# ON AN IMPLEMENTATION OF ABSTRACT INTERPRETATION

By

DOUG WESTCOTT

B.Sc., The University of British Columbia, 1986

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

July 1988

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date Aug. 31, 1988

# Abstract

This thesis describes an implementation of abstract interpretation and its application to strictness analysis and termination analysis. The abstract interpretation is performed based on a lattice-theoretical model of abstraction, or translation, of functions expressed in a lambda-calculus notation and defined over a concrete domain into functions defined over a user-specified, application-dependent, abstract domain. The functions thus obtained are then analyzed in order to find their least fixed-points in the lattice which is the abstract domain, using a method which is a simplification of the frontiers algorithm of Chris Clack and Simon Peyton Jones. In order to achieve the required efficiency, this method is implemented using lattice annotation, along with constraints upon the annotations. The implementation is then applied to the problems of strictness analysis and termination analysis, deriving useful pre-compilation information for many functions. The concrete domains over which the functions are defined may or may not include lists.

# Contents

# List of Figures

# Acknowledgement

I would like to thank my supervisor, Dr. Harvey Abramson, for his guidance during the preparation of this thesis, and for giving me the freedom to select and pursue the topic of abstract interpretation. I would also like to thank Dr. Alan Wagner for reading the final draft.

Additionally, I would like to thank my fellow students and the department staff for their friendliness and helpfulness during my stay at UBC, and also NSERC and UBC for their unfailing financial support.

# Chapter 1

# Introduction

In this chapter, the idea of abstract interpretation shall be introduced and explained, following which the objectives of this thesis shall be discussed.

## 1.1. Abstract Interpretation

In this section, two important questions with regard to abstract interpretation shall be answered. They are the questions "What is it?" and "Why do it?".

### 1.1.1. What is abstract interpretation?

Abstract interpretation is a process of program analysis which involves both program translation and interpretation. More simply put, it is a technique that enables a computer program to be analyzed without ever being interpreted or executed. That is not to say that during the process of abstract interpretation *no* program is interpreted, but what is interpreted at that time is an *abstraction*, or translation, of the original program. Thus the process is as follows:

1. Program $P$ is to be analyzed.
2. $P$ is translated into $P^a$.
3. $P^a$ is executed, or interpreted.
4. The results of $P^a$'s execution are analyzed, indirectly giving an analysis of $P$.

This process can also be expressed in a diagram, shown in Figure 1.1, as going from $P$ to $info_P$ via

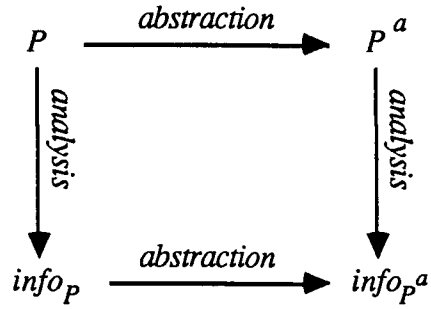



Figure 1.1. The abstraction and analysis process

*abstraction*$^{-1}$ o *analysis* o *abstraction.*

This is possible because in the theoretical framework which is used for abstract interpretation, abstraction is a process of simplification, which can easily be reversed (in a set-theoretical way). The process of abstract interpretation as explained thus far can be more fully illustrated by the following example.

The usual example which is set forth to explain the concept of abstract interpretation is the *rule of signs*. This rule is well known and can be expressed as follows (where $Z$ is the set of all integers):

For all $a,b \in Z$:

$a > 0$ and $b > 0 \Rightarrow ab > 0$
$a > 0$ and $b < 0 \Rightarrow ab < 0$
$a < 0$ and $b > 0 \Rightarrow ab < 0$
$a < 0$ and $b < 0 \Rightarrow ab > 0$

Or, more simply put:

+ x + = +
+ x – = –
– x + = –
– x – = +

where + and – denote any positive or negative integers, respectively.

To apply the process of abstract interpretation to this example, consider the computer program "$a$ x $b$", where $a$ and $b$ are integer variables. Although this particular

program is simple to analyze, in general computer programs are not so easily analyzable, because they require actual execution to determine their outcome. However, in order to demonstrate the usefulness of abstract interpretation as a technique of *general* computer program analysis, this simple program shall be analyzed in the same way as any other, more complex, program would be analyzed using this technique.

What is desired from the abstract interpretation process is the information given by following the *analysis* arrow in the abstract interpretation diagram presented earlier. This can be found by following the arrows around the diagram in a different manner. First, the *abstraction* arrow is followed. This requires a translation of the original program into a new and simpler program; where this new program is defined over a simpler domain than before. Thus the old program "$a$ x $b$" taking values from the integers becomes the new program "$a$ x $b$" (the same in this trivial case) taking values from the set {+, −}. More formally expressed, the old program "$a_Z\ x_{Z\text{x}Z\to Z}\ b_Z$" becomes "$a_S\ x_{S\text{x}S\to S}\ b_S$" (the subscripts denoting variable and function types), where $S = \{+, -\}$, a set greatly reduced in size from $Z$ and which is also an easily reversible simplification of $Z$ (this latter fact will be useful later).

The second arrow to be followed is the *analysis* arrow giving information about the abstracted program. This consists in tabulating the effects of the abstracted program given what is known about the simplified domain $S$. But what is known about $S$ is precisely what is expressed by the *rule of signs*. Therefore, all possible outcomes of the abstracted program can be determined, merely by interpreting that program over all possible values in $S$. Thus, the abstracted program has two possible outcomes:

$+ \text{x} + = +$ $\quad + \text{x} - = -$
$- \text{x} - = +$ $\quad - \text{x} + = -$

The third arrow to be followed (in the reverse direction) is also an *abstraction* arrow. This will give information about the original program based upon information given through analysis of the abstracted program. Since the abstraction process assigns the value "+" to every integer in the set $Pos$ = {positive integers} and "–" to every integer in the set $Neg$ = {negative integers}, the reverse action can be taken and information about the original program can be derived, as shown in Figure 1.2. Notice that given the sign of $a$ and $b$ individually, it is now possible, having done this abstraction and analysis, to assign a sign to the result of "$a$ x $b$" without ever having to actually multiply the integers $a$ and $b$. This process shows itself to be much more powerful when performed on larger and more complex examples, but this simple example serves to illustrate its essential nature.

Information about abstracted program ($s, t \in S$):

$$s = +, t = + \text{ or } s = -, t = - \Rightarrow st = +$$
$$s = +, t = - \text{ or } s = -, t = + \Rightarrow st = -$$

Derived information about original program ($a, b \in Z$):

$$a \in Pos, b \in Pos \text{ or } a \in Neg, b \in Neg \Rightarrow ab \in Pos$$
$$a \in Pos, b \in Neg \text{ or } a \in Neg, b \in Pos \Rightarrow ab \in Neg$$

Figure 1.2. Derivation of information using abstraction information

### 1.1.2. Why do abstract interpretation?

Viewed from a purely theoretical perspective, abstract interpretation is a good thing to do because it allows more information to be derived about a computer program than would otherwise be possible. This in itself is reason enough to do abstract interpretation, because in most areas of scientific research, one cannot tell *a priori* what information is necessary

and what is not. However, since computer science is often concerned with the practical application of scientific knowledge, other justifications for performing abstract interpretation are necessary in order for it to be established as a genuinely useful technique of program analysis. Fortunately, abstract interpretation has proven to have useful applications in areas connected with the optimization of computer programs.

Specifically, when abstract interpretation is performed upon a program, and information is thus derived about that program, changes can then be made to that program or to the program's interpreter which will take into account the new information known about the program and thus speed up or otherwise optimize its execution.

For example, suppose the operation of multiplying two signed integers was vastly more complicated than that of multiplying two unsigned integers. Then the program "$a$ x $b$" described in the *rule of signs* example given previously could be optimized or transformed into a program like this:

*if* ($a > 0$ **and** $b > 0$) **or** ($a < 0$ **and** $b < 0$) *then*
    $abs(a) * abs(b)$
*else*
    $-abs(a) * abs(b)$

where "*" is the operator which multiplies two unsigned integers. Note that even though the program is now much larger than before, the complicated "x" operator is now replaced by a couple of tests and the simple "–", "*", *abs* and boolean operators — giving a great savings in execution time!

Alternatively, the interpreter for the program could be modified to take advantage of the new information known about the builtin "x" operator. Thus, whenever a "x" operation is about to be performed, the two signed integers involved can be tested for their sign (which is then removed), and the resulting unsigned product can then be signed appropriately.

So, even in this simple case of abstract interpretation, it is theoretically possible to apply the new information during a compilation process and thus obtain increases in program efficiency thereafter.

Another, less trivial and more common example of the useful application of abstract interpretation is in the optimization of lazy evaluation of functional programs. Lazy evaluation is the process of taking a functional program and evaluating step by step only what is necessary to be evaluated at each step. Thus in the program "*a* x *b*" the expressions *a*, *b*, and "*a* x *b*" must all be evaluated in order to produce a result, and so lazy evaluation results in the same answer as any other kind of evaluation (which would normally produce correct answers). But in the program "*a* **and** *b*" it is not always necessary to evaluate the expression *b*. For example, if *a* = *false*, then the value of *b* is irrelevant. So in this case, evaluating "*a* **and** *b*" in a lazy fashion will save some work if the value of *b* is not needed. This lazy method of evaluation has proved to be valuable in the interpretation of functional languages.

A problem arises, however, when the functional language interpreter is capable of parallelism. This is because the process of lazy evaluation is a serial process. That is, if expressions were evaluated in parallel, then unnecessary work might be done. In fact, in the case of the program "*false* **and** *b*", if *b* was a non-terminating expression then normal lazy evaluation would produce the result *false*, whereas attempting evaluations in parallel may well initiate an unnecessarily non-terminating computation. This problem can be solved, or reduced, with the aid of abstract interpretation. It is possible to analyze the program before execution and determine where parallel evaluation is definitely safe and where it might not be. This information can then be incorporated into the program, to indicate where the usual lazy mode of evaluation can be safely departed from, thus gaining an increase in the run-time efficiency of the program due to the increased use of parallel

evaluations. So in the case of "*false* **and** *b*", no parallel evaluation would be allowed (in case *b* was a non-terminating expression), but in the case of "*a* x *b*", all of *a*, *b*, and "x" can be evaluated in parallel because it is known that all of those values will be needed in order to produce an answer.

## 1.2. Thesis Objectives

This thesis has two main objectives. The first is to describe an implementation of abstract interpretation which has been successful in that it is generalized and efficient, and the second is to describe its successful application to two distinct problems: strictness analysis and program termination analysis.

### 1.2.1. Implementation of abstract interpretation

The primary objective of this thesis is to demonstrate an implementation of abstract interpretation which meets certain requirements; specifically, an implementation that automates and generalizes the process of abstract interpretation to a degree that has not previously been demonstrated. Earlier work in this area has been primarily theoretical in nature, and what implementation has been done has been limited in scope and effectiveness. The aim of this implementation of abstract interpretation is to overcome some of those limitations, primarily those relating to the powerfulness or generality of the abstraction mechanism, but also those connected with the efficiency of the abstraction process. These goals shall now be discussed in more detail, following a brief description of how the abstraction process is accomplished using the unifying mathematical concept of the *lattice*.

#### 1.2.1.1. How is abstract interpretation done?

Effective abstract interpretation is made possible by one very important integrating concept — the lattice — in terms of which all aspects of abstract interpretation, the programs being

analyzed, and the domains those programs operate over and give results in, are viewed. That is, the program to be analyzed is viewed as a function from one lattice to another, the values of variables in the program and the results of the program are themselves values in a lattice, the abstracted program is also a function over lattices and is obtained though lattice operations, and the information derived from the abstraction is computed with respect to a lattice. This method of treating abstract interpretation serves to give the process a solid mathematical basis and a good theoretical framework for further research, something which previous *ad hoc* approaches have lacked.

The main feature of a lattice is that it is a set which is partially ordered (although there are a few further restrictions which will be mentioned later). That is, it is a set consisting of a number of elements, called nodes, between any two of which may exist a less-than relationship. Thus, a normal set can be considered as a lattice which has very few (in fact, zero) relations among the nodes, and a set such as the integers can be considered as a lattice in which every node is ordered with respect to every other node. An example of a more common type of lattice is that shown in Figure 1.3, where the lattice is the set of all subsets of the set {1, 2, 3}, and the relations between the nodes are defined by the "subset" relation. The diagram shows these relations as arcs connecting the nodes; node A is less-than node B if and only if it is connected to node B by a pathway of arcs proceeding upwards only. Notice that direction of arc traversal is important in a lattice diagram. As such, lattices are like directed graphs, with the restriction that no cycles may exist in the graph.

A further, more precise, definition of lattices will be given in the next chapter; the preceding description will suffice for now.

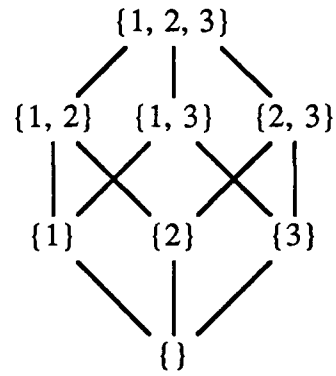



Figure 1.3. The powerset lattice of the set {1, 2, 3}

Given the lattice upon which a computer program is defined, the abstraction lattice upon which the translated program will be defined, and the abstraction function which performs the translation, it is a simple matter to compute the abstraction function upon the original program to come up with the abstracted program. The restrictions upon the abstraction function are few, only existing to ensure that the function is well-behaved in certain ways. This enables the theoretical foundation for the process to hold the result of the abstraction accountable to certain rules, thus allowing a theoretical "stamp of approval" to be given to the process as a whole. Once the abstraction is performed, the resulting abstract program must then undergo a fixed-point finding process, which reduces any recursion in the program to the point where it is the most manageable (guaranteeing termination upon execution, if possible). This fixed-point finding process is actually a part of the abstraction function, but the fixed-point finding method is always the same, whereas the abstraction function changes from application to application, so the two processes are separated and treated individually. What the fixed-point finding process involves is a search through a function lattice for the best function to represent the abstracted program under analysis. With reference to a lattice diagram, this search is always upwards moving, thus if the function lattice is finite, it is guaranteed to terminate. Once the fixed-point is found, the

computation of the fixed-point (itself a function) over all possible arguments represents the analysis of the abstracted computer program. It then remains up the to user to apply the abstraction process in reverse (a simple conceptual step) and to use the information thus derived to optimize or otherwise affect the original program.

Now that the abstract interpretation process has been explained in some detail, the goals of this thesis can be expressed more meaningfully in the following sections.

#### 1.2.1.2. Generality

The primary goal of this implementation of abstract interpretation is to perform the function of abstract interpretation in a generalized manner. That is, where previous implementations have placed restrictions upon the types of functions to be abstracted, and lattices to be used, this implementation will allow more possibilities, with fewer restrictions, in order to give a more powerful tool for performing this process. In the past, the two major points of restriction have been upon the types of functions to be abstracted, and upon the type of abstraction lattice to be used. These restrictions, and their removal, shall now be explained in more detail.

A major difficulty which has come up often in the history of abstract interpretation is the question of how to deal with higher-order functions — that is, functions which take functions as arguments. Oftentimes little information could be derived about such functions, and until recently, no theoretical basis existed for the analysis of such functions. Now that such a theoretical basis does exist (see, for example, [Burn, Hankin & Abramsky, 1986]), higher-order functions can be treated the same as any other kind of function during the abstract interpretation process, and cause no extra difficulty.

Another restriction which has been placed upon both practical implementations and theoretical explications of the process of abstract interpretation is one which only allows lattices of a particular type (*flat* — which will be explained later). Again due to recent

theoretical developments, such a stringent requirement is no longer necessary, although it is unfortunately still mandatory for the lattice over which the abstracted program is defined to be (in a certain sense) finite. The relaxation of this last restriction currently makes it impossible to guarantee termination of the fixed-point finding process.

#### 1.2.1.3. Efficiency

Another goal of this implementation is to demonstrate the efficiency, and thus practicality, of a certain method of performing the fixed-point finding process. Historically, this has been a computationally intensive procedure, but recently (see, for example, [Clack & Peyton Jones, 1985]) has been improved by the use of an approach called the "frontiers" method, the most efficient method for finding fixed-points used so far. What is demonstrated in this implementation is the effectiveness of a simplification and generalization of the frontiers method (actually, such a simplification that it no longer bears much resemblance to the original method).

### 1.2.2. Demonstration of applicability

The second objective of the implementation is to demonstrate its applicability to different kinds of abstract interpretation problems. Two types of abstract interpretation problems have been selected as examples of the effectiveness of this implementation; they are strictness analysis and program termination analysis, and will now be described briefly.

#### 1.2.2.1. Strictness analysis

Strictness analysis is a common problem on which to perform abstract interpretation. Often, the problem of abstract interpretation is formulated only in the context of strictness analysis. As described previously, the process of strictness analysis is the analyzing of a program to see at what points departures from the normal mode of lazy evaluation may be safely made, in order to take advantage of the parallelism available in the computer. Much

success has been demonstrated in the application of abstract interpretation to the problem of strictness analysis, and the application of this implementation of abstract interpretation to strictness analysis is simply to demonstrate its generality and usefulness.

#### 1.2.2.2. Termination analysis

Termination analysis is concerned with whether a given program will terminate or not. This is an unsolvable problem, and no pretension of having actually discovered a general solution will be made here. However, some useful information may be derivable about a program after a not-too-excessive amount of computation has been performed, so this problem can still be viewed as a justifiable one to perform abstract interpretation on. Typically, the information derived about a program will be for what input arguments it will definitely not terminate. Other inputs to the program may or may not allow it to terminate, but due to the hard nature of this problem, such cannot be determined.

Both termination analysis and strictness analysis shall be performed upon a simple functional language, one which is defined without lists, and also upon a functional language which incorporates list structures, in order to demonstrate the general applicability of this method of program analysis.

# Chapter 2

# Abstract Interpretation

Now that the concept of abstract interpretation has been introduced, the nature of its theoretical base — the lattice — can be expanded upon, followed by an in-depth explanation and example of the way in which lattices are used to do abstract interpretation.

## 2.1. Lattice Preliminaries

Before the lattices used in abstract interpretation can be discussed in detail, it is necessary to present some preliminary information regarding lattice theory.

### 2.1.1. General information

Following is some general information about lattices. First, a precise definition is given. This is followed by a discussion of lattice semantics. For a more in-depth look at some of the lattice concepts mentioned here, see [Stoy, 1977].

#### 2.1.1.1. Definition

As mentioned earlier, a lattice is a partially-ordered set $S$ in which the elements of $S$ satisfy two additional requirements beyond those normally associated with partial-orderedness. These requirements will be discussed later. As for the partial-orderedness of $S$, this consists in the elements of $S$ having the possibility, but not the necessity, of being ordered by a binary ordering relation "$\leq$". For partially-ordered sets in general, the ordering given

to the set elements is unrestricted, except for the following three rules, which are necessary to allow a consistent and useful theoretical framework to be built:

1. For all $x \in S, x \leq x$.
2. For all $x, y \in S, x \leq y$ **and** $y \leq x \Rightarrow x = y$.
3. For all $x, y, z \in S, x \leq y$ **and** $y \leq z \Rightarrow x \leq z$.

