))$

the domain of closures

Functionality

$scnd(x) = fst(snd(x))$

$trd(x) = snd(snd(x))$

$\langle x, y, z\rangle = \langle x, \langle y, z\rangle\rangle$

$Bind\colon \mathbf{Ex} \longrightarrow \mathbf{Cl} \longrightarrow \mathbf{V}$

$$Bind(ex)(x) = (\ trd(x)\ )(\ scnd(x)\ )(ex)(\ fst(x)\ )$$

$Apply\colon \mathbf{E} \longrightarrow \mathbf{Var} \longrightarrow \mathbf{Cl}$

$$Apply(nil)(x) = \bot_{\mathbf{Cl}}$$

$$Apply(\langle\langle e, l\rangle, r\rangle)(x) = \textbf{ if } e(x) \neq not\_found \textbf{ then } e(x)$$

$$\textbf{else } Apply(r)(x)$$

$e[\mathtt{x} := v]/\langle a, b\rangle = \langle\langle f, \langle a, b\rangle\rangle, e\rangle$

Where $f = \lambda u.\ \textbf{if } u = x \textbf{ then } v \textbf{ else } not\_found$

$I\colon \mathbf{Ex} \longrightarrow \mathbf{E} \longrightarrow \mathbf{Cl}$

Semantics

Exp $\longrightarrow$ **var**

$$I[\![\mathtt{x}]\!](e) = Apply(e)[\![\mathtt{x}]\!]$$

Exp $\longrightarrow$ **λvar.Exp1**

$$I[\![\lambda\mathtt{x}.\mathtt{e1}]\!](e) = \langle e, [\![\lambda\mathtt{x}.\mathtt{e1}]\!], \lambda a\lambda b\lambda e\lambda v. I[\![\mathtt{e1}]\!](e[\mathtt{x} := v]/\langle a, b\rangle)\rangle$$

Exp $\longrightarrow$ **Exp1(Exp2)**

$$I[\![\mathtt{e1(e2)}]\!](e) = \mathit{Bind}[\![\mathtt{e2}]\!](I[\![\mathtt{e1}]\!](e))\,(I[\![\mathtt{e2}]\!](e))$$

End of semantics 5.4. ▮

It is not at all trivial that the addition of irrelevant information doesn't affect the correctness of the semantics. This operation has a drastic effect on the type of the values involved. It may also affect termination of the program when the computation of the additional information enters an infinite loop. The main difference between 5.3 and 5.4 is the fact we have to gather additional information to handle environments properly. This is best expressed in a pair wise comparison of the equations.

5.3— $I[\![\mathtt{x}]\!](e) = \mathit{Apply}(e)[\![\mathtt{x}]\!]$

5.4— $I[\![\mathtt{x}]\!](e) = \mathit{Apply}(e)[\![\mathtt{x}]\!]$

5.3— $I[\![\lambda\mathtt{x.e1}]\!](e) = \langle e, \qquad \lambda e\lambda v.I[\![\mathtt{e1}]\!](e[\mathtt{x} := v])\rangle$

5.4— $I[\![\lambda\mathtt{x.e1}]\!](e) = \langle e, [\![\lambda\mathtt{x.e1}]\!], \lambda a\lambda b\lambda e\lambda v.I[\![\mathtt{e1}]\!](e[\mathtt{x} := v]/\langle a, b\rangle)\rangle$

5.3— $I[\![\mathtt{e1(e2)}]\!](e) = \mathit{Bind} \qquad (I[\![\mathtt{e1}]\!](e))\,(I[\![\mathtt{e2}]\!](e))$

5.4— $I[\![\mathtt{e1(e2)}]\!](e) = \mathit{Bind}[\![\mathtt{e2}]\!](I[\![\mathtt{e1}]\!](e))\,(I[\![\mathtt{e2}]\!](e))$

5.3— $\mathit{Bind} \qquad (x) = \qquad \mathit{snd}(x)(\mathit{fst}(x))$

5.4— $\mathit{Bind}(ex)(x) = (\ \mathit{trd}(x)\ )(\ \mathit{scnd}(x)\ )(ex)(\ \mathit{fst}(x)\ )$

Look at the rule for abstraction. The last component of the closure requires two more parameters in 5.4 than in 5.3. The parameter a is the lexical scope and is enclosed as the second component of the closure. The parameter b is the call return address and is supplied to the *Bind* function when the application is made. *Bind* had to be redefined to handle the extra information in order to preserve the meaning. We hope at this point the reader will agree that our representation is correct. We shall now continue with the next transformation.

5.5 Environments Viewed as Trees

We should observe that within an actual compiler, call frames form a tree rather than a list structure. This is exemplified in the diagram below.

$$\begin{tikzcd}
 & \langle f1, ([\![\lambda \mathtt{x}.\mathtt{e1}]\!], [\![\mathtt{e2}(\mathtt{e3})]\!])\rangle & \\
\langle f2, ([\![\lambda \mathtt{y}.\mathtt{e4}]\!], [\![\mathtt{e5}(\mathtt{e6})]\!])\rangle \arrow[ur] & & \langle f3, ([\![\lambda \mathtt{z}.\mathtt{e7}]\!], [\![\mathtt{e8}(\mathtt{e9})]\!])\rangle \arrow[ul]
\end{tikzcd}$$

The frame $f1$ results from a call of the value of the abstraction $\lambda\mathtt{x}.\mathtt{e1}$ resulting from evaluation of the application $\mathtt{e2}(\mathtt{e3})$. Within the scope of $\lambda\mathtt{x}.\mathtt{e1}$ there are two other abstractions that are instantiated: $\lambda\mathtt{y}.\mathtt{e4}$ and $\lambda\mathtt{z}.\mathtt{e7}$. These abstractions are called from the applications $\mathtt{e5}(\mathtt{e6})$ and $\mathtt{e8}(\mathtt{e9})$ respectively. This yields to the creation of the corresponding frames $f2$ and $f3$. The resulting environments share the frame $f1$, causing the occurrence of a tree structure where the act of pushing a new frame on the environment corresponds to the act of spawning a new child node at the bottom of the tree. In this example, $f2$ and $f3$ are spawned from $f1$. Any environment occurring in the evaluation of an expression can be viewed as a branch of the tree starting from the root and including all son nodes down to some descendent that was the last frame included in the environment.