Given a partially-ordered set S then, one can say it is also a lattice exactly when it satisfies the following additional two rules:

4. For all $x, y \in S$, *lub* $\{x, y\}$ exists.
5. For all $x, y \in S$, *glb* $\{x, y\}$ exists.

where the complementary concepts of *least upper bound* (*lub*) and *greatest lower bound* (*glb*) are defined as follows (where, if $X$ is a subset of $S$ and $y \in S$, then $X \leq y$ means "$x \leq y$ for all $x \in X$", and $y \leq X$ is defined similarly):

- *least upper bound* — if $S$ is a partially ordered set and $X$ is a subset of $S$, then let $X' = \{ x \in S \mid X \leq x \}$. Then if $X'$ is non-empty and contains an element $s$ such that $s \leq X'$, then that element $s$ is the least upper bound of $X$. That is, *lub* $X = s$. Note that for arbitrary partially ordered sets such an element $s$ does not have to exist.

- *greatest lower bound* — this can be defined in a manner similar but complementary to the definition above. That is, to find the greatest lower bound of $X$, where $X$ is a subset of $S$, let $X' = \{ x \in S \mid x \leq X \}$, Then, if $X'$ contains an element $s$ such that $X' \leq s$, then *glb* $X = s$. In this case also, the greatest lower bound $s$ does not have to exist.

### 2.1.1.2. Semantics

Given the flexible nature of the ordering relation in a lattice structure, that relation and structure have been used to represent many different types of information. The most common meaning of the "≤" relationship among elements, or nodes, of a lattice historically may have been that of dependency. For example, many textbooks contain in the preliminary pages a diagram similar to Figure 2.1, indicating, for example, that to understand chapter 6, one must first understand chapters 2 and 4, and therefore one must also understand chapters 1 and 3, of which chapter 1 is sufficient for the understanding of

chapter 3. But this is not the semantics normally given to node relations in lattices used in the context of abstract interpretation.

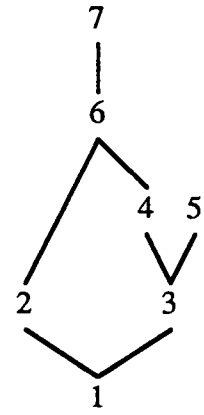



Figure 2.1. An example of textbook chapter dependencies

The interpretation given to lattice nodes and relationships in abstract interpretation lattices is that of approximation. That is, as one proceeds "up" the lattice from a "bottom" node to a "top" node, one's information becomes more and more correct, or constrained, and is less and less of an approximation. An example of this is the powerset lattice given previously, except inverted, as shown in Figure 2.2, where $X \leq Y$ means $X$ contains $Y$ as a subset. In this example, the element {1, 2, 3} is at the bottom of the lattice, and is the least informed or selective, whereas as one moves up the lattice the elements become more and more informed, until finally at the top of the lattice the element {} is found, which indicates information that is over-constrained, or too selective. This is the type of semantics which will be given to the lattices used in the abstract interpretation process from this point on.

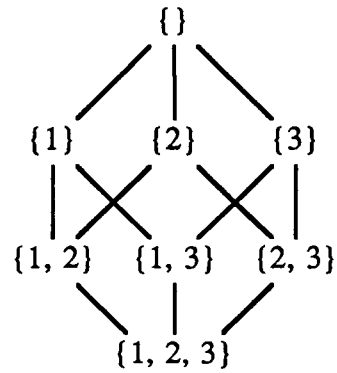



Figure 2.2. Inverted powerset lattice

## 2.1.2. Specific concepts

As is normally the case with mathematical objects, lattices have been categorized and combined in various ways, resulting in a rich variety of higher-order concepts. Some of these categories and constructions shall be explained in the following sections.

### 2.1.2.1. Lattice types

Due to the lack of restriction placed upon lattices up to this point, there is room for a further categorization of lattice types in order to allow a more powerful theoretical framework to be built. One restriction which can be placed on an abstract interpretation lattice to facilitate (in fact, to make feasible at all) computation is that of finiteness. That is, the abstract interpretation lattice must consist of either a finite set or a composition of finite sets (the exact type of composition will be explained later). Other categorizations commonly applied to lattices used in abstract interpretation include the following.

#### 2.1.2.1.1. Complete lattices

A lattice $L$ is *complete* if both *lub* $X$ and *glb* $X$ exist for every subset $X$ of $L$. It turns out that if $L$ is not complete, then $L$ can easily be made complete (by adding new elements as required so that the necessary *lub*'s and *glb*'s do exist), a fact that is important in the development of a supporting theoretical framework for abstract interpretation.

#### 2.1.2.1.2. Flat lattices

One type of lattice which appears very often in abstract interpretation is the *flat* lattice. A lattice $L$ is flat if the only ordering relationships which exist among the elements of $L$ are those between two distinguished elements of $L$ — *top* and *bottom* — and the rest of $L$. That is, if $x, y \in L$, and $x \leq y$ but $x$ and $y$ are different elements, then one of $x$ or $y$ must be *top* or *bottom*, as appropriate (*bottom* $\leq$ *top*, of course). A typical example of this type of lattice is the flat lattice of integers, as shown in Figure 2.3. This is a common way ofrepresenting elementary data types in a programming language when used in connection with abstract interpretation.

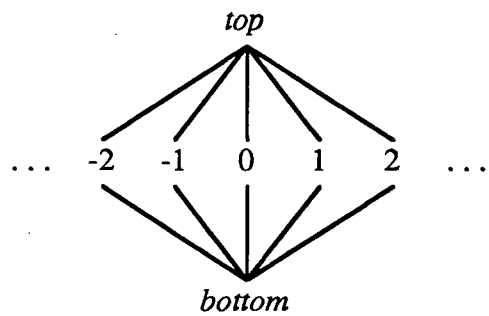



Figure 2.3. The flat lattice of integers

### 2.1.2.2. Lattice compositions

A useful method for creating lattices with a regular structure is to compose two or more lattices together to obtain a single, composite, lattice. All of the following types of compositions can easily be extended to compositions of more than two lattices; they are merely defined here as binary relations in order to simplify their definitions.

#### 2.1.2.2.1. Product lattices

The lattice $L_1$ x $L_2$ obtained as the *product* of the two complete lattices $L_1$ and $L_2$ has the following structure: Each element in $L_1$ x $L_2$ is a pair $(l_1, l_2)$ with $l_1 \in L_1$ and $l_2 \in L_2$.

Every possible such pair is present in $L_1$ x $L_2$, and these pairs are ordered so that $(l_1, l_2) \leq (m_1, m_2)$ exactly when $l_1 \leq m_1$ **and** $l_2 \leq m_2$. It can be shown that since the lattices $L_1$ and $L_2$ are complete, then so is the product lattice $L_1$ x $L_2$. One example of such a product lattice is shown in Figure 2.4.

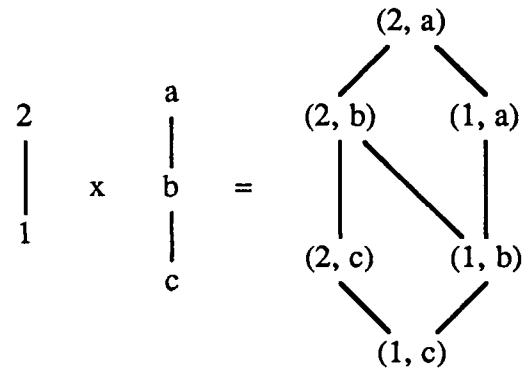



Figure 2.4. An example of a product lattice

### 2.1.2.2.2. Sum lattices

The lattice $L_1 + L_2$ defined as the *sum* of the two complete lattices $L_1$ and $L_2$ has the following structure: Each element $l_1 \in L_1$ and $l_2 \in L_2$ can be found in $L_1 + L_2$, along with two additional, new elements, *bottom* and *top*. The ordering relationships among the elements of $L_1 + L_2$ are defined so that $l_1 \leq l_2$ exactly when either $l_1, l_2 \in L_1$ and $l_1 \leq l_2$ or $l_1, l_2 \in L_2$ and $l_1 \leq l_2$, except for the new elements *top* and *bottom*, which have the property that *bottom* $\leq l$ and $l \leq$ *top* for all $l \in L_1 + L_2$. One example of such a sum lattice is that shown in Figure 2.5.

It follows from the definitions that if $L_1$ and $L_2$ are both complete lattices, so is $L_1 + L_2$. This sum operation is sometimes called a *separated* sum, in order to distinguish it from other kinds of sum operations. It is, however, the only kind of sum appearing in this thesis.

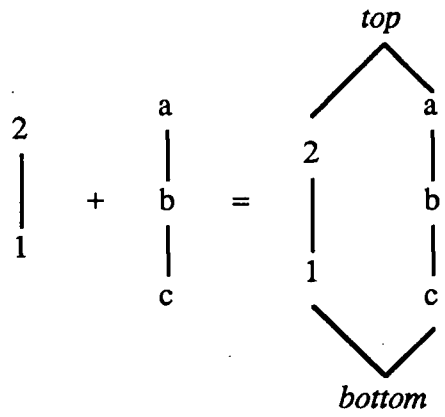



Figure 2.5. An example of a sum lattice

### 2.1.2.3. Lattice functions

No theory concerning lattice structures is complete without defining functions which operate within, and between, lattices, as opposed to the sum and product composition functions, which operate on lattices. Functions between two lattices abound in general, and are not difficult to work with, if they fall into one or more of the following categories: equal, monotonic, and continuous.

Two functions $f, g : L_1 \rightarrow L_2$ are *equal* exactly when they produce equal results for every possible argument. That is, $f = g \Leftrightarrow f\,x = g\,x$ for all $x \in L_1$. Determining if two functions are equal is a computationally difficult task. One example of this is the pair of functions $f, g : Z \rightarrow Z$:

$$f\,x = x + x$$
$$g\,x = 2 * x.$$

Although $f$ and $g$ are equal, one may have to compute an infinite number of cases to determine this.

A lattice function $f : L_1 \rightarrow L_2$ is *monotonic* if for any two arguments $x_1, x_2 \in L_1$, $x_1 \leq x_2 \Rightarrow f\,x_1 \leq f\,x_2$. This formalizes the notion concerning lattice semantics, that if more (no less) information is known about one argument of a function with respect to

another, then it can be expected (for such well-behaved functions) that the same relationship will hold for the results. As is obvious from the definition, compositions of monotonic functions are also monotonic.

A lattice function $f: L_1 \rightarrow L_2$ is *continuous* when it is monotonic and for all subsets $X$ of $L_1$, $f\ (lub\ X) = lub\ \{ f\ (lub\ X') \mid X'$ is a subset of $X$, and $X'$ is finite$\}$. That is, $f$ is extremely "well-behaved". In fact, when the lattices $L_1$ and $L_2$ are finite, then monotonicity of $f$ guarantees continuity of $f$. As usual, compositions of continuous functions are themselves continuous.

Functions can also be ordered and placed in a lattice framework, where the ordering of two functions $f, g : L_1 \rightarrow L_2$ is defined as $f \leq g \Leftrightarrow f\ x \leq g\ x$ for all $x \in L_1$, and the usual set of functions that $f$ and $g$ are drawn from is $[L_1 \rightarrow L_2]$, defined as the lattice of all continuous functions mapping $L_1 \rightarrow L_2$.

One example of this function lattice is the lattice $[L \rightarrow L]$ composed of all continuous functions from $L \rightarrow L$, where $L$ and $[L \rightarrow L]$ are as shown in Figure 2.6.Note that it is possible to define one other function $f: L \rightarrow L$, as

$$f\ x = \lambda\ x\ .\ if\ x = 1\ then\ 2\ else\ 1$$

but that $f$ is not present in the function lattice $[L \rightarrow L]$ because $f$ is not monotonic ($1 \leq 2$ but $f\ 1 = 2, f\ 2 = 1$, and therefore $f\ 1 \leq f\ 2$ is not true) and therefore not continuous.

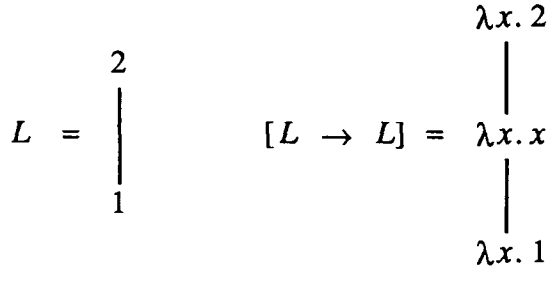



Figure 2.6. An example of a function lattice

## 2.2. Lattices for Abstract Interpretation

Out of the infinite number of lattices available for use in abstract interpretation, usually two special lattices are chosen and used throughout the process. One of these is the *concrete* lattice, and the other is the *abstract* lattice, and the abstract interpretation process is a mapping from concrete to abstract lattices. Both are function lattices, composed from simpler lattices, and will be defined and discussed later, following a brief explanation of notation used in naming their component lattices.

### 2.2.1. Notation

Since both of the concrete and abstract function lattices are built by the composition of an infinite number of simpler lattices, it simplifies their description to use a formal notation to identify the component lattices. First, a few lattices are identified as base lattices; these are lattices consisting of the primitive elements contained in the concrete and abstract lattices. Some examples are the lattices of *integers* and *booleans* (see Figure 2.7).

Let these base lattices be the elements of the set *Bases*. Thus for the above example, $Bases = \{integers, booleans\}$, where *integers* and *booleans* are the names given to the lattices described above. Then, every other lattice in the composition is a function lattice, composed of functions of a particular type. This is expressed as follows: $f$ is a function of type $T$, where $T$ is a *type expression*, and a type expression is one of two things:

1. A primitive type $a \in Bases$, or
2. A composite type $a \rightarrow b$, where $a$ and $b$ are both type expressions.

Thus, a few example type expressions are *integers*, *booleans*, $integers \rightarrow booleans$, $(integers \rightarrow integers) \rightarrow booleans$, and so on. The type expression

$$(integers \rightarrow integers) \rightarrow booleans$$

corresponds to a function that maps integer-valued functions on integers to one of the values in the lattice of booleans.

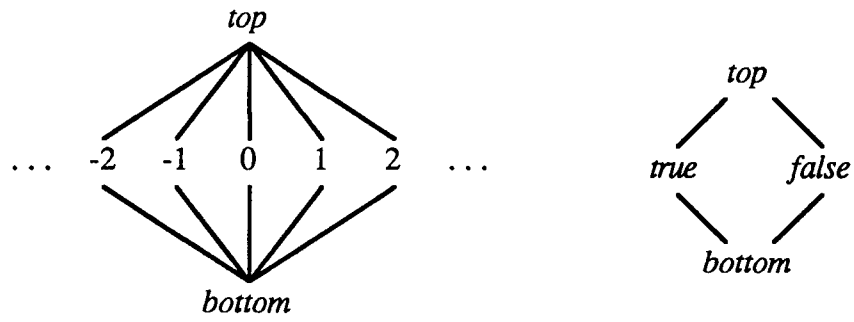



Figure 2.7. The concrete lattices of *integers* and *booleans*

The composition of these function lattices can now be accomplished with relative ease. Let $L = + \{ L_t \mid t \in TE \}$, where $TE$ is the set of all possible type expressions, and $L_t = [L_a \rightarrow L_b]$ if $t = a \rightarrow b$, and $L_t = t$ if $t \in Bases$. Then $L$ is the sum lattice obtained by the composition of all the base lattices and all lattices of continuous functions thereby defined. That is,

$$L = integers + booleans + [integers \rightarrow booleans] + [integers \rightarrow integers] + ...$$

Note that although $L$ itself is necessarily an infinite lattice, each of its component lattices may still be finite, even though this is not the case above.

Both of the concrete and abstract lattices can now be easily described using these notions.

## 2.2.2. Concrete lattice

The concrete lattice used in the abstraction process is the lattice containing the functions to be abstracted. It is composed as described in the previous section, where the base lattices contained in *Bases* are usually the *integers*, *booleans*, *reals*, *characters*, *lists*, and other user-defined data-types, although this list may be shortened for some implementations of abstract interpretation. Such a reduced set of base lattices is used to allow the abstract interpretation implementation to concentrate on a select few areas, without loss of generality. In this implementation of abstract interpretation, the set *Bases* consists of the

types *integers*, *booleans*, and sometimes *lists*; the other data-types being redundant and basically unnecessary for the purposes of the implementation.

### 2.2.3. Abstract lattice

The abstract lattice used is dependent upon the type of information desired from the abstraction process. It is composed as an infinite sum in the same way as the concrete lattice, but has for its base components simpler lattices than the lattices *integers* and *lists*. For this implementation, the base lattices must all be finite, in order to ensure the termination of the abstract interpretation process, part of which involves searching though a function lattice. Other restrictions are sometimes imposed upon the base abstract lattices, such as that they must be complete lattices, or flat lattices. These restrictions are made to ensure that the theoretical underpinnings of the abstract interpretation process will guarantee its success.

Currently, the only requirement that is restrictive in the selection of abstract lattices is that the base lattices be finite. For example, were one wanting to analyze a program for data-type information, one could not use a lattice of data-types as a base abstract lattice, because it is infinite. On the other hand, many useful abstract base lattices which are also finite have been employed in the abstract interpretation process, and with good results, so the finiteness restriction is workable, if not desirable.

## 2.3. Methods of Abstract Interpretation

Progress in the area of abstract interpretation has sometimes gone on independently in each of two different aspects of the area. At first, most work was done in getting the theoretical framework established and verified, while the practical side of the area received little attention. Then, more recently, as a minimal theoretical basis was established for the process, and work on this side of the area slowed, the practical issues in implementing

abstract interpretation were emphasized. Currently, each side of the problem is making slow but sure progress, with both the theoretical and practical aspects of the research contributing to the work as a whole.

In the next two sections, both of these aspects of the process of abstract interpretation shall be discussed: First, the historical development of the theoretical underpinnings for this process will be presented, followed secondly by a description of some of the practical solutions devised for the problem.

## 2.3.1. Theoretical basis

The history of the theoretical development of abstract interpretation can be divided into two parts, the first consisting solely of the Cousots' seminal work on the subject [Cousot & Cousot, 1977], and the second being the work which follows. This later work is cast in a common framework based on a functional approach which is different than the Cousots' original approach, which was to develop the theory entirely within the domain of imperative programming languages. Thus, the next section describes the Cousots' work, complicated as it is by its selection of the imperative style of function representation. This is followed by a section describing the later work done on abstract interpretation's theoretical basis, all of which has in common the functional style.

### 2.3.1.1. Cousots

Patrick and Radhia Cousot are held to be the pioneers of the current approach taken with respect to the problem of abstract interpretation. Their 1977 paper [Cousot & Cousot, 1977] provided a much-needed mathematical definition and foundation for the process in a lattice-theoretical form. Not only did they establish the basic foundations for the process, they extended it in many ways due to their use of the lattice as a framework.

The first thing the Cousots did was to define the syntax and semantics of a simple, imperative, flowchart language. Thus, a program, or function, was a finite flowchart, or equivalently, a finite directed graph. This representation of the program under analysis would be its most concrete interpretation. The Cousots then went on to define more abstract interpretations. To do this, they defined the concept of a context, $C_q$, which was associated with each arc $q$ in a program $P$, and which was "the set of all environments which may be associated to $q$ in all the possible computation sequences of $P$" (an environment being the usual association of variable names and values). Then, a context vector, $C_v$, associates a context with each of the nodes in a program. This context vector is an enumeration of all of the possible execution sequences of $P$.

The contexts of $P$ can then be made naturally into a complete semi-lattice of abstract contexts *A-Cont* (in fact, *A-Cont* is a complete lattice), where *A-Cont* is defined as an abstract interpretation of $P$. This abstract interpretation of $P$ is distinguished from other more or lesser informed abstract interpretations of $P$ in that for any computation sequence of $P$, $C_v(q)$ contains lesser or more associated environments. In fact, all of the abstract interpretations of $P$ themselves form a lattice, with the ordering among the elements (each element is an *A-Cont*) reflecting the distinction between more and lesser informed abstract interpretations. For example, one *A-Cont* $A_1$ may contain the possibility of a certain variable representing 0 at a certain point in the program, whereas another *A-Cont* $A_2$ may not. Then if $A_1$ and $A_2$ are in other respects the same, it is the case that $A_1$ is more informed than $A_2$.

The Cousots discuss in depth the consequences of this definition of abstract interpretation, covering topics such as its consistency, its limitations, and how to approximate fixpoints (solutions to systems of recursive equations, involving a search through a lattice) in infinite lattices. However, these aspects of their work shall not be

discussed further here. It is possible to present matters such as those much more clearly when viewed from a functional perspective, as is done in more recent work than that of the Cousots.

#### 2.3.1.2. Later work

A recurring name in the area of abstract interpretation in the years following the Cousots' work is that of Alan Mycroft. His doctoral thesis [Mycroft, 1981] and other parts of his published work concentrate on establishing a sound lattice-theoretical basis for the process of abstract interpretation, especially in connection with its practical application to the optimization problem of strictness analysis. Another important work in this area, also concerned especially with strictness analysis application, is [Burn, Hankin & Abramsky, 1986], which represents accurately the current state of research in the area of abstract interpretation, at least on the theoretical side.

Mycroft's first published work on abstract interpretation [Mycroft, 1980] addressed the issue of transforming the *call-by-need* method of parameter passing in applicative programs into *call-by-value* wherever possible (call-by-need is the same as call-by-name except that the first use of the parameter causes not only its evaluation but also its replacement with the result of the evaluation, thus making subsequent references to the parameter computationally very inexpensive). The problem is to do this whenever possible throughout the program text, and yet to avoid introducing argument evaluations which may be unnecessary. That is, the problem is essentially that of strictness analysis, except for the fact that parallelism is not available as a means of efficiently utilizing the resulting call-by-value transformations.

Mycroft solves the problem by defining interpretations of the applicative programs which are different than the usual interpretation. His alternate interpretations are translations of the applicative program defined over one lattice into a system of recursive equations

defined over the two element boolean lattice {0, 1}, with $0 \leq 1$. The consistency of this method of program interpretation and its associated semantics is argued by recourse to the lattice-theoretical basis upon which they are built. Additionally, since the applicative programs under transformation or interpretation are themselves systems of recursive equations, giving recursive equations as the result of translation, it is necessary to define a fixed-point finding method which is guaranteed to terminate (that is, the fixpoint of the resulting expression is to be found using the text of the expression, not through its evaluation). The method used by Mycroft is the now usual method of following the *Ascending Kleene Chain* (AKC) upwards from the bottom of the function lattice which contains the results of the alternate abstract interpretations.

Later, this was generalized into a comprehensive method of abstract interpretation, and theoretically justified for first-order functions on flat base domains in some of Mycroft's other work, such as [Mycroft, 1981] and [Mycroft & Nielson, 1983]. The lattice-theoretical construct used to prove the consistency and validity of this approach to abstract interpretation is called a *powerdomain* — a lattice consisting of all subsets of another lattice, ordered by inclusion. Depending on the particular type of powerdomain desired, different modifications to this basic definition can be made, each having a different effect on what can be proved about this lattice-based method of abstract interpretation. Thus, a different powerdomain construction than Mycroft's was used in [Burn, Hankin & Abramsky, 1986], resulting in the ability to prove Mycroft's method of abstract interpretation correct for arbitrary (including higher-order) functions defined over flat base domains.

At this point in the theoretical development of abstract interpretation, attention has turned to the problem of justifying abstract interpretation over non-flat base domains, in order to legitimatize currently implemented systems which perform abstract interpretation

on functional programs incorporating lists, and other user-defined, non-flat data-structures. One attempt made in this direction has been the work of John Hughes, in [Hughes, 1985] and [Hughes, 1987], where he uses the concept of the *context* in which an expression may be evaluated in order to analyze programs using lists.

At first, this concept was informally defined, but in [Hughes, 1987] he clarified it as an "abstraction of sets of continuations," thus giving a solid, if non-intuitive, basis for his method. It turns out, however, that using this method of context analysis leads one into difficulty in trying to solve certain recursive functional equations, and when deciding upon which heuristics to use to introduce the correct approximations. An additional drawback is that the theoretical justification, and the method itself, apply only to first-order, untyped languages.

These problems were alleviated somewhat by Wadler and Hughes [Wadler & Hughes, 1987], who proposed the use of *projections* as opposed to continuations for justifying context analysis when dealing with programming languages defined over non-flat base lattices. A lattice projection $p$ is a function which simplifies, or approximates, lattice elements. It satisfies the properties

1. $p\ x \leq x$,
2. $p\ (p\ x) = p\ x$.

By using projections to deal with contexts, thus giving contexts a theoretical basis, all of the above-mentioned difficulties with continuations can be avoided. Unfortunately, projections currently justify abstract interpretation over non-flat base lattices only for first-order untyped functions. However, as for flat base lattices, Wadler and Hughes are optimistic that the theory can be extended.

Absence of a complete theoretical framework for abstract interpretation has tended to do little to stop the progress of practical work in the area, however, and such progress

has occurred, despite its theoretical status. Some of these practical developments shall now be discussed.

### 2.3.2. Practical developments

Once abstract interpretation was viewed from a functional perspective, it didn't take long for practical implementations to be created (see, for example, [Mycroft, 1980]). Initially, these dealt with the problems of program termination and strictness analysis for programs which were first-order and defined over flat base lattices. Later on, applications to data-type analysis were implemented (see, for example, [Mishra & Keller, 1984]), and also further applications to strictness analysis (see, for example, [Clack & Peyton Jones, 1985]), all designed to work with higher-order functions. Additionally, implementations described in [Mishra & Keller, 1984] and [Hughes, 1985] are designed to perform abstract interpretation on functions defined over non-flat base lattices, although the functions are still untyped.

One computationally expensive part of the abstract interpretation process is finding least fixpoints for the translated system of recursive functions defined over the abstract lattice. This problem has usually been solved by computing the AKC and comparing the successive functions derived to determine equality, and thus the point at which to halt. However, the comparison of two functions for equality, whether done by exhaustive computation on all possible arguments, or textually (by specifying the function in a certain canonical manner), is computationally expensive. Thus, to have any type of strictness analysis or other type of program analysis performed on a regular basis, as for example, in an optimizing compiler, it would be desirable to discover an inexpensive means of finding fixpoints.

Fortunately, an algorithm proposed in [Clack & Peyton Jones, 1985], known as the frontiers algorithm, has proved to be more faster at finding fixpoints than the aforementioned approach. The advantage of the frontiers algorithm is its use of the lattice structure existing upon the set of all arguments to a function. This enables the algorithm to avoid computing the function upon certain arguments. That is, the frontiers algorithm adheres to the definition of functional equality in that it compares function values over all possible function arguments, but it does so in a way which saves some unnecessary computation. This is done by defining frontiers within the function's result lattice which allow some function values to be deduced rather than computed. As an example (one which also appears in [Martin & Hankin, 1987]), suppose a function over a concrete domain is abstracted to give a function f over an abstract domain $A, f : A \rightarrow A$. Suppose also that the single base lattice of the abstract domain $A$ is the two element lattice $2 = \{0, 1\}$, where $0 \leq 1$, and $f : 2 \rightarrow 2 \rightarrow 2$ (or $f : 2 \times 2 \rightarrow 2$) is defined as $f\,x\,y = x$ **and** $y$. Then, the product lattice of all possible argument pairs to $f$ is as shown in Figure 2.8, and each element of this lattice, when computed upon by $f$, determines a value in the result lattice 2. Suppose that the value of $f\,1\,0$ is computed and found to be 0. Now,instead of having to compute $f\,0\,0$ explicitly, it can be deduced from the fact that $(0, 0) \leq (1, 0)$ in the argument lattice and from the monotonicity of $f$ that the value of $f\,0\,0$ must be 0. The 0 value for $f$ at the point (1, 0) is part of a frontier. That is, it determines a point $p$ in the argument lattice and a value $v$ such that every point in the argument lattice below $p$ also must have the value $v$. Martin and Hankin also explain how to expand the frontiers in the argument lattice to efficiently discover function values for all arguments and completely determine the function. Function equality can then be quickly and easily determined by comparing the two argument lattices with their associated values.

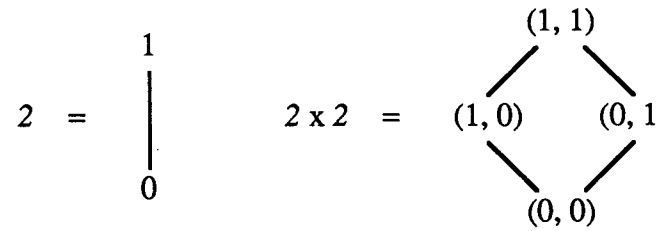



Figure 2.8. The product lattice *2* x *2*

There are a few drawbacks associated with the frontiers algorithm as presented in [Martin & Hankin, 1987]. These are its use of multiple-argument first-order functions and its close ties to the base domain *2*. These problems can be overcome, and Martin and Hankin do present extensions to the frontiers algorithm which cope with more general cases, but by eliminating the dependence upon multiple-argument functions and by incorporating the concept of an annotated lattice, the algorithm can be greatly simplified, as will be shown later on.

## 2.4. Detailed Example

To pull together the concepts mentioned thus far, and to help give a proper perspective upon the ideas yet to be discussed, an example of abstract interpretation follows. The domain of abstraction is the one based on the two-element lattice *2*, described previously, and will be used in strictness analysis; thus the element 0 has the meaning of definite non-termination, while the element 1 means termination or non-termination. The concrete lattice is as usual, based upon integers and booleans, the only necessary data-types for this example. The translation portion of the abstraction process follows.

| **Concrete** | **Abstract** |
|---|---|
| $i \in$ *integers* | 1 |
| $b \in$ *booleans* | 1 |
| $v$ (a variable) | $v$ |
| *if* | $\lambda\, a\, .\, \lambda\, b\, .\, \lambda\, c\, .\, a$ **and** $(b$ **or** $c)$ |
| +, -, *, /, etc. | $\lambda\, a\, .\, \lambda\, b\, .\, a$ **and** $b$ (all are strict functions) |
| $\lambda\, x\, .\, e$ | $\lambda\, x\, .\, abs(e)$ |
| $f$ (user-defined) | $f$ |

Thus, the two functions

$$f = \lambda\, x^{Int}\, .\, \lambda\, y^{Int}\, .\, if\ (=\, x\ 0)\ y\ (f\ (-\, x\ 1)\ y)$$
$$g = \lambda\, x^{Int}\, .\, \lambda\, y^{Int}\, .\, \lambda\, z^{Int}\, .\, if\ (=\, x\ 0)\ (+\, y\ z)\ (+\, y\ x)$$

become, in translation,

$$\begin{aligned} abs(f) &= abs(\lambda\, x^{Int}\, .\, \lambda\, y^{Int}\, .\, if\ (=\, x\ 0)\ y\ (f\ (-\, x\ 1)\ y)) \\ &= \lambda\, x^2\, .\, abs(\lambda\, y^{Int}\, .\, if\ (=\, x\ 0)\ y\ (f\ (-\, x\ 1)\ y)) \\ &= \lambda\, x^2\, .\, \lambda\, y^2\, .\, abs(if\ (=\, x\ 0)\ y\ (f\ (-\, x\ 1)\ y)) \\ &\ \ \vdots \\ &= \lambda\, x^2\, .\, \lambda\, y^2\, .\, x \textbf{ and } (y \textbf{ or } abs(f)\ x\ y) \end{aligned}$$

$$\begin{aligned} abs(g) &= abs(\lambda\, x^{Int}\, .\, \lambda\, y^{Int}\, .\, \lambda\, z^{Int}\, .\, if\ (=\, x\ 0)\ (+\, y\ z)\ (+\, y\ x)) \\ &= \lambda\, x^{Int}\, .\, \lambda\, y^{Int}\, .\, \lambda\, z^{Int}\, .\, abs(if\ (=\, x\ 0)\ (+\, y\ z)\ (+\, y\ x)) \\ &\ \ \vdots \\ &= \lambda\, x^2\, .\, \lambda\, y^2\, .\, \lambda\, z^2\, .\, z \textbf{ and } ((y \textbf{ and } z) \textbf{ or } (y \textbf{ and } x)) \end{aligned}$$

The abstraction process is not complete, since the fixpoints of $f$ and $g$ must be found. Since $f^a : 2 \to 2 \to 2$ ($f^a = abs(f)$), the computation of the AKC for $f^a$ is performed as follows:

$$f^a_0 = \lambda\, x^2\, .\, \lambda\, y^2\, .\, 0$$

$$\begin{aligned} f^a_1 &= \lambda\, x^2\, .\, \lambda\, y^2\, .\, x \textbf{ and } (y \textbf{ or } f^a_0\ x\ y) \\ &= \lambda\, x^2\, .\, \lambda\, y^2\, .\, x \textbf{ and } (y \textbf{ or } 0) \\ &= \lambda\, x^2\, .\, \lambda\, y^2\, .\, x \textbf{ and } y \end{aligned}$$

$$\begin{aligned} f^a_2 &= \lambda\, x^2\, .\, \lambda\, y^2\, .\, x \textbf{ and } (y \textbf{ or } f^a_1\ x\ y) \\ &= \lambda\, x^2\, .\, \lambda\, y^2\, .\, x \textbf{ and } (y \textbf{ or } (x \textbf{ and } y)) \\ &= \lambda\, x^2\, .\, \lambda\, y^2\, .\, x \textbf{ and } y \end{aligned}$$

Therefore $f^a = \lambda\, x^2\, .\, \lambda\, y^2\, .\, x$ **and** $y$. Also $g^a = abs(g)$, since $g$ is not recursive. Note that the fixpoint of $f$ was found by textual manipulation of the function definition, and not by the frontiers algorithm or any simplification thereof (this will be done later on). Now that $f^a$ and $g^a$ are known, they can be analyzed — the second step in the process of abstract interpretation.

We compute $f^a$ and $g^a$ on their respective argument lattices to find out that

$f^a\ 1\ 0 = 0$
$f^a\ 0\ 1 = 0$

$g^a\ 0\ 1\ 1 = 0$
$g^a\ 1\ 0\ 1 = 0$
$g^a\ 1\ 1\ 0 = 1$

Note that, by the monotonicity of $g$, if $g^a\ 1\ 0\ 1 = 0$ then $g^a\ x\ 0\ y = 0$, for any $x$ and $y$.

As the third part in the abstraction process, the information gleaned can be used with the original functions $f$ and $g$ to determine that $f$ does not terminate whenever either of its arguments is not terminating, and that $g$ does not terminate when any of its first two arguments are non-terminating. Thus $f$ is strict in both of its arguments, and $g$ is strict in its first two arguments.

# Chapter 3

# Implementation

In this chapter, the abstract interpretation implementation is discussed. First, the objectives of the implementation are presented, then an explanation of the various routines used is given, and finally an example of the implementation in operation is presented.

## 3.1. Objectives

The objectives of this implementation are threefold. First, the implementation must be generalized; it must be able to handle more instances of abstract interpretation than previous implementations. Second, the abstract interpretation must be performed in a clear, straightforward, and efficient manner. Third, it must perform abstract interpretation automatically, requiring minimal interaction with the user. These objectives, and the extent to which they have been achieved, will now be discussed.

### 3.1.1. Generality

This implementation meets the generality criterion in several ways. First is its ability to work with arbitrarily-structured concrete base lattices. Previous implementations have not, in general, incorporated this ability, because of the theoretical difficulties involved. However, as is usual in abstract interpretation, an informal justification is found adequate

to enable practical progress to be made, while waiting for a formalized theoretical basis to catch up.

The relaxation of restrictions upon the concrete base lattices allows, in particular, non-flat base lattices to be used. This means that functional languages which employ the list data structure and other higher-level data-structures (the majority of functional languages by far) are now eligible to undergo the abstract interpretation process. This increase in acceptance is a key ingredient if abstract interpretation is to be a tool in practical functional program analysis. Such a relaxation was made possible in practice because a representation of the concrete lattice was not required, and because of the automated and modular means of program translation employed. Nowhere is the concrete lattice explicitly represented, thus enabling all restrictions other than completeness upon the type of base lattices used to be removed. Also, since the method of translation from functions over the concrete lattice to functions over the abstract lattice is modular and automated, the non-flat structure of the concrete base lattices is possible, as long as the user can translate from the concrete base lattices to the abstract base lattices.

Secondly, this implementation is more general in its acceptance of higher-order functions. Other implementations can only process first-order functions. This is a result of the particular lattice, or type of lattice, used in the abstraction process. However, since this implementation has few restrictions in that area, the restriction to first-order functions is unnecessary. Also, the theoretical basis for abstracting higher-order functions has not always existed, thus the lack of practical implementations incorporating this ability. However, since [Burn, Hankin & Abramsky, 1986], the basis for performing abstract interpretation on higher-order functions is as sound as that for first-order functions. The only reason against abstracting higher-order functions is the complexity of such an endeavor. Algorithms such as the frontiers algorithm help in alleviating this problem.

A third generalization in this implementation is its treatment of abstract base lattices. Historically, these lattices have been flat and finite. However, in this implementation the former restriction was not necessary. The abstract base lattices must still be finite, because of the need for an explicit representation in the implementation, and also to guarantee termination of the development of the AKC in the fixpoint finding process.

### 3.1.2. Simplicity and efficiency

The qualities of simplicity and efficiency in this implementation are not independent but interacting. This is demonstrated in the modification of the frontiers algorithm used to find fixpoints. The original frontiers algorithm presented in [Martin & Hankin, 1987] is defined for first-order functions of multiple arguments over the abstract base lattice *2*. This algorithm is extended, first to deal with arbitrary base lattices, and second, to deal with higher-order functions. The result is an increasingly complicated algorithm. To rectify this, our approach is to view the problem of finding fixpoints by remembering from the start the desired applicability to arbitrary-order single argument functions defined over arbitrary finite base lattices. When this information is combined with the frontiers algorithm, a similar algorithm arises, which is simpler and easier to implement. The efficiency of the original algorithm has not been changed; the new algorithm eliminates unnecessary function evaluations in exactly the same way as before.