We plan to use this organization of frames into a treelike structure to formulate the approximation that will ensure termination of the semantics. Within a branch, a frame can be viewed as a function of its path name consisting of the list of all labels occurring from the root node down to the frame. For example, the path name for $f1$ is $\langle \lambda \mathtt{x}.\mathtt{e1}, \mathtt{e2(e3)} \rangle$ and the path name from $f2$ is the list $\langle \lambda \mathtt{x}.\mathtt{e1}, \mathtt{e2(e3)} \rangle$, $\langle \lambda \mathtt{y}.\mathtt{e4}, \mathtt{e5(e6)} \rangle$. From a given environment this path name is guaranteed unique for every frame because the lengths of the paths to each frame are all different. This allows a last transformation of the exact semantics before we formulate our approximate semantics. We want to make the environment appear as a function from path names to frames. The domain equations are:

$\mathbf{L} = (\mathbf{Ex} \times \mathbf{Ex})$

L is the domain of labels for frames

$\mathbf{P} = \{nil\} + (\mathbf{L} \times \mathbf{P})$

P is the domain of path names

$\mathbf{F} = \mathbf{Var} \longrightarrow (\mathbf{Cl} + \{not\_found\})$

F is the domain of frames

$\mathbf{E} = (\mathbf{P} \longrightarrow \mathbf{F}) \times \mathbf{P}$

E is the domain of environments

Now an environment is considered as a pair $\langle e, p\rangle$ where e is a function mapping path names to frames and p is the path name of the topmost frame in the environment. The path names to other frames in the same environment are identical to the prefixes of p. This allows to define *Apply* as:

$Apply$: $\mathbf{E} \longrightarrow \mathbf{Var} \longrightarrow \mathbf{Cl}$

$$Apply(\langle e, nil\rangle)(x) = \bot_{\mathbf{Cl}}$$

$$\begin{aligned} Apply(\langle e, \langle l, r\rangle\rangle)(x) = \ & \textbf{if } e(\langle l, r\rangle)(x) \neq not\_found \\ & \textbf{then } e(\langle l, r\rangle)(x) \\ & \textbf{else } Apply(\langle e, r\rangle)(x) \end{aligned}$$

The update of an environment is defined as:

$e[\mathrm{x} := v]/\langle a, b\rangle = \langle fst(e)[\mathrm{p} := f], p\rangle$

Where $f = \lambda u.$ **if** $u = x$ **then** v **else** not_found

and $p = \langle\langle a, b\rangle, snd(e)\rangle$

This means we find the frame f that assigns v to x, we compute the path p to the frame f and we make f the topmost frame of the environment. We can now define the semantics as:

Semantics 5.5 A Semantics with Environments as Function of Path Names

Domains

Var : *The domains of variables*

Ex : *The domain of expressions*

$\mathbf{RetAdd} = \mathbf{Ex}$

The domain of return addresses

$\mathbf{LexScope} = \mathbf{Ex}$

The domain of lexical scopes

$\mathbf{F} = (\mathbf{Var} \longrightarrow (\mathbf{Cl} + \{not\_found\}))$

the domain of frames

$\mathbf{L} = (\mathbf{LexScope} \times \mathbf{RetAdd})$

the domain of labels frames

$\mathbf{P} = \{nil\} + (\mathbf{L} \times \mathbf{P})$

the domain of path names

$\mathbf{E} = (\mathbf{P} \longrightarrow \mathbf{F}) \times \mathbf{P}$

the domain of environments

$\mathbf{V} = \mathbf{Cl} \longrightarrow \mathbf{Cl}$

the domain of values

$\mathbf{Cl} = (\mathbf{E} \times (\mathbf{Ex} \times (\mathbf{LexScope} \longrightarrow \mathbf{RetAdd} \longrightarrow \mathbf{E} \longrightarrow \mathbf{V})))$

the domain of closures

Functionality

$scnd(x) = fst(snd(x))$

$trd(x) = snd(snd(x))$

$\langle x, y, z\rangle = \langle x, \langle y, z\rangle\rangle$

$Bind$: $\mathbf{Ex} \longrightarrow \mathbf{Cl} \longrightarrow \mathbf{V}$

$Bind(ex)(x) = (\ trd(x)\ )(\ scnd(x)\ )(ex)(\ fst(x)\ )$

$Apply$: $\mathbf{E} \longrightarrow \mathbf{Var} \longrightarrow \mathbf{Cl}$

$Apply(\langle e, nil\rangle)(x) = \bot_{\mathrm{Cl}}$

$$
\begin{aligned}
Apply(\langle e, \langle l, r\rangle\rangle)(x) = \ & \textbf{if } e(\langle l, r\rangle)(x) \neq not\_found \\
& \textbf{then } e(\langle l, r\rangle)(x) \\
& \textbf{else } Apply(\langle e, r\rangle)(x)
\end{aligned}
$$

$e[\mathrm{x} := v]/\langle a, b\rangle = \langle fst(e)[\mathrm{p} := f], p\rangle$