Instead of a function being represented by frontiers in a lattice composed as the product of argument lattices, it is represented by an annotated lattice (no products are necessary because the function has only a single argument). This annotated lattice is a normal lattice except that each element is annotated, potentially by another annotated lattice. The annotation in this representation is the function's value at that point, or a constraint

upon its value. The modified frontiers algorithm and the concept and usage of annotated lattices shall be explained in more detail later on in this chapter.

A second way in which the implementation is simple is in its division of the abstract interpretation process into modular and independent parts, as shown in Figure 3.1. After the user has specified certain translation and reduction rules, he gives the concrete function specifications to the implementation and observes the abstraction process take place in three parts. First, the functional specification is analyzed for purposes of internal representation. Second, this internal representation is simplified and translated to a function specification over the abstract lattice. Third, the least fixpoint of the abstract function specification is found. Underlying the latter two modules of the system is a fourth module, which is a collection of routines for the management of the various lattices being processed. This modularization of the implementation aids in its description and in the implementation process.

### 3.1.3. Automaticity

The third aim of this implementation is automaticity. That this has been achieved is demonstrated by the limiting of the user's interaction with the system to the steps of defining the abstract base lattice(s), the basic translation steps required (all other translation steps can be reduced to these), and the operational semantics of builtin functions upon the abstract lattice. On completion of these steps, the user inputs a set of function definitions, or specifications, and awaits the result. The output must then be interpreted by the user as the last step in the process (as shown in Figure 1.1).

Were this system to be integrated into a larger system, such as an optimizing compiler, and a definite abstract interpretation assigned, such as strictness analysis, then

the initial setup and the last step could also be automated. This would result in an automated abstract interpretation system, tailored for strictness analysis in an optimizing compiler.

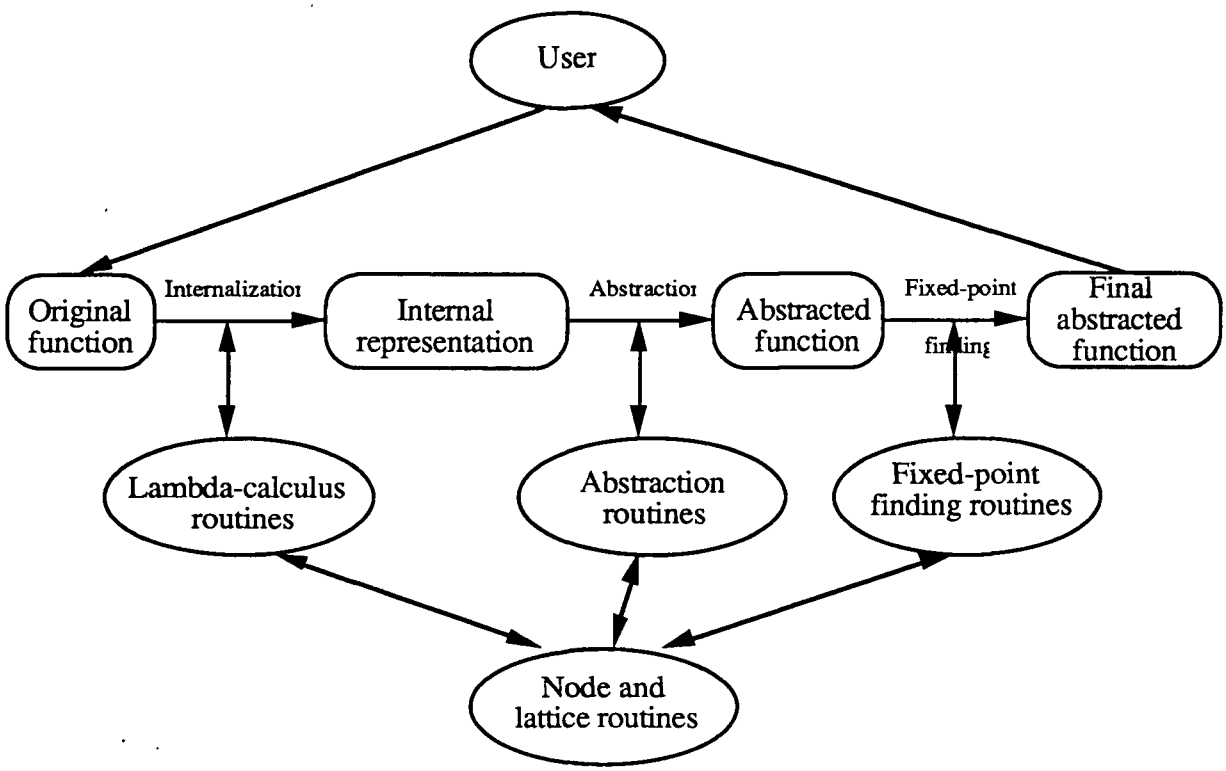



Figure 3.1. A pictorial description of the abstract interpretation system

## 3.2. Routines

This implementation has been modularized according to the routine's function. All routines for dealing with the λ-calculus representation of the functions are in one module, as are those for the management of lattices, the abstraction or translation process, and the fixpoint finder. These routines will be discussed in detail, with special attention paid to their role in fulfilling the implementation objectives.

### 3.2.1. Lambda-calculus routines

The λ-calculus is the language chosen to represent the functions defined over the concrete lattice. It is simple, yet powerful enough to express any function expressible by any other formal language. The routines associated with the λ-calculus can be divided into two categories: those which internalize a λ-calculus program, and those which reduce or interpret internalized programs.

#### 3.2.1.1. Internalization

The λ-calculus functional specification given by the user must be represented internally before it can be used by the system. This requires parsing the λ-calculus expression and creating an internal data-structure to represent the expression. To ensure parsing of the input λ-calculus expression, the user must adhere to the grammar for the language, shown below,

| | | |
|---|---|---|
| <λ-expression> | → | <primitive λ-expression> <λ-expression>* |
| <primitive λ-expression> | → | <constant> \| |
| | | <variable> \| |
| | | "λ" <variable> "." \| |
| | | "letrec" <assocs> ";" \| |
| | | "(" <λ-expression> ")" |
| <assocs> | → | <assoc> (";" <assoc>)* |
| <assoc> | → | <variable> "=" <λ-expression> |

where <variable> has the usual meaning, and <constant> is any integer or boolean. Notice in addition to the standard definition of λ-calculus expressions, that the "letrec" construct is included as a space saving mechanism to enable user-defined functions to be declared for later use. Its semantics is fully described in the next section.

To parse the incoming λ-calculus expression, the grammar has been re-expressed in a notation known as a Definite Clause Translation Grammar (DCTG). This notation [Abramson, 1984], along with an appropriate DCTG compiler and interpreter, allows the

grammar to be executed directly on the input text and to return a parse of the expression, if one exists.

The DCTGs used in this implementation allow the parsing to be divided into two parts, lexical analysis and token-based parsing, as is often the case in the design of language parsers. The second part of the process is easier to understand and implement if the lexical analysis is out of the way.

The output of the lexical analyzer consists of a series of atomic tokens, or lexemes. Suppose the input program is that shown in Figure 3.2. Then the lexemes output from the lexical analyzer are

[variable(mul), constant(2), variable(a)],

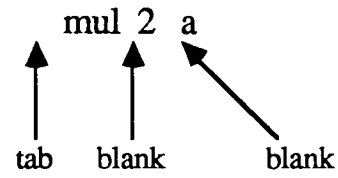



Figure 3.2. An example of an input program

where the tabs and blank spaces have been eliminated. Notice that "mul" is identified as a variable, even though it obviously is a builtin function call to multiply two numbers. This aspect of the parsing process is handled later, by the reduction routines.

A more accurate example of what the lexical analyzer would accept as input and produce as output is the following, where type notations are included throughout the program:

mul(int, (int, int)) 2 a(int).

The analysis of this typed program produces the following list of lexemes:

[v(mul, (int, (int, int))), c(2, int), v(a, int)].

This typing notation is similar to the notation defined previously, using type expressions for naming function types and lattices.

The second part of the parsing process is the analysis of the lexemes produced by the lexical analyzer. The mechanics are similar. A sequence of atomic symbols, lexemes this time instead of characters, is analyzed with respect to their structure and an internal representation is formed. The DCTG used for this purpose is like the grammar shown previously. As an example, the <primitive λ-expression> rules are shown in Figure 3.3.

```
prim_lexp ::= tCONST^^C
<:>
sem(Sem) ::- C^^sem(Sem).

prim_lexp ::= tVAR^^V
<:>
sem(Sem) ::- V^^sem(Sem).

prim_lexp ::= tLAMBDA, !, tVAR^^V, tDOT, lexp^^E
<:>
sem(l(Var,Exp)) ::- V^^sem(Var), E^^sem(Exp).

prim_lexp ::= tLETREC, !, list_of([assoc, tCOMMA])^^S, tSEMICOLON
<:>
sem(letrec(Assocs)) ::- S^^list(Assocs).

prim_lexp ::= tLP, lexp^^E, tRP
<:>
sem(paren(Sem)) ::- E^^sem(Sem).
```

Figure 3.3. The grammar rules for a primitive λ-expression

Each grammar rule represents an alternative construction in the parse tree. For example, the node "prim_lexp" branches downwards into 1, 3, or 4 child nodes as directed by the program text under analysis. In the "^^Var" notation, the prolog variable Var is identified with that node in the parse tree. If prim_lexp were expanded downwards to the nodes tLP, lexp, and tRP by the last rule, the "lexp" node would be named (in that

instantiation only) by the variable E. The node names are used in the second half of each rule, where a prolog-like semantic predicate is listed. These semantic predicates are interpreted in the same way as normal prolog predicates, except if a node-naming variable Var precedes a clause name C (in the format "Var^^C"). In which case, the semantic predicate C associated with the rule headed by the node identified with Var in the current parse tree is executed. This method of associating executable predicates with grammar rules allows an internal representation to be constructed directly from within the parse tree. As can be seen from the portion of the DCTG for the λ-calculus, the internal representation is a prolog-functor, formatted as follows:

| **Text** | **Internal** |
|---|---|
| C (a constant) | c(C, Type) (Type is deduced from C) |
| V(Type) | v(V, Type) |
| letrec V(Type) = E, ...; | letrec([(v(V, Type), E'), ...]) |
| λ V(Type) . E | l(v(V, Type), E') |
| E F | @(E', F') |

where E' and F' are the internal representations of E and F, respectively. Thus, the previous example program

mul(int, (int, int)) 2 a(int)

is represented internally as the functor

@(@(v(mul, (int, (int, int))), c(2, int)), v(a, int)).

In the unparenthesized λ-calculus program "*a b c*", the internally represented program is constructed as if the original program was actually "(*a b*) *c*". This is not a result of the parsing, but occurs later by a manipulation of the internal data structure. This is because the DCTG grammar formalism does not allow left-recursive grammar rules, which the desired above interpretation of "*a b c*" would require. Thus, the initial result of the parse of "*a b c*" is "*a* (*b c*)" and must be rearranged to get the desired result. This manipulation is simple and is only noted here to explain the reason for the "paren()" construction (used to

flag explicit parentheses) found in the semantic portion of the last rule in the DCTG fragment.

#### 3.2.1.2. Reduction

Interpretation of the λ-calculus is known as reduction, because a λ-calculus expression is "executed" by simplifying it, or reducing it, until it can be reduced no more. This reduction process is defined by a few simple rules, which serve to establish semantics for the language, in the same way that its syntax was established previously:

| **Expression/Environment** | **Reduction (1 step)/New Environment** |
|---|---|
| Var / Env | lookup(Var,Env) / Env |
| λ V . E / Env | λ V . E' / Env |
| λ V . E F / Env | E / Env + assoc(V,F) |
| *if* E F G / Env | *if* E' F G / Env |
| *if true* F G / Env | F / Env |
| *if false* F G / Env | G / Env |
| *f* E / Env | builtin(*f*, E, Env) / Env |
| E F / Env | E' F / Env |
| E' F / Env | E' F' / Env |
| letrec(Assocs) E / Env | E / Env + Assocs |

where E, F, and G are λ-calculus expressions and E', F', and G' are their reductions, respectively. The symbol *f* denotes a builtin function other than *if*. These rules of reduction are standard except for the letrec construction which is reduced by adding the function definitions to the environment.

An extension to the reduction rules which has not been mentioned allows a function represented as an annotated lattice (as opposed to the usual internal functor representation) to be applied to an argument. Thus, for example, if the function *not* over the concrete base lattice of *booleans* in Figure 2.7 was represented as an annotated lattice, like Figure 3.4, then its internal representation could be applied to a value, say *false*, and be reduced to get the value *true*. This rule enables the fixpoint finding process to be expressed more simply than otherwise.

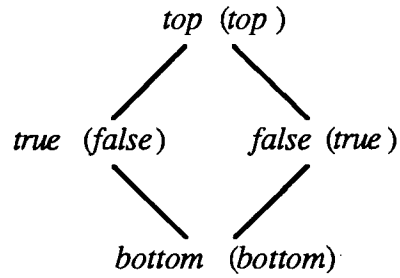



Figure 3.4. The function *not* represented as an annotated lattice

As an example of how reduction proceeds, consider the step-by-step reduction of

letrec *double* = λ *x* . * 2 *x*,
*zero* = λ *x* . *if* (= *x* (*double x*)) *yes no*;

*zero* 1,

which defines a function *zero* that produces *yes* if its argument is 0, and *no* otherwise:

| **Expression** | **Environment** |
|---|---|
| letrec ...; *zero* 1 | { } |
| *zero* 1 | {...} (contents of above letrec) |
| (λ *x* . *if* (= *x* (*double x*)) *yes no*) 1 | {...} |
| *if* (= 1 (*double* 1)) *yes no* | {...} |
| *if* (= 1 ((λ *x* . * 2 *x*) 1)) *yes no* | {...} |
| *if* (= 1 (* 2 1)) *yes no* | {...} |
| *if* (= 1 2) *yes no* | {...} |
| *if false yes no* | {...} |
| *no* | {...} |

The type information in the previous example and description of the reduction process has been omitted in order to simplify the discussion.

Normal order reduction (leftmost-outermost) has been employed throughout this description and the implementation in order to preserve lazy evaluation. However, to make the reduction process more efficient, sometimes parts of the internal representation are flagged to indicate they can be reduced no farther. This does not change the reduction order but helps to avoid re-examining already-reduced sub-expressions.

## 3.2.2. Node and lattice manipulation routines

Necessary to the implementation are routines which perform various operations on, and within, the abstract interpretation lattices. These routines look up nodes inside lattices, add new nodes to lattices, generate new lattices from old, and do other operations required by the implementation. They are categorized according to whether they deal with nodes or lattices.

### 3.2.2.1. Nodes

The routines that deal with nodes are primarily those that create nodes, compare nodes, and perform operations upon nodes.

#### 3.2.2.1.1. Representation

Every node in an annotated lattice has four properties. The first property is its name. This can be assigned by the user, as in his initial definition of the abstract base lattice(s), or by the implementation itself during the automatic creation of new nodes. The name of any node must be unique within its surrounding lattice.

The second and third properties of every node are the nodes above and below the node in the containing lattice. These sets may be empty, but they must always exist and be associated with the node. They are necessary for the correct determination of, and for correct inference based on, the lattice's internal structure.

The final property associated with each node is specific to the implementation. It is the "annotation" or note that is present with each node. This annotation may serve any purpose in general, but in the implementation it is restricted to being the value of a certain function evaluated on that node, where the function is determined by context. Thus, given $f : L \rightarrow L$, each node $n$ contained in $L$ could be annotated with the value $f(n)$, giving an explicit representation of $f$ over $L$ as an annotated lattice.

For example, consider $g : A \rightarrow B$, where $g$, $A$, and $B$ are as shown in Figure 3.5. The four nodes comprising lattice $A$, annotated with their values in $B$ as determined by $g$, are represented in this implementation as follows:

```
node(a, {}, {b, c, d}, 3),
node(b, {a}, {d}, 2),
node(c, {a}, {d}, 2),
node(d, {a, b, c), {}, 1).
```

This representation is not minimal. For example, since the structure of the lattice has been defined and node $a$ is above node $b$, and node $b$ is above node $d$, it is unnecessary to record in the definition of node $a$ that below it are the nodes $b$, $c$, and $d$. That $d$ is below $a$ could be deduced from other node definitions. Thus, the inclusion of node $d$ in the below set for node $a$ is redundant, but it has been added for reasons of efficiency. At this level of data-structuring it is efficient to trade an increase in space for a decrease in time.

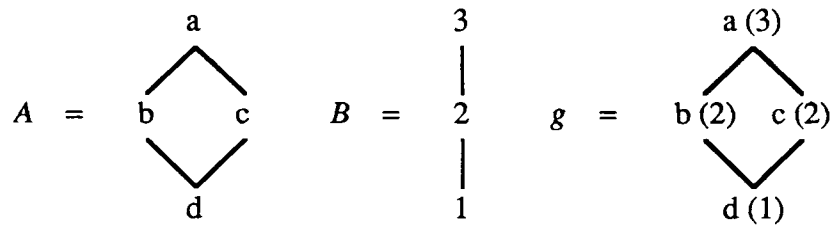



Figure 3.5. The function $g : A \rightarrow B$ represented as an annotated lattice

### 3.2.2.1.2. Routines

The routines that manupulate node structures fall into three categories: predicates, node-valued functions, and node creation.

The predicates which are defined over nodes are the usual binary comparison operators: =, <, and ≤. The first is based on comparison of node names. The second predicate is implemented as a search for the first node's name in the second node's below

set. If the search is successful, then the comparison is true, otherwise it is false. The last predicate is defined as a combination of the first two.

The node-valued functions defined are the binary functions *greatest lower bound* (*glb*) and *least upper bound* (*lub*). The *glb* function is implemented as a search for the maximum (highest, in a diagram) node in the intersection of the below sets of the two nodes given as arguments. The *lub* function is defined in a similar but opposite fashion. These functions also take sets of nodes as arguments. In that case, for example, $glb\ \{a, b, c\}$ is defined as $glb\ \{a, glb\ \{b, glb\ \{c\}\}\}$, where $glb\ \{n\} = n$ for any node $n$.

To create a node ready for inclusion into an existing lattice structure, its name and intended position in the lattice are required. This can be fulfilled by producing a pair of complete above and below sets for the node, but this is a tedious and unnecessary chore. Since the node is created only once, it is simpler to give the node creation routine a pair of minimal above and below sets, and let it deduce the complete above and below sets automatically. Thus the node creation routine is an enumeration mechanism which expands minimal above and below sets to their redundant but complete counterparts. For example, if the above set for a node $n$ is denoted $above(n)$, then the minimal above set M would cause to be generated a complete above set through the computation of

$$\mathrm{M} \cup \left( \bigcup_{n \in \mathrm{M}} above(n) \right)$$

### 3.2.2.2. Lattices

The routines in the implementation that deal with lattices also deal with nodes, but only as part of the greater lattice structure. These routines perform such functions as adding nodes to lattices, deleting nodes from lattices, finding the bottom of a lattice, and minimizing lattices.

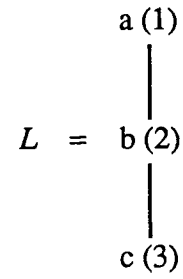
#### 3.2.2.2.1. Representation

Since the nodes themselves contain a description of what is above and below them in the lattice, the lattice structure itself need be no more than a mere set of nodes. However, this type of lattice structure presents an efficiency problem. Since the purpose of a lattice data-structure is to group a set of nodes for later retrieval and analysis, those nodes should be organized for the efficient performance of such tasks. Thus the organizational structure chosen for the nodes is that of the sorted binary tree.

A sorted binary tree is a binary tree with the added restriction on every node $n$ that the names of $left(n)$, $n$, and $right(n)$ be ordered alphabetically (in increasing order, in this case), where $left(n)$ and $right(n)$ indicate the left and right children of $n$, respectively. Also, each node in the tree contains the size of the tree it heads. This information helps in keeping the sorted binary tree balanced. Thus a lattice is stored as a sorted binary tree, each node in the tree also being the root of a sorted binary tree, where a node is a five element functor. The first element is the key by which the tree is sorted. In the implementation, this is the name of the lattice node stored there. The second element of the functor is the actual lattice node structure, and the third element is the size of the tree rooted at that node. The fourth and fifth elements are the structures denoting the left and right subtrees, respectively. An example of this lattice structure for a three element lattice can be seen in Figure 3.6, where the empty sorted binary tree is denoted by []. Notice that the maximum length of time to access an element of a lattice is reduced from $O(n)$ to $O(log\ n)$ using a sorted binary tree as opposed to a set for storing the lattice nodes, assuming that the tree is balanced.

#### 3.2.2.2.2. Routines

There are three types of lattice routines in the implementation. They deal with either a single node within a lattice, all nodes in a lattice, or they create new lattices from old.




```
L =    tree(   b,
               node(b, {a}, {c}, 2),
               size(3),
               tree(a, node(a, { }, {b, c}, 1), size(1), [], []),
               tree(c, node(c, {a, b}, { }, 3), size(1), [], []))
```

Figure 3.6. An annotated lattice *L* and its internal representation

Among the routines concerned with individual lattice nodes are those which add nodes to, and delete nodes from, a given lattice. The addition is simple, since the lattice is a sorted binary tree. The tree is traversed down to the appropriate empty leaf node, which is then replaced with a new node that has two empty leaf nodes as subtrees.

Deletion of nodes from a lattice is more complex. If the node is a leaf node, then it may be removed and replaced by an empty leaf node. If the node has only one child node, then the node to be deleted may be removed and the child node may take its place. But if the node has two non-empty subtrees as children, then these subtrees must be merged together to form a single sorted binary tree, and balanced. This newly balanced tree then takes the place of the node to be deleted. The routine which does this takes into account the size of the two trees to be merged. It adds the smaller tree bit by bit onto the larger tree to produce a new sorted, balanced, binary tree.

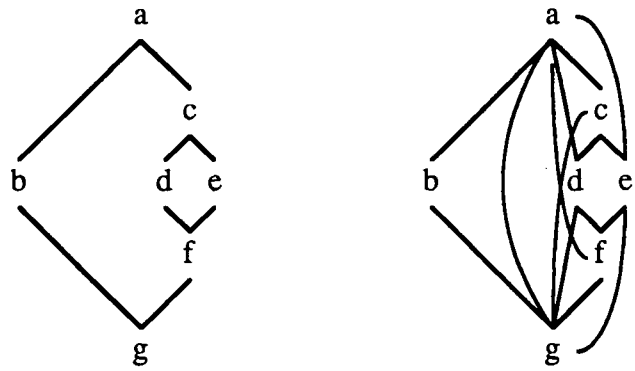
Another routine concerned with single nodes looks up nodes by name. Given the name of a node, and its containing lattice, this routine searches the lattice for the node's name, and on finding it, returns the node structure containing all information about that node.

A further such routine finds the bottom of a lattice. This node is guaranteed to exist, so the routine is implemented as a search. To find the bottom of the lattice, the routine checks whether or not the below set for a node is empty. If so, then that node is the bottom (if another node exists which satisfies this condition, then the greatest lower bound of the two nodes would not exist, contradicting the definition of a lattice).

Among the routines which deal with multiple nodes is one which takes a full lattice and computes from it a minimal lattice. This routine takes as input a lattice consisting of nodes which all have complete above and below sets (a normal lattice) and returns the lattice modified so that each node's above and below set is reduced to the minimal number of elements necessary to still be able to deduce the lattice's structure. This routine is useful in producing a user readable representation of a lattice, since the usual pictorial representation of a lattice is minimal as opposed to full. See Figure 3.7 for an example of the two kinds of lattice representation.

Another routine which deals with lattices is the empty lattice routine. This returns an empty lattice, to which nodes can be added as desired.

The last type of lattice routine to be discussed is that which generates new lattices from old. The new lattice generated by this routine is a function lattice. Given two lattices $A$ and $B$, this routine creates $[A \rightarrow B]$, the lattice of all continuous functions mapping $A$ to $B$. But, in the case of finite lattices, continuous is the same as monotonic. Thus, the creation of $[A \rightarrow B]$ involves the enumeration, and subsequent ordering, of all monotonic functions $f: A \rightarrow B$. This enumeration is guaranteed to terminate, since if the monotonicity condition is ignored, then the number of different functions possible is $|A|^{|B|}$, which is finite since both $A$ and $B$ are finite. When the monotonicity condition, and thus the lattice structure, is considered, then the number $|A|^{|B|}$ is simply an upper bound on the size of $[A \rightarrow B]$.




Minimal Full

Figure 3.7. A comparison of two lattice representations

To generate $[A \rightarrow B]$, first all monotonic functions mapping $A$ to $B$ are discovered. These are then ordered into a lattice structure (according to the usual definition of order among functions). The hard part in this is the discovery of all monotonic functions mapping $A$ to $B$. This is accomplished as follows: Since a monotonic function $f : A \rightarrow B$ is an annotation of $A$ with values from $B$ (with the condition that $x \leq y$ in $A \Rightarrow fx \leq fy$ in $B$), then a copy of $A$ can be made and annotated, and can be considered as one of the functions in $[A \rightarrow B]$. These functions can be generated iteratively, starting with a known function. This initial function is the bottom of $[A \rightarrow B]$, the function which assigns to all elements in $A$ the bottom of $B$. Given this function, another function can be generated by assigning to any element $a \in A$ a value $b \in B$ and ensuring that for all nodes $n \in A$ such that $n \geq a$, the value assigned to them, $v$, satisfies $v \geq b$. This can be done easily in prolog, using its natural backtracking mechanism. Nodes are annotated starting from the bottom of $A$, moving upwards one node at a time, and always retaining the minimum value assignable to each new node. Thus, due to monotonicity, whenever a new node in $A$ is considered for annotation, there is available only a subset of values in $B$. The process is continued until the entire lattice is annotated, at which point the new function thus generated is recorded

and backtracking takes place. Once the backtracking is finished, all functions have been generated and there remains only the process of ordering them into a lattice structure, giving the resulting function lattice $[A \rightarrow B]$.

### 3.2.3. Abstraction routines

The abstraction routines perform the first part of the abstraction process: the translation. The translation is automated, leaving only the specification of a few primitive translations and builtin functions to the user. Once the user has made these specifications, the rest of the translation is based upon them. This makes the abstraction mechanism less general than theoretically allowable, but in practice this is not a restriction.

The translation routines fall into two categories, depending upon the type of data being translated: either a λ-calculus expression, or a type expression.

#### 3.2.3.1. Lambda-calculus expressions

The translation of λ-calculus expressions is performed recursively, with abstractions and applications carried through the translation intact, and only the more primitive elements of the expression being changed. This automated part of the translation can be summarized as follows:

| **Expression** | **Abstraction** |
|---|---|
| E F | *abs*(E) *abs*(F) |
| λ V . E | λ *abs*(V) . *abs*(E) |
| letrec V = E, ...; | letrec *abs*(V) = *abs*(E), ...; |

The translation of constants, variable names, and function names remains to be specified. These translations are supplied by the user.

#### 3.2.3.2. Type expressions

The translation of types must occur for abstraction to remain consistent, and for the fixpoint finding routines to operate automatically. To translate types automatically, the

implementation requires the user to specify only the translations of the base lattices. This is because the specification of concrete and abstract lattices is formalized (each being based on a sum), the key ingredients of which are the base lattices. For example, if the concrete base lattices are *integers* and *booleans*, then these may both map into the single type *2*, the abstract base lattice. If the concrete base lattices consist of *integers*, *booleans*, and *lists*, these may be mapped as follows:

| **Concrete Type** | **Abstract Type** |
|---|---|
| *integers* | *2* |
| *booleans* | *2* |
| *lists* | *4* |

where *4* is an abstract base lattice designed to produce abstraction information about lists (*4* is an actual lattice used in an application and will be described later).

Given the user-supplied specification of how translation is to proceed on base types, the translation of a type expression is performed according to its structure, as follows:

| **Expression** | **Abstraction** |
|---|---|
| $a \rightarrow b$ | $abs(a) \rightarrow abs(b)$ |
| $a$ | $abs(a)$ (as specified by the user) |

### 3.2.4. Fixpoint finding routines

Finding the least fixpoint of a function, or set of functions, requires the iteration and evaluation of successive approximations to the function. This proceeds until a limit point is reached. For example, if $f : A \rightarrow B$ is

$$f = body\_of\_f,$$

and $f$ is recursive, then one can define $F : [A \rightarrow B] \rightarrow B$ as

$$F = \lambda f \,.\, body\_of\_f$$

and follow the AKC to its limit by the successive computation of the following functions:

$F_0 = bottom_B$
$F_1 = F\ F_0$
$F_2 = F\ F_1$
etc.

until $F_i = F_{i+1}$ and thus $F_i = F\ F_{i-1}$ is the least fixpoint of $f$. The problem occurs in the determination of $F_i = F_{i+1}$. This is solved in the implementation by using an explicit representation of functions as annotated lattices, thus making comparison easy but generation expensive. Also used is a modification and simplification of the frontiers algorithm.

### 3.2.4.1. Frontiers algorithm modification

To represent a function $f : A \rightarrow B$ explicitly, first the lattices $A$ and $B$ must themselves be explicitly represented. For example, if $f$ is of type *integers* $\rightarrow$ *integers* $\rightarrow$ *integers* (as are the usual binary mathematical operators), then $f$ maps *integers* to *integers* $\rightarrow$ *integers* and the explicit representation of $f$ is the lattice *integers* annotated with values from [*integers* $\rightarrow$ *integers*]. Thus the [*integers* $\rightarrow$ *integers*] lattice must be constructed using the lattice routines described previously. Once all the lattices concerned have been constructed, annotation can proceed, using a modification of the frontiers algorithm, as follows.

A node $n$ in the argument lattice $A$ is selected and $f$ is evaluated on $n$, enabling $n$ to be annotated with the value of $f(n)$. Since $f$ is monotonic, all points in $A$ above $n$ are annotated indicating their value $v$ must satisfy $v \geq n$, and all points in $A$ below $n$ are annotated complementarily. This process is repeated iteratively, so that all nodes in $A$ are either annotated with an actual function value or have the annotation *between*$(a, b)$ to indicate that their future value $v$ must satisfy $a \leq v \leq b$. Obviously then, whenever any node is constrained so far as to have the annotation *between*$(a, a)$, the annotation is replaced with $a$ as the only value possible (relieving $f$ of ever having to be evaluated at that node).

This is continued until the lattice is completely annotated, giving the complete explicit representation of $f$. If $f$ is typed so that the annotations to $A$ are functions themselves, as in *integers* $\rightarrow$ (*integers* $\rightarrow$ *integers*), then the annotations are made not as $\lambda$-calculus expressions, but as further annotated lattices. This property enables all functions to be considered as single-argument functions, and simplifies the algorithm over the original frontiers algorithm. Also, the simplification of multiple frontiers into a simple constraint allows the algorithm to be used in a natural manner on arbitrary finite lattices.

#### 3.2.4.2. Heuristics

One question that remains pertains to the nodes that are chosen for function evaluation. That these choices make a difference to efficiency can be illustrated by Figure 3.8. It increases efficiency to choose a heuristic which selects nodes in an optimal order, thus avoiding as many expensive function evaluations as possible. But, such an optimal heuristic has not been discovered in this implementation. However, it is possible to produce some heuristics which guide the node evaluation process in a more efficient manner than otherwise.

One such rule requires the node selection to be ordered by the number of arcs associated with each node, from most to least. Another rule orders the nodes by the number of arcs associated with each node in the minimal lattice representation. Yet another rule is to order the nodes by their distance from other already evaluated nodes. Other heuristics are possible. The rule used in this implementation is the second rule mentioned above, namely, that nodes are selected according to the number of arcs connecting them to other nodes in the minimal lattice representation.

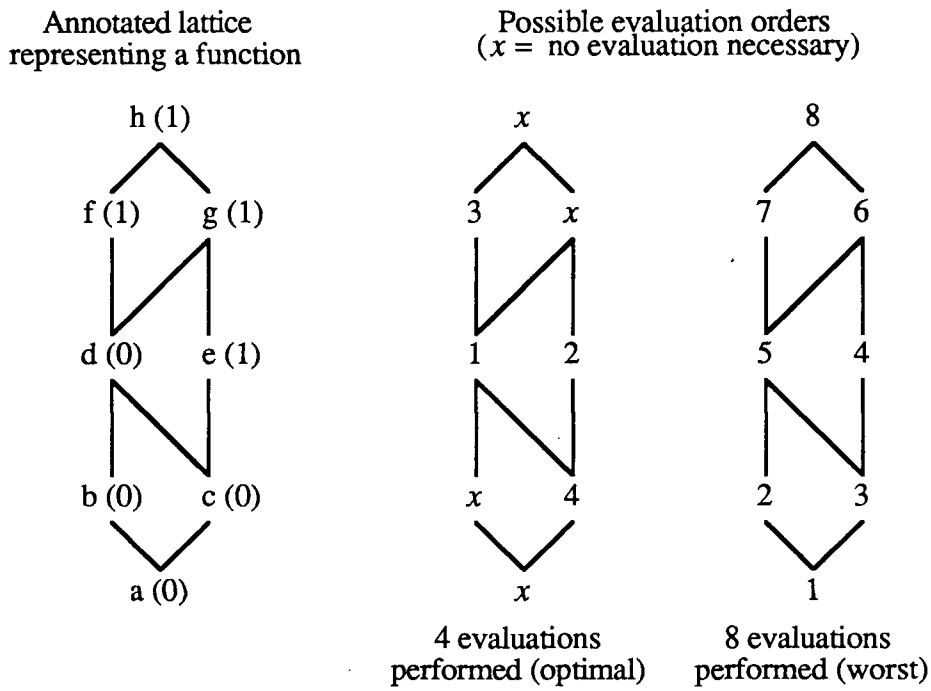



Figure 3.8. A comparison of two possible evaluation orders

## 3.3. Example

As an example of how the various implementation routines work together to perform abstract interpretation, consider $f : C \rightarrow A$, defined as

$$f = \lambda\, v\, .\ \textbf{or}\ v\, f\, (not\ v),$$

a complicated way of expressing the value *true*. The concrete base lattice for the above function need only consist of *booleans*, and the abstract base lattice is 2. The function as described so far is illegal input to the system, since it uses a typed λ-calculus. The actual input to the system is

```
letrec f(bool, bool) =
        λ v(bool) . or(bool, (bool, bool)) v(bool) f(bool, bool) (not(bool, bool) v(bool));
```

This is translated by the lexical analyzer into the following list of lexemes:

[letrec, v(f, (bool, bool)), punct(eq), λ, v(v, bool), punct(dot), ...]

This list is parsed into an internal representation of the λ-calculus expression, consisting of the following (types left out for clarity):

letrec([(f, l(v, @(@(or, v), @(f, @(not, v))))])

This λ-calculus expression is then abstracted, assuming that the user has supplied appropriate definitions for primitive types, constants, and builtin functions. In this case, these definitions are assumed:

| **Expression** | **Abstraction** |
|---|---|
| bool | $2$ ($2 = \{0, 1\}$, where $0 \leq 1$) |
| constant | 1 |
| *not v* | *not v* (considering 0 and 1 as truth values) |
| **or** *x y* | **or** *x y* (considering 0 and 1 as truth values) |

Thus, f is translated into

letrec([($f^a$, l($v$, @(@(**or**, $v$), @($f^a$, @(*not*, $v$))))])

or, equivalently,

letrec $f^{2 \to 2} = \lambda\, v^2$ . $\mathbf{or}^{2 \to 2 \to 2}\, v^2\, f^{2 \to 2}\, (not^{2 \to 2}\, v^2)$

where the types are super-scripted for clarity. This expression would never terminate if evaluated, and thus the fixpoint finding mechanism must be applied. The first approximation to $f$ is denoted $f_0$, and is the function found at the bottom of the function lattice $[2 \to 2]$

$$f_0 = \lambda\, v^2 \,.\, 0$$

Carrying through the expansion of the AKC, the rest of the approximations are

$$\begin{aligned} f_1 &= \lambda\, v \,.\, \mathbf{or}\, v\, f_0\, (not\ v) \\ &= \lambda\, v \,.\, \mathbf{or}\, v\, 0 \\ &= \lambda\, v \,.\, v \end{aligned}$$

$$\begin{aligned} f_2 &= \lambda\, v \,.\, \mathbf{or}\, v\, f_1\, (not\ v) \\ &= \lambda\, v \,.\, \mathbf{or}\, v\, (not\ v) \\ &= \lambda\, v \,.\, 1 \end{aligned}$$

$$f_3 = \lambda v \,.\, \textbf{or}\ v\ f_2\ (\textit{not}\ v) = \lambda v \,.\, \textbf{or}\ v\ 1 = \lambda v \,.\, 1$$

giving the limit, or least fixpoint, of $f$ as $f_2 = \lambda v \,.\, 1$. This is done by the implementation using a different representation for $f$, as follows:

$$f_0 = \lambda v^2 \,.\, \begin{matrix} 1\,(0) \\ \mid \\ 0\,(0) \end{matrix}$$

$$f_1 = \lambda v \,.\, \textbf{or}\ v\ f_0\ (\textit{not}\ v) = \begin{matrix} 1\,(1) \\ \mid \\ 0\,(0) \end{matrix}$$

$$f_2 = \lambda v \,.\, \textbf{or}\ v\ f_1\ (\textit{not}\ v) = \begin{matrix} 1\,(1) \\ \mid \\ 0\,(1) \end{matrix}$$

$$f_3 = \lambda v \,.\, \textbf{or}\ v\ f_2\ (\textit{not}\ v) = \begin{matrix} 1\,(1) \\ \mid \\ 0\,(1) \end{matrix}$$

If $f$ returned a function as a value (like the **or** function), then instead of having elements of *2* as annotations, it would have further annotated lattices as annotations, as shown in Figure 3.9.