Where $f = \lambda u.$ **if** $u = x$ **then** v **else** not_found

and $p = \langle\langle a, b\rangle, snd(e)\rangle$

I: $\mathbf{Ex} \longrightarrow \mathbf{E} \longrightarrow \mathbf{Cl}$

Semantics

Exp $\longrightarrow$ **var**

$$I[\![\mathtt{x}]\!](e) = Apply[\![\mathtt{e}]\!](x)$$

$$\mathbf{Exp} \longrightarrow \lambda\mathtt{var}.\mathbf{Exp1}$$

$$I[\![\lambda\mathtt{x}.\,\mathtt{e1}]\!](e) = \langle e, [\![\lambda\mathtt{x}.\,\mathtt{e1}]\!], \lambda a\lambda b\lambda e\lambda v.I[\![\mathtt{e1}]\!](e[\mathtt{x} \ := \ v]/\langle a, b\rangle)\rangle$$

$$\mathbf{Exp} \longrightarrow \mathbf{Exp1}(\mathbf{Exp2})$$

$$I[\![\mathtt{e1}(\mathtt{e2})]\!](e) = Bind[\![\mathtt{e2}]\!](I[\![\mathtt{e1}]\!](e))(I[\![\mathtt{e2}]\!](e))$$

End of semantics 5.5. ▮

The reader should observe the semantic equations in 5.5 are in exact syntactic identity with the corresponding equations in 5.4. The only differences lie in the domain equations and in the definition of Apply and environment updates as we had discussed them before the introduction of 5.5.

5.6 Approximate Trees

We had already shown that a call frame is uniquely identified by its path name within a given environment. This is not true across environments. For example, recursion may cause the same abstraction to be called several times causing the occurrence of several frames sharing the same path names as in the diagram below.

$$\begin{tikzcd}
 & \langle f1, \langle [\![\lambda\ \mathtt{x}.\mathtt{e1}]\!], [\![\mathtt{e2}(\mathtt{e3})]\!]\rangle\rangle & \\
\langle f2, \langle [\![\lambda\ \mathtt{y}.\mathtt{e4}]\!], [\![\mathtt{e5}(\mathtt{e6})]\!]\rangle\rangle \arrow[ur] & & \langle f3, \langle [\![\lambda\ \mathtt{y}.\mathtt{e4}]\!], [\![\mathtt{e5}(\mathtt{e6})]\!]\rangle\rangle \arrow[ul]
\end{tikzcd}$$

The function $\lambda y . e4$ is called recursively from $\lambda x . e1$. Two frames $f2$ and $f3$ are generated from two distinct recursive calls. They share the same path name in the tree of frames despite the fact they occur in distinct environments. This suggests the modifications to 5.5 we will introduce in semantics 5.6. The first approximation we have in mind is to join all those frames sharing the same path name into a single one. Now, at any moment, at most one frame will exist for every path name in the compiler run-time system. Therefore, parameter passing is no longer the assignment of a value to a variable into the frame but the addition of the new value to the sum (join) of all values this parameter has taken in previous calls of the same function. This is so because we do not want a frame to lose information about one environment because that same frame is reused for a new call.

The handling of environments becomes more complex than the domain equation $\mathbf{E} = (\mathbf{P} \longrightarrow \mathbf{F}) \times \mathbf{P}$ suggests. The component of type $\mathbf{G} = (\mathbf{P} \longrightarrow \mathbf{F})$ now represents the tree like data structure that contains all environments in the run-time system. Every call may modify that treelike structure because actual parameters are added to existing information rather than assigned to a variable. The type of the semantic function I is changed from $\mathbf{Ex} \longrightarrow \mathbf{E} \longrightarrow \mathbf{Cl}$ to $\mathbf{Ex} \longrightarrow \mathbf{E} \longrightarrow (\mathbf{Cl} \times \mathbf{G})$ to provide some permanence to the impact of parameter passing on frames. The evaluation of every expression will return not only a closure in $\mathbf{Cl}$ but also a modified tree of frames that will be part of the environment supplied to the evaluation of the next expression. Similarly, the domain $\mathbf{V} = \mathbf{Cl} \longrightarrow \mathbf{Cl}$ is modified to

$\mathbf{V} = (\mathbf{Cl} \times \mathbf{G}) \longrightarrow (\mathbf{Cl} \times \mathbf{G})$ to reflect the changes occurring to the tree of frames when a function is called. This introduces the notion of a global tree of frames that is side-affected as the evaluation progress. The value of an expression depends not only of the syntax of the expression but also on the current state of that tree. This is the meaning of the pair $\mathbf{Cl} \times \mathbf{G}$ that occurs in many domains. The **Cl** component stands for the abstraction that is the value of the expression and the **G** component stands for the current state of the tree.

Given some trees, environments are uniquely specified by their path name, so we can replace the environment component of closures by the corresponding path name, changing the type of closures from $(\mathbf{E} \times (\mathbf{Ex} \times (\mathbf{LexScope} \longrightarrow \mathbf{RetAdd} \longrightarrow \mathbf{E} \longrightarrow \mathbf{V})))$ to the corresponding type $(\mathbf{P} \times (\mathbf{Ex} \times (\mathbf{LexScope} \longrightarrow \mathbf{RetAdd} \longrightarrow \mathbf{E} \longrightarrow \mathbf{V})))$. The path name component can in turn be transformed into an environment if it is paired with the current tree of frame when the closure is bound into a function to be called. This is consistent with the already discussed notion of tree of frames as a global data structure that is side-effected as the program execution progresses. This reflects exactly how a compiler run-time system should work.

We are now ready to write the new semantics.