Once the least fixpoint of $f$ is found, abstract interpretation is complete, and the user can use the information thus derived to infer properties of $f$. If, for example, the application was strictness analysis, then from the fixpoint definition of $f$, $f$ evaluated on any argument is 1, so $f$ is not strict at all. Thus, to evaluate $f$ and its argument concurrently would be to take the chance of performing unnecessary (and possibly non-terminating) computations.

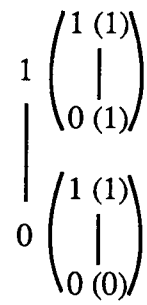



Figure 3.9. The **or** function represented as an annotated lattice

# Chapter 4

# Applications

Abstract interpretation was initially proposed as a unified model of program analysis for solving problems including global data-flow analysis, type-checking, program performance analysis, and program correctness analysis (see [Cousot & Cousot, 1977]). Recently, the trend of practical abstract interpretation has been to restrict itself to less and less general problems, namely those on which it can perform successfully and in an efficient manner. The trend of theoretical abstract interpretation has been to strengthen the foundations of a more generalized process than that which exists. Currently, the theoretical basis is strong enough to support some practical applications, not the myriads of applications envisioned by the Cousots, but those such as program termination analysis [Mycroft, 1980], data-type analysis [Mishra & Keller, 1984], and many types of strictness analysis [Abramsky, 1985], [Burn, 1987], [Wadler & Hughes, 1987]. Abstract interpretation is not being applied to some of the other problems that the Cousots described, because they are more effectively solved by another model of program analysis, because the theoretical foundations have not yet been sufficiently developed, or because they have been eliminated by the functional framework within which abstract interpretation is performed.

Two problems which are still effectively dealt with by abstract interpretation are strictness analysis and program termination analysis. These problems, and the application

of the implementation to their solution, will now be discussed in depth. Actual output of the system when given the following examples can be found in appendix A.

## 4.1. Strictness Analysis

The most common use to which abstract interpretation has been put is strictness analysis. A function $f$ with argument of type $T$ (where $T$ is assumed to also be the concrete base lattice for that type), is strict if and only if *abs(f) bottom = bottom*, where *bottom* is the bottom of the lattice *abs(T)*. Since the bottom of the type lattice indicates undefinedness, or non-termination, $f$ being strict means that $f$ is non-terminating whenever its argument is non-terminating.

For example, suppose the following two functions are defined:

letrec $f = \lambda\, x\, .\, *\ 2\ x$,
$g = \lambda\, x\, .\, \lambda\, y\, .\, \mathit{if}\ (=\ x\ 0)\ y\ x$;

Then $f$ is strict, since $f$ is non-terminating whenever $x$ is non-terminating. Also, $g$ is strict, and so is $g\ 0$. But $g\ 1$, which is derived as shown, is not strict:

$$\begin{aligned} g\ 1 &= (\lambda\, x\, .\, \lambda\, y\, .\, \mathit{if}\ (=\ x\ 0)\ y\ x)\ 1 \\ &= \lambda\, y\, .\, \mathit{if}\ (=\ 1\ 0)\ y\ 1 \\ &= \lambda\, y\, .\, \mathit{if\,false}\ y\ 1 \\ &= \lambda\, y\, .\, 1 \end{aligned}$$

As mentioned previously, strictness analysis is useful in dealing with parallelism in functional programs. Normally, since the mode of evaluation is lazy, parallel evaluations do not occur and the program is limited to sequential operation. But if one recalls that the motivation behind the lazy evaluation is to evaluate only what is necessary, then under certain conditions the sequential mode of evaluation can be relaxed. That is, one can evaluate two expressions simultaneously only if one knows that both expressions will have to be evaluated anyway. For example, in the expression "$*\ a\ b$", the expressions $a$ and $b$ can be evaluated simultaneously, rather than evaluating $a$ first and $b$ second, since it is

known that the multiplication function always evaluates, or needs, both of its arguments. Thus, the key concept is "need"; the function is strict if and only if it needs its argument.

Once a function or set of functions have been analyzed for strictness, it is a simple matter to annotate them for later compilation. This allows an optimizing compiler to take advantage of all of the parallelism existing in the program, thus rendering obsolete the necessity of any direct user control over which parts of the program are allowed to execute simultaneously and which parts are not. Exactly those expressions in a program which are safe to evaluate in parallel are evaluated thusly, and no others.

The lattice used for representing information in strictness analysis need only represent two conditions: non-termination and otherwise. Such a lattice is the two-element lattice *2*, with elements 0 and 1, where $0 \leq 1$. The 0-element represents definite non-termination, and the 1-element represents anything else. This is a natural way of defining an abstract base lattice for strictness analysis, as can be seen from the mapping for constants thus derived, shown in Figure 4.1.

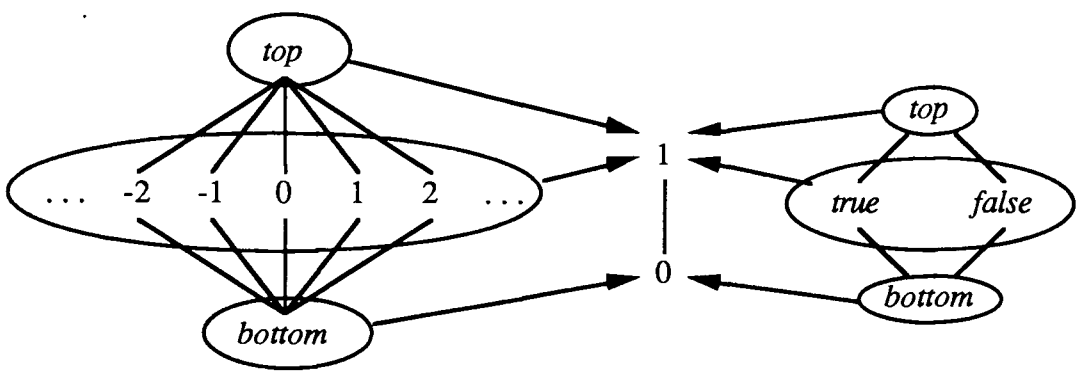
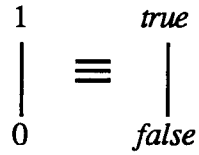



Figure 4.1. The mapping of constants into the abstract base lattice *2*

This definition of the abstract base lattice is also useful because of its equivalence with the two-element boolean lattice (not the concrete base lattice of *booleans*, shown in Figure 4.1), as shown below:




which means that instead of using the standard lattice functions *glb* and *lub* on *2*, one can instead use the functions **and** and **or**, respectively, where 1 is considered true and 0 considered false. Thus the abstractions for builtin functions, shown in Figure 4.2, are simplified.

| *a* | *b* | *abs(f)a b* |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

where *f* is one of +, –, *, /, etc.

| *a* | *b* | *c* | *abs(if)a b c* |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

Figure 4.2. Abstraction definitions for builtin functions

As can be seen from the tables in Figure 4.2, the arithmetic functions such as * can be expressed as

$$*^a = \lambda\, a\, .\, \lambda\, b\, .\, a \textbf{ and } b$$

while the *if* function can be simplified to

$$if^a = \lambda\, a\, .\, \lambda\, b\, .\, \lambda\, c\, .\, a \textbf{ and } (b \textbf{ or } c).$$

According to these definitions (and to intuition), the mathematical functions need all of their arguments, while the *if* function needs its first argument and one of the next two (more precisely, *if* is strict and so is one of *if a* and *if a b*).

So far, strictness analysis which deals with higher-level data structures such as lists has been ignored. This is because a different abstract base lattice which represents list

information is required. In the next section, the discussion will be restricted to the simpler case of strictness analysis without lists.

### 4.1.1. Simple strictness analysis

The process of strictness analysis can now be performed on functions not involving lists, using what has been defined above. For example, consider

letrec $f = \lambda x \,.\, * \; 2 \; x$.

It is already known that $f$ is strict, but one should also be able to determine this using formal abstract interpretation. The abstracted definition of $f$ is

$$\begin{aligned} abs(f) &= abs(\lambda x \,.\, * \; 2 \; x) \\ &= \lambda x \,.\, abs(*) \; abs(2) \; x \\ &= \lambda x \,.\, \textbf{and} \; 1 \; x \\ &= \lambda x \,.\, x \end{aligned}$$

and the complete definition of $f$ is

| $x$ | $f\,x$ |
|---|---|
| 0 | 0 |
| 1 | 1 |

According to the strictness analysis condition, which can be stated for the lattice *2* as

$f$ is strict $\Leftrightarrow abs(f)\; 0 = 0$,

$abs(f)\; 0 = 0$, and so $f$ is strict. Were this information to be given to a compiler and used in the evaluation of $f\,x$, $f$ and $x$ could be evaluated simultaneously, thus increasing the efficiency of the evaluation process.

Another, more complicated, example of strictness analysis is the following factorial function and its analysis:

letrec $f = \lambda n \,.\, if \; (= n \; 0) \; 1 \; (* \; n \; (f \; (- \; n \; 1)))$

The abstracted form of $f$ is

$$
\begin{aligned}
abs(f) &= abs(\lambda\, n\, .\, \mathit{if}\ (=\ n\ 0)\ 1\ (*\ n\ (f\ (-\ n\ 1)))) \\
&= \lambda\, n\, .\, abs(\mathit{if}\ (=\ n\ 0)\ 1\ (*\ n\ (f\ (-\ n\ 1)))) \\
&= \lambda\, n\, .\, abs(=\ n\ 0)\ \textbf{and}\ (abs(1)\ \textbf{or}\ abs(*\ n\ (f\ (-\ n\ 1)))) \\
&= \lambda\, n\, .\, (n\ \textbf{and}\ 1)\ \textbf{and}\ (1\ \textbf{or}\ (n\ \textbf{and}\ abs(f)\ (n\ \textbf{and}\ 1))) \\
&= \lambda\, n\, .\, n\ \textbf{and}\ (1) \\
&= \lambda\, n\, .\, n.
\end{aligned}
$$

Thus, the factorial function is strict.

Another example, which requires the finding of a least fixpoint, is the factorial function defined in a different manner. This is the accumulating factorial function:

$$\text{letrec } f = \lambda\, x\, .\, \lambda\, y\, .\, \mathit{if}\ (=\ x\ 0)\ y\ (f\ (-\ x\ 1)\ (*\ x\ y))$$

The abstracted form of this function is

$$abs(f) = \lambda\, x\, .\, \lambda\, y\, .\, x\ \textbf{and}\ (y\ \textbf{or}\ f\ x\ (x\ \textbf{and}\ y))$$

The least fixpoint of this function can be found as follows:

$$
\begin{aligned}
f_0 &= \lambda\, x\, .\, \lambda\, y\, .\, 0 \\
f_1 &= \lambda\, x\, .\, \lambda\, y\, .\, x\ \textbf{and}\ (y\ \textbf{or}\ f_0\ x\ (x\ \textbf{and}\ y)) \\
&= \lambda\, x\, .\, \lambda\, y\, .\, x\ \textbf{and}\ (y\ \textbf{or}\ 0) \\
&= \lambda\, x\, .\, \lambda\, y\, .\, x\ \textbf{and}\ y \\
f_2 &= \lambda\, x\, .\, \lambda\, y\, .\, x\ \textbf{and}\ (y\ \textbf{or}\ f_1\ x\ (x\ \textbf{and}\ y)) \\
&= \lambda\, x\, .\, \lambda\, y\, .\, x\ \textbf{and}\ (y\ \textbf{or}\ (x\ \textbf{and}\ (x\ \textbf{and}\ y))) \\
&= \lambda\, x\, .\, \lambda\, y\, .\, x\ \textbf{and}\ (y\ \textbf{or}\ (x\ \textbf{and}\ y)) \\
&= \lambda\, x\, .\, \lambda\, y\, .\, x\ \textbf{and}\ y
\end{aligned}
$$

Therefore, $f_2 = f_1 = abs(f) = \lambda\, x\, .\, \lambda\, y\, .\, x\ \textbf{and}\ y$. The analysis of this function is as follows:

| $x$ | $f\ x$ |
|---|---|
| 0 | $\lambda\ y\, .\, 0$ |
| 1 | $\lambda\ y\, .\, y$ |

where both $\lambda\, y\, .\, 0$ and $\lambda\, y\, .\, y$ are strict functions. Thus $abs(f)\ 0 = \lambda\, y\, .\, 0 = bottom_{[2 \rightarrow 2]}$, and the accumulating factorial function is strict in both of its arguments.

### 4.1.2. Strictness analysis with lists

To incorporate lists into strictness analysis, the lattice used for list abstraction must be defined. But, before this can be done, the information it represents must be decided. Since the problem at hand is that of strictness analysis, one important feature of an abstract base lattice for lists is that it represent various kinds of non-termination in list structures. For example, it should differentiate between the two list functions *sumlist* and *length*, as defined below:

letrec *sumlist* $= \lambda\, x\, .\, if\_null\ x\ 0\ (+\ (hd\ x)\ (sumlist\ (tl\ x)))$,
*length* $= \lambda\, x\, .\, if\_null\ x\ 0\ (+\ 1\ (length\ (tl\ x)))$;

since the first one always evaluates every element of the list argument, and the second one never does. This means that while the *sumlist* function needs both the list and the elements of the list, the *length* function needs only the list. This information would be useful, and an abstract base lattice which captures it could be put to use in strictness analysis.

Another useful quality of list structures is length, since lists are composite data structures and many functions operate by recursively breaking them up into their primitive constituents. But there are an infinite number of possible list lengths, and to represent all of these would require an infinite sized abstract base lattice. This is not permitted by the current implementation, as the abstract base lattice representation is explicit and not implicit.

One abstract base lattice which has been proposed for the analysis of lists reached the first goal of list abstraction mentioned above, but not the second. That is the lattice *4*, defined in [Burn, 1987], and shown in Figure 4.3. In this definition, "undefined" is equivalent to non-terminating, since the only way an expression evaluation can be undefined is for the evaluation to give no value. This can only happen, since λ-expression reduction is well-defined, when the evaluation is non-terminating. Thus, the abstract base lattice *4* captures information about an infinite number of lists in a finite number of

elements, as required by the implementation, but does not capture information about list length. This lattice will be the one used in the following examples of strictness analysis.

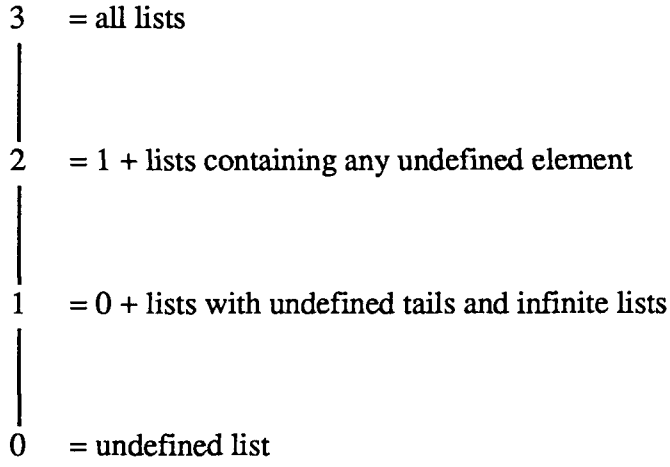



Figure 4.3. The abstract base lattice *4*

The abstractions that go along with the primitive list operations must be defined. These builtin list operations are *hd*, *tl*, *cons*, and *if_null*, and their abstract counterparts are defined as shown in Figure 4.4. The function *if_null* is used instead of *null* as a primitive list function for reasons which follow.

If *null* was a builtin list primitive, its abstract counterpart would be

| $x$ | $abs(null)\ x$ |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 1 |
| 3 | 1 |

and since the definition of $abs(if)$, or $if^a$, is

$$if^a = \lambda\, a\, .\, \lambda\, b\, .\, \lambda\, c\, .\, a \textbf{ and } (b \textbf{ or } c),$$

the difference between abstractions of (1) "$if^a\ (null^a\ a)\ b\ c$" and (2) "$if\_null^a\ a\ b\ c$" can be seen in the following ($4 \rightarrow 2 \rightarrow 2 \rightarrow 2$ case only, without loss of generality):

| $x$ | $abs(hd)\ x$ | $abs(tl)\ x$ |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 1 | 1 |
| 2 | 1 | 3 |
| 3 | 1 | 3 |

where:

$abs(hd) : 4 \rightarrow 2$
$abs(tl) : 4 \rightarrow 4$
$abs(cons) : 2 \rightarrow 4 \rightarrow 4$

| $x$ | $y$ | $abs(cons)\ x\ y$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 2 | 2 |
| 0 | 3 | 2 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |
| 1 | 2 | 2 |
| 1 | 3 | 3 |

| $l$ | $abs(if\_null)\ l\ a\ b$ |
|---|---|
| 0 | 0 |
| 1 | $b$ |
| 2 | $b$ |
| 3 | $abs(if)\ 1\ a\ b$ |

where:

| $l$ | $abs(if\_null)\ l\ a\ b$ |
|---|---|
| 0 | $b$ |
| 1 | $b$ |
| 2 | $b$ |
| 3 | $lub\ \{a, b\}$ |

where:

$abs(if\_null) : 4 \rightarrow 2 \rightarrow 2 \rightarrow 2$ $abs(if\_null) : 4 \rightarrow 4 \rightarrow 4 \rightarrow 4$

Figure 4.4. Definitions of list primitive abstractions

$abs((1)) = (null^a\ a)$ **and** $(b$ **or** $c)$
$= if\ a = 0\ then\ 0\ else\ b$ **or** $c$

$abs((2)) = if\ a = 0\ then\ 0\ else\ (if\ a = 1$ or $a = 2\ then\ c\ else\ b$ **or** $c)$

The latter definition ($abs((2))$) is more refined than the former. This refinement helps to give more information.

As an example of how strictness analysis can give information about different kinds of list functions, consider the functions defined below:

letrec $f = \lambda\ x\ .\ if\_null\ x\ 0\ 1,$
$length = \lambda\ x\ .\ if\_null\ x\ 0\ (+\ 1\ (length\ (tl\ x))),$
$sum = \lambda\ x\ .\ if\_null\ x\ 0\ (+\ (hd\ x)\ (sum\ (tl\ x)));$

Each function has slightly different "needs" in terms of the type of list evaluation it does. These "needs" will now be discovered by performing strictness analysis.

First, consider $f$. Its abstract form, $f^a : 4 \rightarrow 2$, is

$f^a = \lambda\ x\ .\ if\ x = 0\ then\ 0\ else\ 1,$

or, equivalently, using a shorthand notation that will be used extensively from now on:

$$f^a = \lambda x . \quad \begin{array}{l} 0 \rightarrow 0 \\ 1 \rightarrow 1 \\ 2 \rightarrow 1 \\ 3 \rightarrow 1. \end{array}$$

Thus $f$ is non-terminating when its list argument is undefined. So, $f$ can have as an argument a list which contains undefined elements, and these undefined elements are never needed by $f$.