Semantics 5.6 A Semantics with Environments Uniquely Determined by Path Names

Domains

Var : *The domains of variables*

Ex : *The domain of expressions*

$\mathbf{F} = (\mathbf{Var} \longrightarrow (\mathbf{Cl} + \{not\_found\}))$

the domain of frames

$\mathbf{L} = (\mathbf{Ex} \times \mathbf{Ex})$

the domain of labels for frames

$\mathbf{P} = \{nil\} + (\mathbf{L} \times \mathbf{P})$

the domain of path names

$\mathbf{G} = \mathbf{P} \longrightarrow \mathbf{F}$

the domain of trees of frames

$\mathbf{E} = \mathbf{G} \times \mathbf{P}$

the domain of environments

$\mathbf{V} = (\mathbf{Cl} \times \mathbf{G}) \longrightarrow (\mathbf{Cl} \times \mathbf{G})$

the domain of values

$\mathbf{Cl} = (\mathbf{P} \times (\mathbf{Ex} \times (\mathbf{LexScope} \longrightarrow \mathbf{RetAdd} \longrightarrow \mathbf{E} \longrightarrow \mathbf{V})))$

the domain of closures

Functionality

$scnd(x) = fst(snd(x))$

$trd(x) = snd(snd(x))$

$\langle x, y, z\rangle = \langle x, y\rangle z$

$Bind$: $\mathbf{Ex} \longrightarrow (\mathbf{Cl} \times \mathbf{G}) \longrightarrow \mathbf{V}$

$$Bind(ex)(x, g) = (\ trd(x)\ )(\ scnd(x)\ )(ex)(\ \langle g, fst(x)\rangle\ )$$

$Apply$: $\mathbf{E} \longrightarrow \mathbf{Var} \longrightarrow (\mathbf{Cl} \times \mathbf{G})$

$$Apply(\langle e, nil\rangle)(x) = \langle \perp, e\rangle$$

$$\begin{aligned} Apply(\langle e, \langle l, r\rangle\rangle)(x) = \ & \textbf{if } e(< l, r >)(x) \neq not\_found \\ & \textbf{then } \langle e(\langle l, r\rangle)(x), e\rangle \\ & \textbf{else } Apply(\langle e, r\rangle)(x) \end{aligned}$$

$e[\mathrm{x} := v]/\langle a, b\rangle = \langle fst(e)[\mathrm{p} := f \sqcup fst(e)(p)], p\rangle$

Where $f = \lambda u.$ **if** $u = x$ **then** $fst(v)$ **else** not_found

and $p = \langle\langle a, b\rangle, snd(e)\rangle$

I: $\mathbf{Ex} \longrightarrow \mathbf{E} \longrightarrow (\mathbf{Cl} \times \mathbf{G})$

Semantics

Exp $\longrightarrow$ **var**

$I[\![\mathbf{x}]\!](\langle g, p\rangle) = Apply(\langle g, p\rangle)[\![\mathbf{x}]\!]$

Exp $\longrightarrow$ **λvar.Exp1**

$I[\![\lambda\mathbf{x}.\mathbf{e1}]\!](\langle g, p\rangle) = \langle u, g\rangle$

$$\textit{Where } u = \langle p, [\![\lambda x.\, e1]\!], \lambda a \lambda b \lambda e \lambda v. I[\![e1]\!](e[x := v]/\langle a, b\rangle)\rangle$$

Exp $\longrightarrow$ **Exp1(Exp2)**

$$I[\![e1(e2)]\!](\langle g, p\rangle) = Bind[\![e2]\!](u)(I[\![e2]\!](\langle snd(u), p\rangle))$$

$$\textit{Where } u = I[\![e1]\!](\langle g, p\rangle)$$

End of semantics 5.6. ▮

We will compare the semantic equations of 5.5 and 5.6 pair wise, in order to illustrate how these semantics relate to each other.

5.5— $I[\![x]\!](e) = Apply[\![e]\!](x)$

5.6— $I[\![x]\!](\langle g, p\rangle) = Apply(\langle g, p\rangle)[\![x]\!]$

In both semantics, environments have the form $\langle g, p\rangle$ where g is a tree of frames and p is a path name. Except for the different definitions of *Apply*, both definitions are identical. So let's look now at the definitions of *Apply*.

5.5— *Apply*: $\mathbf{E} \longrightarrow \mathbf{Var} \longrightarrow \mathbf{Cl}$

$$Apply(\langle e, nil\rangle)(x) = \bot_{\mathbf{Cl}}$$

$$\begin{aligned} Apply(\langle e, \langle l, r\rangle\rangle)(x) = \; & \textbf{if } e(\langle l, r\rangle)(x) \neq not\_found \\ & \textbf{then } e(<l, r>)(x) \\ & \textbf{else } Apply(\langle e, r\rangle)(x) \end{aligned}$$

5.6— *Apply*: $\mathbf{E} \longrightarrow \mathbf{Var} \longrightarrow (\mathbf{Cl} \times \mathbf{G})$

$$Apply(\langle e, nil\rangle)(x) = \langle \bot, e\rangle$$

$$\begin{aligned} Apply(\langle e, \langle l, r\rangle\rangle)(x) = \ & \textbf{if } e(< l, r >)(x) \neq not\_found \\ & \textbf{then } \langle e(\langle l, r\rangle)(x), e\rangle \\ & \textbf{else } Apply(\langle e, r\rangle)(x) \end{aligned}$$

The only difference is in the type of the returned value. In 5.6 we append to the returned closure the current state of the tree of frames. The rest of the definition is unaffected.

5.5— $I[\![\lambda x.\, e1]\!](e) = \langle e, [\![\lambda x.\, e1]\!], \lambda a \lambda b \lambda e \lambda v. I[\![e1]\!](e[x := v]/\langle a, b\rangle)\rangle$

5.6— $I[\![\lambda x.\, e1]\!](\langle g, p\rangle) = \langle u, g\rangle$

Where $u = \langle p, [\![\lambda x.\, e1]\!], \lambda a \lambda b \lambda e \lambda v. I[\![e1]\!](e[x := v]/\langle a, b\rangle)\rangle$

In 5.6, u denotes a closure that corresponds exactly to the closure created in 5.5 except that only the path name of the environment is included. This closure u is paired with the tree of frames g.

5.5— $I[\![e1(e2)]\!](e) = Bind[\![e2]\!](I[\![e1]\!](e))(I[\![e2]\!](e))$

5.6— $I[\![e1(e2)]\!](\langle g, p\rangle) = Bind[\![e2]\!](I[\![e1]\!](\langle g, p\rangle))(I[\![e2]\!](\langle snd(u), p\rangle))$

Again semantics 5.5 is almost identical to 5.6 except we have to pipeline the side effects that may occur to the tree of frames across successive calls to I. The function *Bind* has been defined to allow such pipelining to the occurrence of I that is involved in the abstraction that is the value of **e1**. We refer to the semantics of *Bind* for details.

5.5— $\mathbf{Cl} = (\mathbf{E} \times (\mathbf{Ex} \times (\mathbf{LexScope} \longrightarrow \mathbf{RetAdd} \longrightarrow \mathbf{E} \longrightarrow \mathbf{V})))$

$$Bind(ex)(x) = (\ trd(x)\ )(\ scnd(x)\ )(ex)(\ fst(x)\ )$$

5.6— $\mathbf{Cl} = (\mathbf{P} \times (\mathbf{Ex} \times (\mathbf{LexScope} \longrightarrow \mathbf{RetAdd} \longrightarrow \mathbf{E} \longrightarrow \mathbf{V})))$

$Bind\colon \mathbf{Ex} \longrightarrow (\mathbf{Cl} \times \mathbf{G}) \longrightarrow \mathbf{V}$

$$Bind(ex)(x, g) = (\ trd(x)\ )(\ scnd(x)\ )(ex)(\ \langle g, fst(x)\rangle\ )$$

Remember that in 5.6 a closure doesn't include an environment but only a path name. The closure is usually supplied with the current tree of frame that can be paired with the included path name to obtain the wanted environment. This forces the evaluation of the body of an abstraction with the current state of the tree of frames, allowing propagation of side–effects involved in parameter passing. The side effect is visible in the definition of environment update.

5.5— $e[\mathrm{x} := v]/\langle a, b\rangle = \langle fst(e)[\mathrm{p} := f], p\rangle$

Where $f = \lambda u.$ **if** $u = x$ **then** v **else** *not_found*

and $p = \langle\langle a, b\rangle, snd(e)\rangle$

5.6— $e[\mathrm{x} := v]/\langle a, b\rangle = \langle fst(e)[\mathrm{p} := f \sqcup fst(e)(p)], p\rangle$

Where $f = \lambda u.$ **if** $u = x$ **then** $fst(v)$ **else** *not_found*

and $p = \langle\langle a, b\rangle, snd(e)\rangle$

The temporary variable f represents in both semantics a newly built frame resulting from the call of a function. In 5.6 v has type $(\mathbf{Cl} \times \mathbf{G})$ and the $\mathbf{G}$ component must be dropped from the frame f with the help of the fst function. When the value of the variable will be retrieved with the *Apply* function, the current state of the tree of frames will be paired with the closure. The path name to the frame f is computed exactly in the same way in both semantics but the global tree of frames is not modified the same way. In 5.5 f supersedes any existing frame with the same path name but in 5.6 we join f with those frames, implementing the approximation we had in mind: every branch of the tree has distinct path names.

5.7 Folding Infinite Trees

We are now very close to a terminating semantics. Let's look at the type of closures.

$$\mathbf{Cl} = (\mathbf{P} \times (\mathbf{Ex} \times (\mathbf{LexScope} \longrightarrow \mathbf{RetAdd} \longrightarrow \mathbf{E} \longrightarrow \mathbf{V})))$$

The domain **Ex** is finite. We can show with an argument similar to the one used in the case of dynamic scoping that the elements of $\mathbf{LexScope} \longrightarrow \mathbf{RetAdd} \longrightarrow \mathbf{E} \longrightarrow \mathbf{V}$ are in one- to-one correspondence with the abstractions that actually occur in the program. Therefore **Cl** would be a finite type for practical purposes provided **P** is a finite type. Unfortunately this is not the case, as the equations show:

$$\mathbf{L} = (\mathbf{LexScope} \times \mathbf{RetAdd})$$

$$\mathbf{P} = \{nil\} + (\mathbf{L} \times \mathbf{P})$$

Elements of **P** are lists of labels of arbitrary long length. We can ask the question wether we can impose a bound on the length of these lists, turning the domain of closures into a practically finite domain. The answer is yes. The domain **Ex** is practically bounded in the context of a program to the expressions that actually occur in that program. The first component of a label is the abstraction that was instantiated to build a frame and the second component is the application that generated the call. If the frame gets longer than a certain bound, say the number of pairs of expressions that occur in the program, then the same expression is calling itself recursively from the same point of call because the same label will occur twice in the path name. We propose to introduce a new approximation that will identify all those frames resulting from the call of the same abstraction at the same application. This allows us to avoid path names that repeat the same label. The approximation will be defined with the help of two functions.

$Parent\colon \mathbf{L} \longrightarrow \mathbf{P} \longrightarrow \mathbf{P} \longrightarrow \mathbf{P}$

$Parent(l)(nil)(p) = p$

$Parent(l)(\langle q,r\rangle)(p) = \textbf{ if } q = l \textbf{ then } \langle q,r\rangle \textbf{ else } Parent(l)(r)(p)$

$Fold\colon \mathbf{P} \longrightarrow \mathbf{P}$

$Fold(\langle q,p\rangle) = Parent(q)(p)(\langle q,p\rangle)$

The function $Parent(l)(\langle q, r\rangle)(p)$ finds a portion of the path name $\langle q, p\rangle$ that begins with the root and ends with the label l. If no such portion exists then the value p is returned. The function *Fold* is used to avoid those path names $\langle q, p\rangle$ where the label q occurs twice. We assume that no label occurs twice in p. We look if q has a parent in p and return $\langle q, p\rangle$ if none is found. The result is the path name of a frame that has label q and whose path name complies with the requirements of the intended approximation.