Consider also the *length* function. Its abstract interpretation is

$$\begin{array}{l} length^a = \lambda x . 0 \rightarrow 0 \\ \qquad\qquad\quad 1 \rightarrow length^a\ (tl^a\ 1) = length^a\ 1 \\ \qquad\qquad\quad 2 \rightarrow length^a\ (tl^a\ 2) = length^a\ 3 \\ \qquad\qquad\quad 3 \rightarrow 1 \end{array}$$

and the least fixpoint can be found as follows:

$$length^a_0 \quad = \lambda x . 0$$

$$\begin{array}{ll} length^a_1 & = \lambda x . 0 \rightarrow 0 \\ & \qquad 1 \rightarrow length^a_0\ 1 = 0 \\ & \qquad 2 \rightarrow length^a_0\ 3 = 0 \\ & \qquad 3 \rightarrow 1 \end{array}$$

$$\begin{array}{ll} length^a_2 & = \lambda x . 0 \rightarrow 0 \\ & \qquad 1 \rightarrow length^a_1\ 1 = 0 \\ & \qquad 2 \rightarrow length^a_1\ 3 = 1 \\ & \qquad 3 \rightarrow 1 \end{array}$$

$$\begin{array}{ll} length^a_3 & = \lambda x . 0 \rightarrow 0 \\ & \qquad 1 \rightarrow length^a_2\ 1 = 0 \\ & \qquad 2 \rightarrow length^a_2\ 3 = 1 \\ & \qquad 3 \rightarrow 1 \end{array}$$

The least fixpoint of $length^a$ is

$$\begin{array}{l} length^a = \lambda x . 0 \rightarrow 0 \\ \qquad\qquad\quad 1 \rightarrow 0 \\ \qquad\qquad\quad 2 \rightarrow 1 \\ \qquad\qquad\quad 3 \rightarrow 1 \end{array}$$

Therefore, *length* is the kind of list function that needs its list argument, and also needs a defined or finite tail upon its argument.

Consider also the *sum* function defined above. Its abstract interpretation is as follows:

$$sum^a = \lambda x\,.\, if\_null^a\, x\, 1\, ((hd^a\, x)\ \textbf{and}\ (sum^a\, (tl^a\, x))),$$

$$\begin{aligned} = \lambda x\,.\, & 0 \rightarrow 0 \\ & 1 \rightarrow (hd^a\ 1)\ \textbf{and}\ sum^a\ (tl^a\ 1) = 1\ \textbf{and}\ sum^a\ 1 = sum^a\ 1 \\ & 2 \rightarrow (hd^a\ 2)\ \textbf{and}\ sum^a\ (tl^a\ 2) = 1\ \textbf{and}\ sum^a\ 3 = sum^a\ 3 \\ & 3 \rightarrow 1 \end{aligned}$$

and its least fixpoint can be found as follows:

$$sum^a_0 = \lambda x\,.\, 0$$

$$\begin{aligned} sum^a_1 = \lambda x\,.\, & 0 \rightarrow 0 \\ & 1 \rightarrow sum^a_0\ 1 = 0 \\ & 2 \rightarrow sum^a_0\ 3 = 0 \\ & 3 \rightarrow 1 \end{aligned}$$

$$\begin{aligned} sum^a_2 = \lambda x\,.\, & 0 \rightarrow 0 \\ & 1 \rightarrow sum^a_1\ 1 = 0 \\ & 2 \rightarrow sum^a_1\ 3 = 1 \\ & 3 \rightarrow 1 \end{aligned}$$

$$\begin{aligned} sum^a_3 = \lambda x\,.\, & 0 \rightarrow 0 \\ & 1 \rightarrow sum^a_2\ 1 = 0 \\ & 2 \rightarrow sum^a_2\ 3 = 1 \\ & 3 \rightarrow 1 \end{aligned}$$

Therefore, the least fixpoint of $sum^a$ is

$$\begin{aligned} sum^a = \lambda x\,.\ \ & 0 \rightarrow 0 \\ & 1 \rightarrow 0 \\ & 2 \rightarrow 1 \\ & 3 \rightarrow 1, \end{aligned}$$

which implies that perhaps *sum* does not need to evaluate each element of its list argument. This could be more informative, since *sum* does evaluate its list arguments' elements. The reason that this information is not captured in the analysis of *sum* is that $hd^a$ and $tl^a$ are treated separately, for $hd^a\ 2 = 1$ *and* $tl^a\ 2 = 3$ are both true, but if $hd^a$ and $tl^a$ are both evaluated upon the same expression, then for $abs(x) = 2$, $tl^a\ abs(x) = 3 \Rightarrow hd^a\ abs(x) = 0$,

and $hd^a\ abs(x) = 1 \Rightarrow tl^a\ abs(x) = 2$. This relationship can be captured by an alternate definition of *sum*, which goes deeper into the list structure of its argument, as follows:

letrec $sum = \lambda\, x\, .\ if\_null\ x\ 0\ (if\_null\ (tl\ x)\ (hd\ x)\ (+\ (hd\ x)\ (+\ (hd\ (tl\ x))\ (sum\ (tl\ (tl\ x))))))$;

which has the abstraction

$$sum^a = \lambda\, x\, .\ if\_null^a\ x\ 1\ (if\_null^a\ (tl^a\ x)\ (hd^a\ x)\ ((hd^a\ x)\ \textbf{and}\ (hd^a\ (tl^a\ x))\ \textbf{and}\ (sum^a\ (tl^a\ (tl^a\ x)))))$$

$$\begin{aligned}
= \lambda\, x\, .\quad & 0 \rightarrow 0 \\
& 1 \rightarrow (tl^a\ x =)\ \ 0 \rightarrow 0 \\
& \qquad\qquad\quad 1 \rightarrow (hd^a\ x)\ \textbf{and}\ (hd^a\ (tl^a\ x))\ \textbf{and}\ (sum^a\ (tl^a\ (tl^a\ x))) \\
& \qquad\qquad\quad 2 \rightarrow (hd^a\ x)\ \textbf{and}\ (hd^a\ (tl^a\ x))\ \textbf{and}\ (sum^a\ (tl^a\ (tl^a\ x))) \\
& \qquad\qquad\quad 3 \rightarrow (hd^a\ x)\ \textbf{or}\ ((hd^a\ x)\ \textbf{and}\ (hd^a\ (tl^a\ x))\ \textbf{and}\ (sum^a\ (tl^a\ (tl^a\ x)))) \\
& 2 \rightarrow (tl^a\ x =)\ \ 0 \rightarrow 0 \\
& \qquad\qquad\quad 1 \rightarrow (hd^a\ x)\ \textbf{and}\ (hd^a\ (tl^a\ x))\ \textbf{and}\ (sum^a\ (tl^a\ (tl^a\ x))) \\
& \qquad\qquad\quad 2 \rightarrow (hd^a\ x)\ \textbf{and}\ (hd^a\ (tl^a\ x))\ \textbf{and}\ (sum^a\ (tl^a\ (tl^a\ x))) \\
& \qquad\qquad\quad 3 \rightarrow (hd^a\ x)\ \textbf{or}\ ((hd^a\ x)\ \textbf{and}\ (hd^a\ (tl^a\ x))\ \textbf{and}\ (sum^a\ (tl^a\ (tl^a\ x)))) \\
& 3 \rightarrow 1
\end{aligned}$$

$$\begin{aligned}
= \lambda\, x\, .\quad & 0 \rightarrow 0 \\
& 1 \rightarrow sum^a\ 1 \\
& 2 \rightarrow 0 \\
& 3 \rightarrow 1.
\end{aligned}$$

Therefore, the least fixpoint of $sum^a$ is

$$sum^a_0 = \lambda\, x\, .\ 0$$

$$\begin{aligned}
sum^a_1 = \lambda\, x\, .\ & 0 \rightarrow 0 \\
& 1 \rightarrow sum^a_0\ 1 = 0 \\
& 2 \rightarrow 0 \\
& 3 \rightarrow 1
\end{aligned}$$

$$\begin{aligned}
sum^a_2 = \lambda\, x\, .\ & 0 \rightarrow 0 \\
& 1 \rightarrow sum^a_1\ 1 = 0 \\
& 2 \rightarrow 0 \\
& 3 \rightarrow 1.
\end{aligned}$$

So,

$$\begin{aligned}
sum^a = \lambda\, x\, .\quad & 0 \rightarrow 0 \\
& 1 \rightarrow 0 \\
& 2 \rightarrow 0 \\
& 3 \rightarrow 1,
\end{aligned}$$

giving the information that *sum* needs the elements of its list argument.

A technique by Burn [Burn, 1987] avoids this cumbersome analysis by incorporating into the λ-calculus pattern matching in its parameter passing mechanisms. The *sum* function could have been defined as follows (using a pattern-matching case structure):

$$\begin{array}{lll} \text{letrec } sum = \lambda\, x\,. & [] & \rightarrow 0 \\ & (cons\ x\ xs) & \rightarrow +\ x\ (sum\ xs), \end{array}$$

or, as in [Burn, 1987]:

$$\begin{array}{ll} sum\ [] & = 0 \\ sum\ x{:}xs & = x + sum\ xs. \end{array}$$

## 4.2. Termination Analysis

Termination analysis is the process of determining whether a given program or function invocation terminates. More generally, this is known as the halting problem, which is unsolvable. However, the intractability of the problem in general does not preclude its solution in the case of a specific program. For example, although it is not possible to develop an algorithm which performs termination analysis on all programs, gives a correct answer, and is guaranteed to terminate itself, it is possible to develop one which does termination analysis upon all programs, gives a *safe* answer, and is guaranteed to terminate. The approach taken by abstract interpretation is not to answer exactly the question of whether or not a program terminates, but to answer it approximately, or safely. A safe answer in the case of termination analysis is one which underestimates the situations in which a program might terminate, but never overestimates. Thus, if a program $P$ is declared by the implementation to terminate upon the set of values $V$, this does not imply that for an $x$ not in $V$, $P(x)$ does not terminate.

As an example of what termination analysis produces, consider $f$ defined as follows:

letrec $f = \lambda\, x\, .\, *\, 2\, x$.

The termination analysis finds that $f$ terminates whenever $x$ terminates, since the abstraction function derived is

$$f^a = \lambda\, x\, .\, x,$$

and the multiplication of $x$ by 2 does nothing to change the termination characteristics of $f$ from those of $x$.

Another, more complex, example of a function and its analysis, is the following analysis of the factorial function:

letrec $f = \lambda\, n\, .\, \mathit{if}\ (=\, n\ 0)\ 1\ (*\, n\ (f\ (-\, n\ 1)))$

This is known to terminate for non-negative arguments. However, a termination analysis produces the following abstract function

$$f^a = \lambda\, n\, .\, \mathit{if}\ n = 0\ \mathit{then}\ 0\ \mathit{else}\ ?$$

which means that if $n$ doesn't terminate then neither does $f\ n$, but otherwise no information is known (the lattice used will be described later). This is a safe answer, if not the most informative answer.

One use to which such information can be put is proving a program correct. Often when proving that a computer program meets its specification, it is necessary that it terminates. Instead of accomplishing this proof through special purpose methods, it is possible to submit the program to a termination analyzer, and receive an informative diagnosis about the conditions for the program's termination. Of course, perfect knowledge about in what situations a program will terminate can not be derived (otherwise the halting problem would be solved).

In order to derive useful termination information using abstract interpretation, an appropriate abstract base lattice over which the abstract interpretation will operate must be defined. The desired information is whether the program terminates or not. That is, first of

all, does the program definitely terminate, and, second, if not, what does it do? The only other option is non-termination, but the system will not always be able to determine this exactly. Thus the necessary conditions upon the abstract base lattice is that it indicate definite termination and possible non-termination (or unknown). An appropriate base lattice *3* can be derived from the concrete base lattices in a natural manner, along with a mapping upon constants, as in Figure 4.5. Non-termination maps to non-termination (0), termination with any value (integer or boolean) maps to termination (1), and unknown (*top*) maps to unknown(?). Thus the abstract base lattice selected is again a simplification of the concrete base lattices.

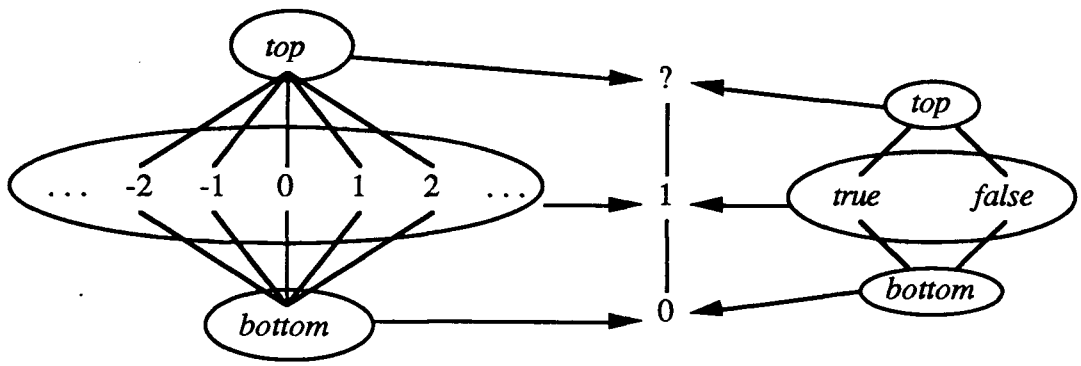



Figure 4.5. The mapping of constants into the abstract base lattice *3*

The abstract interpretation of some primitive builtin functions, such as * and *if*, must be further specified. These are the only non-list-valued functions for which it is necessary to specify an abstraction, since the other such functions are similar. Thus, for multiplication, if either of the arguments is non-terminating, then the result is non-terminating, if both terminate then the result terminates, and otherwise the result is unknown. This is summarized in Figure 4.6. The *if* builtin function, on the other hand, is more complicated, but after a thorough analysis, as shown in Figure 4.6, can be summarized as

| *a* | *b* | *abs(f) a b* |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | ? | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |
| 1 | ? | ? |
| ? | 0 | 0 |
| ? | 1 | ? |
| ? | ? | ? |

where $f$ is one of
+, –, *, /, etc.

| *a* | *b* | *c* | *abs(if) a b c* |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | ? | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 0 | 1 | ? | 0 |
| 0 | ? | 0 | 0 |
| 0 | ? | 1 | 0 |
| 0 | ? | ? | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | ? |
| 1 | 0 | ? | ? |
| 1 | 1 | 0 | ? |
| 1 | 1 | 1 | 1 |
| 1 | 1 | ? | ? |
| 1 | ? | 0 | ? |
| 1 | ? | 1 | ? |
| 1 | ? | ? | ? |
| ? | 0 | 0 | 0 |
| ? | 0 | 1 | ? |
| ? | 0 | ? | ? |
| ? | 1 | 0 | ? |
| ? | 1 | 1 | ? |
| ? | 1 | ? | ? |
| ? | ? | 0 | ? |
| ? | ? | 1 | ? |
| ? | ? | ? | ? |

Figure 4.6. Refined abstraction definitions for builtin functions

$$if^{a}\ a\ b\ c = \quad \textit{if}\ glb\ \{a, lub\ \{b, c\}\} = 0\ \textit{then}\ 0\ \textit{else if}\ a = b = c = 1\ \textit{then}\ 1\ \textit{else}\ ?$$

Normally, the language in which the functions are specified is capable of dealing with higher-level data-structures than the primitive integers and booleans, such as lists. For these languages, an additional abstract base lattice is needed: one which abstracts essential termination and non-termination information about functions that take lists as arguments and/or return lists as values. This addition will be discussed following a more detailed discussion of the simpler model.

### 4.2.1. Simple termination analysis

What has already been defined is sufficient to perform termination analysis without lists. A few examples will now be given.

Consider the function given as an example above:

letrec $f = \lambda\, x\, .\, *\ 2\ x$.

This is abstractly interpreted as:

$$\begin{aligned} f^a &= abs(\lambda\, x\, .\, *\ 2\ x) \\ &= \lambda\, x\, .\, abs(*)\ abs(2)\ abs(x) \\ &= \lambda\, x\, .\, abs(*)\ 1\ x \\ &= \lambda\, x\, .\, (\lambda\, a\, .\, \lambda\, b\, .\, \mathit{if}\ (=\ (\mathit{glb}\ a\ b)\ 0)\ 0\ (\mathit{lub}\ a\ b))\ 1\ x \\ &= \lambda\, x\, .\, \mathit{if}\ (=\ (\mathit{glb}\ 1\ x)\ 0)\ 0\ (\mathit{lub}\ 1\ x) \\ &= \lambda\, x\, .\, \mathit{if}\ (=\ x\ 0)\ 0\ (\mathit{lub}\ 1\ x) \\ &= \lambda\, x\, .\, \mathit{if}\ (=\ x\ 0)\ x\ x \\ &= \lambda\, x\, .\, x \end{aligned}$$

where the last three steps are simplifications based on the nature of the functions *glb* and *lub* (this is performed by the implementation in a more explicit manner, rather than on the definition of $f$ as shown here for clarity). Under abstract interpretation the function $f$, which doubles the value of its argument, is simply the identity function with respect to the termination or non-termination of its argument. This is valuable information to know about a function, if not for reasons of efficient implementation, then for more theoretical reasons; it is now known that for similar mathematical functions invoked with one of the arguments a constant, that the function can be ignored in termination analysis.

Unfortunately, another common type of function cannot be so easily analyzed. Consider the factorial function:

letrec $f = \lambda\, n\, .\, \mathit{if}\ (=\ n\ 0)\ 1\ (*\ n\ (f\ (-\ n\ 1)))$.

The abstract interpretation for this function is

$$\begin{aligned} f^a &= abs(\lambda\, n\, .\, if\ (=\ n\ 0)\ 1\ (*\ n\ (f\ (-\ n\ 1)))) \\ &= \lambda\, n\, .\, abs(if)\ abs(=\ n\ 0)\ abs(1)\ abs(*\ n\ (f\ (-\ n\ 1)))) \\ &= \lambda\, n\, .\, abs(if)\ n\ 1\ (abs(*)\ n\ (f^a\ n)) \end{aligned}$$

where $abs(if)$ and $abs(*)$ are as previously derived. Using the abstract interpretation system to finish the process, the abstraction of $f$ can be represented as

$$f^a = \lambda\, n\, .\, if\ n = 0\ then\ 0\ else\ ?$$

indicating that even if $n$ terminates, then $f$ is still not known to terminate. This is not the most informative answer one could desire about $f$. The problem stems from the fact that $f$ recurses except when its argument has the value 0. This detail is obvious to the user of the function, but not to the implementation, because upon translation from the concrete to the abstract lattices all distinctions between specific integers are lost and zero is treated the same as any other integer. This could be solved in this case by representing the concrete base lattice differently, or introducing a more complicated abstract base lattice, as Figure 4.7 suggests. The problem with representing the abstract base lattice as such a detailed lattice is not theoretical, but practical. The relationship between the size of a lattice and its function lattice is exponential, and thus in this case the function lattice is too large for the implementation to handle. (Such a lattice $L$, containing only 5 elements, would cause the creation of a function lattice $[L \rightarrow L]$, containing 126 elements, and the current implementation which represents functions explicitly as annotated lattices cannot deal with such a large function lattice.)