These functions are indeed doing their job but they have one harmful side-effect. If path names are truncated with the help of *Fold*, they no longer represent the sequence of frames to be searched when a variable is evaluated in a given environment. The order of search no longer match the parenthood relationship in the tree of frames. This forces the introduction of a new domain S of search lists specifying the search order, and of a new function *Push* to introduce a new frame in the search list.

$$\mathbf{S} = \{nil\} + (\mathbf{P} \times \mathbf{S})$$

$$Filter: \mathbf{P} \longrightarrow \mathbf{S} \longrightarrow \mathbf{S}$$

$$Filter(p)(nil) = nil$$

$$Filter(p)(\langle q, s\rangle) = \textbf{ if } p = q \textbf{ then } Filter(p)(s) \textbf{ else } \langle q, Filter(p)(s)\rangle$$

$$Push: \mathbf{P} \longrightarrow \mathbf{S} \longrightarrow \mathbf{S}$$

$$Push(p)(s) = \langle p, Filter(p)(s)\rangle$$

Domain **S** is the domain of lists of path names to be searched when a variable is evaluated. We do not want those lists to grow infinite. The easiest way to avoid it is to not search the same frame twice. If a path name p is pushed on top of a search list, we remove it from the rest of the list. This correct because if a variable is not found in a frame when it is searched for the first time, it will not be found the second time. Search lists are finite because from a practical point of view, the domain of path names is finite. This change forces us to define environments as $\mathbf{E} = \mathbf{G} \times \mathbf{S}$ because search lists now truly represent the frames to be searched. The path name to the current frame occurs on top of the list. In closures, search lists must be included in place of path names, giving the new domain equation:

$$\mathbf{Cl} = (\mathbf{S} \times (\mathbf{Ex} \times (\mathbf{LexScope} \longrightarrow \mathbf{RetAdd} \longrightarrow \mathbf{E} \longrightarrow \mathbf{V})))$$

We can now write the semantics.

Semantics 5.7 A Semantics Using Only Finite Domains

Domains

Var : *The domains of variables*

Ex : *The domain of expressions*

RetAdd = Ex

the domain of return address

$\mathbf{LexScope} = \mathbf{Ex}$

the domain of lexical scopes

$\mathbf{F} = (\mathbf{Var} \longrightarrow (\mathbf{Cl} + \{not\_found\}))$

the domain of frames

$\mathbf{L} = (\mathbf{LexScope} \times \mathbf{RetAdd})$

the domain of labels for frames

$\mathbf{P} = \{nil\} + (\mathbf{L} \times \mathbf{P})$

the domain of path names

$\mathbf{S} = \{nil\} + (\mathbf{P} \times \mathbf{S})$

the domain of search lists

$\mathbf{G} = \mathbf{P} \longrightarrow \mathbf{F}$

the domain of trees of frames

$\mathbf{E} = \mathbf{G} \times \mathbf{S}$

the domain of environments

$\mathbf{V} = (\mathbf{Cl} \times \mathbf{G}) \longrightarrow (\mathbf{Cl} \times \mathbf{G})$

the domain of values

$\mathbf{Cl} = (\mathbf{S} \times (\mathbf{Ex} \times (\mathbf{LexScope} \longrightarrow \mathbf{RetAdd} \longrightarrow \mathbf{E} \longrightarrow \mathbf{V})))$

the domain of closures

Functionality

$$scnd(x) = fst(snd(x))$$

$$trd(x) = snd(snd(x))$$

$$\langle x, y, z\rangle = \langle x, \langle y, z\rangle\rangle$$

$$Bind\colon \mathbf{Ex} \longrightarrow (\mathbf{Cl} \times \mathbf{G}) \longrightarrow \mathbf{V}$$

$$Bind(ex)(x, g) = (\ trd(x)\ )(\ scnd(x)\ )(ex)(\ \langle g, fst(x)\rangle\ )$$

$$Apply\colon \mathbf{E} \longrightarrow \mathbf{Var} \longrightarrow (\mathbf{Cl} \times \mathbf{G})$$

$$Apply(\langle e, nil\rangle)(x) = \langle \bot_{\mathrm{Cl}}, e\rangle$$

$$Apply(\langle e, \langle p, r\rangle\rangle)(x) = \ \mathbf{if}\ e(p)(x) \neq not\_found$$

$$\mathbf{then}\ \langle e(p)(x), e\rangle$$

$$\mathbf{else}\ Apply(\langle e, r\rangle)(x)$$

$$e[\mathrm{x} \ := \ v]/\langle a, b\rangle = \langle fst(e)[\mathrm{p} \ := \ f \sqcup fst(e)(p)], s\rangle$$

$$Where\ f = \lambda u.\ \mathbf{if}\ u = x\ \mathbf{then}\ fst(v)\ \mathbf{else}\ not\_found$$

$$\mathbf{and}\ p = Fold(\langle\langle a, b\rangle, fst(snd(e))\rangle)$$

$$\mathbf{and}\ s = Push(p, snd(e))$$

$$I\colon \mathbf{Ex} \longrightarrow \mathbf{E} \longrightarrow (\mathbf{Cl} \times \mathbf{G})$$

Semantics

Exp $\longrightarrow$ **var**

$$\mathcal{I}[\![\mathtt{x}]\!](\langle g,p\rangle) = Apply(\langle g,p\rangle)[\![\mathtt{x}]\!]$$

Exp $\longrightarrow$ λ**var.Exp1**

$$\mathcal{I}[\![\lambda\mathtt{x}.\mathtt{e1}]\!](\langle g,s\rangle) = \langle u,g\rangle \textbf{ Where}$$

$$u = \langle s, [\![\lambda\mathtt{x}.\mathtt{e1}]\!], \lambda a\lambda b\lambda e\lambda v.\mathcal{I}[\![\mathtt{e1}]\!](e[\mathtt{x} \ := \ v]/\langle a,b\rangle)\rangle$$

Exp $\longrightarrow$ **Exp1(Exp2)**

$$\mathcal{I}[\![\mathtt{e1(e2)}]\!](\langle g,p\rangle) = Bind[\![\mathtt{e2}]\!](\ u\ )(\ \mathcal{I}[\![\mathtt{e2}]\!](\langle snd(u),p\rangle))$$

$$\textit{Where } u = \mathcal{I}[\![\mathtt{e1}]\!](\langle g,p\rangle)$$

End of semantics 5.7. ▮

The semantic equations are unchanged in that semantics. The only differences occur in the auxiliary functions handling environments. let's compare them pair wise.

5.6— $\mathbf{P} = \{nil\} + (\mathbf{L} \times \mathbf{P})$

$\mathbf{E} = \mathbf{G} \times \mathbf{P}$

$\mathbf{Cl} = (\mathbf{P} \times (\mathbf{Ex} \times (\mathbf{LexScope} \longrightarrow \mathbf{RetAdd} \longrightarrow \mathbf{E} \longrightarrow \mathbf{V})))$

5.7— $\mathbf{P} = \{nil\} + (\mathbf{L} \times \mathbf{P})$

$\mathbf{S} = \{nil\} + (\mathbf{P} \times \mathbf{S})$

$\mathbf{E} = \mathbf{G} \times \mathbf{S}$

$$\mathbf{Cl} = (\mathbf{S} \times (\mathbf{Ex} \times (\mathbf{LexScope} \longrightarrow \mathbf{RetAdd} \longrightarrow \mathbf{E} \longrightarrow \mathbf{V})))$$

The domain of path names is not changed, however search lists are used in the definition of environments and closures in place of path names. We still use the paths to find the values of variables in environement but those paths have to be stacked in seach lists.

5.6— $Apply\colon \mathbf{E} \longrightarrow \mathbf{Var} \longrightarrow (\mathbf{Cl} \times \mathbf{G})$

$$Apply(\langle e, nil\rangle)(x) = \langle \perp, e\rangle$$

$$Apply(\langle e, \langle l, r\rangle\rangle)(x) = \textbf{if } e(< l, r >)(x) \neq not\_found \\ \textbf{then } \langle e(\langle l, r\rangle)(x), e\rangle \\ \textbf{else } Apply(\langle e, r\rangle)(x)$$

5.7— $Apply\colon \mathbf{E} \longrightarrow \mathbf{Var} \longrightarrow (\mathbf{Cl} \times \mathbf{G})$

$$Apply(\langle e, nil\rangle)(x) = \langle \perp_{\mathrm{Cl}}, e\rangle$$

$$Apply(\langle e, \langle p, r\rangle\rangle)(x) = \textbf{if } e(p)(x) \neq not\_found \\ \textbf{then } \langle e(p)(x), e\rangle \\ \textbf{else } Apply(\langle e, r\rangle)(x)$$

The function *Apply* is modified to search through a search list instead of a series of beginning portions of the current path name. In 5.6, we try the path name $\langle l_1, \langle l_2, \ldots\rangle\rangle$, then the path name $\langle l_2, \ldots\rangle$ and so on so forth each time pulling out a frame and generating a new path name from the rest of the list. This search order is no longer correct in 5.7. We keep the correct search order in the search list; the current path name te be searched being on top of the list. It is not the list itself as in 5.6.

5.6— $e[x := v]/\langle a,b\rangle = \langle fst(e)[p := f \sqcup fst(e)(p)], p\rangle$

Where $f = \lambda u.$ **if** $u = x$ **then** $fst(v)$ **else** not_found

and $p = \langle\langle a,b\rangle, snd(e)\rangle$

5.7— $e[x := v]/\langle a,b\rangle = \langle fst(e)[p := f \sqcup fst(e)(p)], s\rangle$

Where $f = \lambda u.$ **if** $u = x$ **then** $fst(v)$ **else** not_found

and $p = Fold(\langle\langle a,b\rangle, fst(snd(e))\rangle)$

and $s = Push(p, snd(e))$

In 5.7 the path name p is folded before use. Also we have to compute separately the new search list s and include it in the resulting environment.

5.8 Termination of Lexically Scoped Lambda–Calculus

Now let's examine the termination properties of 5.7. We claim that every domain involved in that semantics is finite for all practical purposes. Just look at the domains.

Var : The domains of variables

Ex : The domain of expressions

Only a finite portion of these domains actually occurs in a program.

$\mathbf{L} = (\mathbf{LexScope} \times \mathbf{RetAdd})$

The cartesian product of finite domains is finite.

$\mathbf{P} = \{nil\} + (\mathbf{L} \times \mathbf{P})$

$\mathbf{S} = \{nil\} + (\mathbf{P} \times \mathbf{S})$

These domains are effectively finite because we restrict ourselves to a specific finite subset of all lists in those domains.

$(\mathbf{LexScope} \longrightarrow \mathbf{RetAdd} \longrightarrow \mathbf{E} \longrightarrow \mathbf{V})$

That domain occurs in the definition of closures. We use only a finite portion of it, generated by the abstractions that actually occurs in the program. The abstraction used for environment updates $e[\mathtt{x} := v]/\langle a, b\rangle$ may also force us to use the join of values generated by abstractions. More precisely, the finite subset of $\mathbf{LexScope} \longrightarrow \mathbf{RetAdd} \longrightarrow \mathbf{E} \longrightarrow \mathbf{V}$ we are using is the set Sub defined by the equations:

$$Sub1 = \{trd(I[\![\lambda\mathtt{x}.\,\mathtt{e1}]\!](e)) \mid \lambda\mathtt{x}.\mathtt{e1}\text{occurs in the program}\}$$

$$Sub = \{\bigsqcup\{x \mid x \in s\} \mid s \subseteq Sub1\}$$

The set Sub is finite because $Sub1$ is finite and therefore has a finite number of subsets. The set $Sub1$ is provably finite because there is only a finite number of expressions λ**x.e1** in the program and $trd(I[\![\lambda\mathbf{x.e1}]\!](e))$ is independent of the environment e. The fact we had not proven that there are finitely many environments does no harm here.

$\mathbf{Cl} = (\mathbf{S} \times (\mathbf{Ex} \times (\mathbf{LexScope} \longrightarrow \mathbf{RetAdd} \longrightarrow \mathbf{E} \longrightarrow \mathbf{V})))$

All of **S**, **Ex** and $\mathbf{LexScope} \longrightarrow \mathbf{RetAdd} \longrightarrow \mathbf{E} \longrightarrow \mathbf{V}$ are finite, therefore **Cl** is finite.

$\mathbf{F} = (\mathbf{Var} \longrightarrow (\mathbf{Cl} + \{not\_found\}))$

$\mathbf{G} = \mathbf{P} \longrightarrow \mathbf{F}$

$\mathbf{E} = \mathbf{G} \times \mathbf{S}$

$\mathbf{V} = (\mathbf{Cl} \times \mathbf{G}) \longrightarrow (\mathbf{Cl} \times \mathbf{G})$

The disjoint union, the cartesian product and the function space of finite domains is finite. This exhausts the list domains involved in 5.7.

Last but not least, the type of I is $\mathbf{Ex} \longrightarrow \mathbf{E} \longrightarrow (\mathbf{Cl} \times \mathbf{G})$. This is a finite domain, it satisfies the ascending chain conditions. Therefore the methods of chapter 4 may be used to find a terminating implementation of semantics 5.7.

CHAPTER 6
CONCLUSION

The thesis is over. It is now time to examine how much was achieved and what still needs to be done. Most of chapter 5 is conjecture. As is pointed out in chapter 3, the methods for proving that representations and approximations are indeed representations and approximations are much too tedious to be presently useful. The chain from semantics 5.1 to semantics 5.7 is plausible but its correctness is not proven. We claim there exists a one–to–one correspondence between the abstractions occurring in the program and the effectively used elements of type $\mathbf{LexScope} \longrightarrow \mathbf{RetAdd} \longrightarrow \mathbf{E} \longrightarrow (\mathrm{Cl} \times \mathrm{G})$ that occur in 5.7. This correspondence is similar to what is proven in 4.9. This claim too is left unproven. We could have used an induction similar to the proof of 4.9, but this approach would require a transformation of 5.7 into a treelike representation of the semantics (as in 4.8) and the correctness of that transformation too would have to be left unproven. The first step to be achieved in future work is the development of a method that allow easy demonstrations of when a semantics is the representation or the abstraction of another. The use of representations has such a great importance in all our work that we cannot claim to have a completely formal framework for the writing of semantics until this is done. However, computer scientists are so used to providing representations without clarifying what the mappings are in a denotational sense that we are confident that our work is still significant in the same way that Strachey's research was significant before Scott's showed it was formally correct.

Other future work is the development of an actual data flow analyzer that uses *fix* as a fixed point operator. We know in theory this should work but we do not know how to implement *fix* in order to get good enough performance to make the analyzer useful. Our work is concerned only with theoretical foundations of the method.

This completes the identification of future work. What was achieved is not negligible. The identification of Cousot's lattices with Scott's domains was proved to be productive. We were able to formulate a semantics for forward data flow analysis of LISP. We were also able to clarify the meaning of the bottom element of a Scott domain, showing that lack of information is sometimes meaningful per se. We have shown that the bottom element is not always synonymous with non termination. If we use a representation of the usual semantics, then it is perfectly possible to find a terminating image for a function that was nonterminating in the original semantics. This may lead to powerful unusual methods of evaluation for arbitrarily high order functional programs, but it forces the implementer to be aware of the lattice structure of the denotational domains.

BIBLIOGRAPHY

Aho, A. V. and Ullman, J. D. 1977. *Principles of Compiler Construction*. Reading, Mass.: Addison-Wesley.

Cousot, P. and Cousot, R. 1978. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximations of Fixpoints. *Proc. of the 5th POPL conference*. 238–252.

MacLane, S. 1971. *Categories for the Working Mathematician*. New York: Springer Verlag.

Scott, D. S. 1976. Data Types as Lattices. *SIAM Journal of Computing*. 5 522–587.

. 1981. Lecture on a Mathematical Theory of Computation. In *Theoretical Foundations of Programming Methodology*. ed. Broy, M. and Schmidt, G. 145–292 Dordrecht, Boston and London: D. Reidel Publishing co. reprint Oxford University PRG Monograph no. 19.

. 1982. Domains for denotational semantics. In *Automata, Language and Programming. Nineth Colloquium. Aarhus, Denmark. July 1982. ed. Nielsen, M.* and *Schmidt, E.* 577–613 Lecture Notes in Computer Science, ed. Goos, G. and Hartmanis, J. Vol. 140 Berlin, Heidelberg and New York: Springer–Verlag.

Sharir, M. 1981. Data Flow Analysis of Applicative Programs. In Automata, Languages and Programming. Eighth Colloquium. Acre. July 1981. ed. Even S. and Kariv O. 98–113. Lecture Notes in Computer Science, ed. Goos, G. and Hartmanis, J. Vol. 115 Berlin, Heidelberg and New York: Springer–Verlag.

Stoy, J. E. 1977. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. Cambridge Mass. and London: The M.I.T. Press.