Other types of functions can be analyzed with termination analysis: these use lists and are explained in the next section.

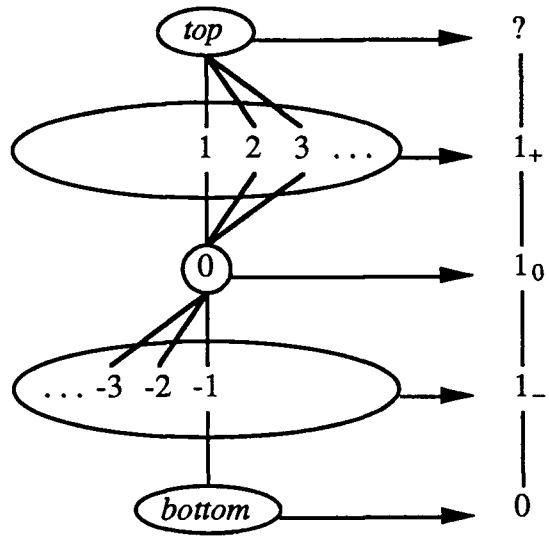



Figure 4.7. A refined mapping of constants

### 4.2.2. Termination analysis with lists

Incorporating lists into termination analysis requires an additional pair of concrete and abstract base lattices and also the abstract counterparts of a number of builtin list functions. The concrete base lattice for lists is not required in practice to be specified, since there is an abstract base lattice and translation scheme which eliminate this need. Thus any reasonable definition of a concrete base lattice may be used (this is useful in the development of the theory behind abstract interpretation with lists, since the concrete base lattice for lists can be chosen specifically to ease the theoretical development). The abstract base lattice definition which is responsible for the lack of restriction on the concrete counterpart is due to Burn [Burn, 1987], and is shown in Figure 4.3. A list is categorized according to where any undefined element might occur. For example, if $x$ represents an expression whose evaluation is non-terminating, then the following categorizations can be made:

| **List** | **Abstraction** |
|---|---|
| [] | 3 |
| [1, 2] | 3 |
| [1, x, 3] | 2 |
| [1, 2, x] | 1 |
| [1, 2, 3, ...] | 1 |
| $x$ | 0 |

Also to be defined are the abstractions of the list functions *hd*, *tl*, *cons*, and *if_null* (instead of *null*, as before). These definitions are as shown in Figure 4.8. An example can now be given of termination analysis applied to functions involving lists.

Consider the previously mentioned function *length*:

letrec $length = \lambda x\,.\,if\_null\ x\ 0\ (+\ 1\ (length\ (tl\ x)))$.

The abstraction of *length* is

$$\begin{aligned} length^a &= abs(\lambda x\,.\,if\_null\ x\ 0\ (+\ 1\ (length\ (tl\ x)))) \\ &= \lambda x\,.\,if\_null^a\ x\ abs(0)\ abs(+\ 1\ (length\ (tl\ x)))) \\ &= \lambda x\,.\,if\_null^a\ x\ 1\ (length^a\ (tl^a\ x)) \end{aligned}$$

$$\begin{aligned} &= \lambda x\,.\,0 \rightarrow 0 \\ &\qquad 1 \rightarrow length^a\ (tl^a\ 1) = length^a\ 1 \\ &\qquad 2 \rightarrow length^a\ (tl^a\ 2) = length^a\ 3 \\ &\qquad 3 \rightarrow if^a\ 1\ 1\ (length^a\ (tl^a\ 3)) = if^a\ 1\ 1\ (length^a\ 3) \end{aligned}$$

and its least fixpoint can be found as follows:

$$length^a_0 = \lambda x\,.\,0$$

$$\begin{aligned} length^a_1 &= \lambda x\,.\,0 \rightarrow 0 \\ &\qquad 1 \rightarrow length^a_0\ 1 = 0 \\ &\qquad 2 \rightarrow length^a_0\ 3 = 0 \\ &\qquad 3 \rightarrow if^a\ 1\ 1\ (length^a_0\ 3) = if^a\ 1\ 1\ 0 = ? \end{aligned}$$

$$\begin{aligned} length^a_2 &= \lambda x\,.\,0 \rightarrow 0 \\ &\qquad 1 \rightarrow length^a_1\ 1 = 0 \\ &\qquad 2 \rightarrow length^a_1\ 3 = ? \\ &\qquad 3 \rightarrow if^a\ 1\ 1\ (length^a_1\ 3) = if^a\ 1\ 1\ ?\ = ? \end{aligned}$$

$$\begin{aligned} length^a_3 &= \lambda x\,.\,0 \rightarrow 0 \\ &\qquad 1 \rightarrow length^a_2\ 1 = 0 \\ &\qquad 2 \rightarrow length^a_2\ 3 = ? \\ &\qquad 3 \rightarrow if^a\ 1\ 1\ (length^a_2\ 3) = if^a\ 1\ 1\ ?\ = ? \end{aligned}$$

| $x$ | $abs(hd)\,x$ | $abs(tl)\,x$ |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 1 | 1 |
| 2 | 1 | 2 |
| 3 | 1 | 3 |

where:

$abs(hd): 4 \rightarrow 3$
$abs(tl): 4 \rightarrow 4$
$abs(cons): 3 \rightarrow 4 \rightarrow 4$

| $x$ | $y$ | $abs(cons)\,x\,y$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 2 | 2 |
| 0 | 3 | 2 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |
| 1 | 2 | 2 |
| 1 | 3 | 3 |
| ? | 0 | 1 |
| ? | 1 | 1 |
| ? | 2 | 2 |
| ? | 3 | 3 |

| $l$ | $abs(if\_null)\,l\,a\,b$ |
|---|---|
| 0 | 0 |
| 1 | $b$ |
| 2 | $b$ |
| 3 | $abs(if)\ 1\,a\,b$ |

where:

$abs(if\_null): 4 \rightarrow 3 \rightarrow 3 \rightarrow 3$

| $l$ | $abs(if\_null)\,l\,a\,b$ |
|---|---|
| 0 | $b$ |
| 1 | $b$ |
| 2 | $b$ |
| 3 | $lub\ \{a, b\}$ |

where:

$abs(if\_null): 4 \rightarrow 4 \rightarrow 4 \rightarrow 4$

Figure 4.8. The abstraction definitions of primitive list functions

Therefore, the least fixpoint of $length^a$ is

$$\begin{aligned} length^a = \lambda\, x\,.\ & 0 \rightarrow 0 \\ & 1 \rightarrow 0 \\ & 2 \rightarrow ? \\ & 3 \rightarrow ?. \end{aligned}$$

Or, equivalently,

$$length^a = \lambda\, x\,.\ if\ lub\ \{1, x\} = 1\ then\ 0\ else\ ?.$$

This abstraction suffers from the same problem as that discovered with the factorial function. That is, in the abstract base lattice definition no distinction is made between the empty list and any other kind of finite list, thus the analysis is less informative than it otherwise might be. It is difficult to solve this problem without introducing a non-finite abstract base lattice for the list concrete base lattice (such a simple solution as introducing a

2.5 element into the existing abstract base lattice, representing an empty list, causes non-monotonicity in some functions, such as the function

letrec $f = \lambda\, x\, .\, \mathit{if\_null}\ x\ 1\ 0$

with the following values:

| $x$ | $f\ x$ |
|---|---|
| 0 | 0 |
| 1 | 0 |
| 2 | 0 |
| 2.5 | 1 |
| 3 | 0 |

It is non-monotonic.).

Another example of termination analysis applied to functions using list data-structures, the analysis of the *sumlist* function, can be found in appendix A.

# Chapter 5

# Conclusion

The process of, and framework underlying, abstract interpretation has changed in recent years, overcoming old limitations and encountering new ones. This development, including the implementation discussed in the previous chapters, shall be summarized in the next section. Also, the implementation's performance with respect to its stated goals will be evaluated. Following will be discussed some of the work to be done in the future and the problems left to be overcome.

## 5.1. Summary and Evaluation

Abstract interpretation, the process of abstracting useful information from a computer program in a systematic manner based on lattice-theoretical methods, was established as a program analysis technique in [Cousot & Cousot, 1977]. Later, it was viewed differently, from the perspective of functional programming, and validated in a practical sense in [Mycroft, 1980], [Mycroft, 1981], and [Mycroft & Nielson, 1983]. At this point it was useable as a program analysis technique only for first-order functions over flat base lattices. Later work [Burn, Hankin & Abramsky, 1986] enabled the technique to be extended to arbitrary higher-order functions over flat base lattices. Current work is aimed at extending the technique's theoretical basis to apply to non-flat base lattices [Hughes, 1985], [Hughes, 1987], and [Wadler & Hughes, 1987], but has yet to see success in this endeavor.

In a practical sense, abstract interpretation has made progress in both of these areas. Some of the work done on abstractly interpreting higher-order functions is described in [Mishra & Keller, 1984] and [Clack & Peyton Jones, 1985], and descriptions of some work done using non-flat base lattices can be found in [Hughes, 1985] and [Clack & Peyton Jones, 1985], although a formal proof of theoretical validity is still lacking for the latter developments. One problem which is encountered in attempting to analyze arbitrary-order functions over finite base lattices is that of efficiency. This has been addressed in [Clack & Peyton Jones, 1985], and [Martin & Hankin, 1987] by introducing and refining a method of function representation and an efficient algorithm for computing such representations. The representation is done with frontiers in lattices, and the algorithm is known as the frontiers algorithm.

In this implementation, the frontiers algorithm and method of function representation have been modified and simplified. Also, the abstract interpretation process has been automated so that the user need only specify the abstraction rules for the base lattices, and for the builtin functions present in the functional language (a $\lambda$-calculus). Thus, as has been shown in the previous chapters, the implementation is automated, application-independent, and has its heart — the representation of abstracted functions and the computation of such — implemented in a simple and straightforward manner, using annotated lattices.

One drawback to the implementation is that it will not allow the definition of arbitrary base lattices — all base lattices must be finite to allow the functions defined over them to be represented explicitly. To remove this restriction would require choosing a different method of function representation and thus function comparison, such as a canonical textual representation. However, such a method of function representation and comparison is currently inefficient, and allows for the possibility of failure in situations

where the explicit representation would succeed. As an example of the difficulties involved in textual comparisons, consider the following two functions, and without evaluating them, justify the fact that they are equal:

letrec $f = \lambda x . + x x,$
$g = \lambda x . * 2 x;$

This problem cannot be solved in general, but it may be solvable in the case of certain classes of functions commonly found in programs (such as arithmetic functions, data-structure composition and decomposition functions, list traversal functions, etc.).

## 5.2. Future Work

The areas in which abstract interpretation can conceivably be extended in the future are two: applications and generality.

Along with the application of abstract interpretation to strictness analysis and termination analysis, other types of analysis have been tried. These include such things as data-typing [Mishra & Keller, 1984] and structure-sharing [Hudak & Bloss, 1985]. Automatic data-typing of programs is not now done through abstract interpretation, since it seems to require an infinitely large abstract base lattice (to account for the infinite number of potential user-defined data-types in a typed language). The structure-sharing analysis (analyzing the uses of a data-structure to see if it can be destructively updated instead of copied, thus saving time and storage space) may still produce results in the future.

Even more detailed strictness analyses have been and are continuing to be developed. In [Wadler & Hughes, 1987] etc., such concepts as contexts and projections are introduced in an effort to analyze programs for context-sensitive strictness. These efforts have met with limited success, but this area is still in the process of being formalized.

Other types of analysis have been suggested but not tried. For example, in [Cousot & Cousot, 1977], such problems as data-flow analysis, program testing, program performance, program correctness, and program transformation are also potential candidates for abstract interpretation. Obviously, some of these applications require an infinite size abstract base lattice. If this was ever made possible in a practical manner then abstract interpretation would prove to be a powerful program analysis technique in these areas as well.

Another area in which the process of abstract interpretation can and should be expanded is its generality. Again, infinite size abstract base lattices should be useable. Perhaps this could be done by defining certain types of lattices which would allow an easily computable canonical representation of functions, and at the same time ensure that the AKC does terminate. (It is possible to have infinite size lattices where all AKC's must terminate, as is the case with the concrete base lattice of integers.)

A second way in which abstract interpretation could be expanded is in the type of language it analyzes. For example, instead of the usual λ-calculus, abstract interpretation could instead analyze an extended λ-calculus, such as the language Miranda™ [Turner, 1985], which allows concise function definitions, pattern-matching λ-abstractions, and user-defined polymorphic types. Alternatively, abstract interpretation could be extended to handle not only functional languages, but relational languages as well. It would be instructive to see how abstract interpretation handles the organization of concurrency in a concurrent prolog as opposed to conventional methods.

# Bibliography


[Abramsky, 1985] S. Abramsky. Strictness Analysis and Polymorphic Invariance (Extended Abstract). In H. Ganzinger and N. D. Jones, editors, *Lecture Notes in Computer Science 217: Programs as Data Objects*, pages 1–23, Springer-Verlag, Copenhagen, Denmark, October 1985.

[Abramson, 1984] H. Abramson. Definite Clause Translation Grammars, In *Proceedings of the 1984 International Symposium on Logic Programming*, pages 233–241, Atlantic City, New Jersey, February 1984.

[Burn, 1987] G. L. Burn. Evaluation Transformers — A Model for the Parallel Evaluation of Functional Languages (Extended Abstract). In G. Kahn, editor, *Lecture Notes in Computer Science 274: Functional Programming Languages and Computer Architecture*, pages 446–470, Springer-Verlag, Portland, Oregon, September 1987.

[Burn *et al.*, 1986] G. L. Burn, C. Hankin, and S. Abramsky. Strictness Analysis for Higher-Order Functions. *Science of Computer Programming*, 7(3):249–278, November 1986.

[Clack *et al.*, 1985] C. Clack and S. L. Peyton Jones. Strictness Analysis — a Practical Approach. In J-P. Jouannaud, editor, *Lecture Notes in Computer Science 201: Functional Programming Languages and Computer Architecture*, pages 35–49, Springer-Verlag, Nancy, France, September 1985.

[Cousot *et al.*, 1977] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, January 1977.

[Hankin *et al.*, 1986] C. L. Hankin, G. L. Burn, and S. L. Peyton Jones. A Safe Approach to Parallel Combinator Reduction (Extended Abstract). In B. Robinet and R. Wilhelm, editors, *Lecture Notes in Computer Science 213: ESOP 86: European Symposium on Programming*, pages 99–110, Springer-Verlag, Saarbrucken, Germany, March 1986.

[Hughes, 1985] J. Hughes. Strictness Detection in Non-Flat Domains. In H. Ganzinger and N. D. Jones, editors, *Lecture Notes in Computer Science 217: Programs as Data Objects*, pages 112–135, Springer-Verlag, Copenhagen, Denmark, October 1985.

[Martin *et al.*, 1987] C. Martin and C. Hankin. Finding Fixed Points in Finite Lattices. In G. Kahn, editor, *Lecture Notes in Computer Science 274: Functional Programming Languages and Computer Architecture*, pages 426–445, Springer-Verlag, Portland, Oregon, September 1987.

[Mishra *et al.*, 1984] P. Mishra and R. M. Keller. Static Inference of Properties of Applicative Programs. In *Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 235–244, Salt Lake City, Utah, January 1984.

[Mycroft, 1980] A. Mycroft. The Theory and Practice of Transforming Call-by-Need into Call-by-Value. In B. Robinet, editor, *Lecture Notes in Computer Science 83: Fourth International Symposium on Programming*, pages 269–281, Springer-Verlag, Paris, France, April 1980.

[Mycroft, 1981] A. Mycroft. *Abstract Interpretation and Optimizing Transformations for Applicative Programs*. Ph.D. Thesis, Edinburgh University, Edinburgh, Scotland, 1981.

[Mycroft *et al.*, 1983] A. Mycroft and F. Nielson. Strong Abstract Interpretation using Power Domains (Extended Abstract). In J. Diaz, editor, *Lecture Noes in Computer Science 154: Automata, Languages and Programming, 10th Colloquium*, pages 536–547, Springer-Verlag, Barcelona, Spain, July 1983.

[Stoy, 1977] J. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. The MIT Press Series in Computer Science*, MIT Press, Cambridge, Massachusetts, 1977.

[Turner, 1985] D. A. Turner. Miranda: A Non-Strict Functional Language with Polymorphic Types. In J-P. Jouannaud, editor, *Lecture Notes in Computer Science 201: Functional Programming Languages and Computer Architecture*, pages 1–16, Springer-Verlag, Nancy, France, September 1985.

[Wadler *et al.*, 1987] P. Wadler and R. J. M. Hughes. Projections for Strictness Analysis. In G. Kahn, editor, *Lecture Notes in Computer Science 274: Functional Programming Languages and Computer Architecture*, pages 385–407, Springer-Verlag, Portland, Oregon, September 1987.

# Appendix A

# Sample Runs

This appendix contains some sample runs of the system. Most of the functions which are analyzed in the following pages have already been analyzed in a more compact manner in Chapter 4. They have been reproduced here in order to demonstrate the manner in which the system actually performs. Both kinds of analysis discussed in the previous pages are shown here: first strictness analysis, then termination analysis, both being performed on functions which may contain lists.

As a preliminary note of explanation, the way in which lattices are represented in the following pages is in a tabular form, as follows:

*node_name annotation above_nodes below_nodes,*

where *node_name* is the name of the node, *annotation* is its value according to the function under analysis, and *above_nodes* and *below_nodes* are the nodes above and below it in the lattice, respectively. If *annotation* is itself a lattice, then that lattice is reproduced at that point, separated from previous and following text by asterisks. Also, note that node names are not significant, but are merely labels created at the moment of printing for purposes of cross-reference.

As for the notations used in the functional representations, some of them have been mentioned previously, but not all. Thus, their meanings are as follows:

| Notation | Meaning |
|---|---|
| ^ | λ |
| $n | variable name (generated internally) |
| v(N,T) | variable of name N and type T |
| l(V,E) | λ V . E |
| c(V,T) | constant of value V and type T |

## A.1. Examples of Strictness Analysis

### A.1.1. *Double* function

```
Script started on Thu Jul  7 15:47:41 1988
grads(1)% pp
C-Prolog version 1.5
| ?- [main].
main consulted 142368 bytes 7.28333 sec.

yes
| ?- ft4(8).
```

---

Reordered Semantics

```
letrec([(v(f,(int,int)),l(v($1,int),@(@(v(mul,(int,int,int)),v($1,int)),c(2,int))))])
```

---

Reprinted Text

```
letrec
      f = ^ $1 . mul $1 2
```

---

Abstract Expression

```
letrec([(v(f,(2,2)),l(v($1,2),@(@(v(glb_a,(2,2,2)),v($1,2)),c(1,2))))])
```

---

Reduced Abstract Expression

```
letrec([(v(f,(2,2)),l(v($1,2),v($1,2)))])
```

---

Pretty Reduced Abstract Expression

```
letrec
      f = ^ $1 . $1
```

---

Previous Definitions

```
letrec
      f = ^ $2 . 0
```

---

Latest Definitions

```
letrec
      f = ^ $1 . $1
```

Only 2 out of 2 possible evaluations actually performed!

---

First Function Lattice

```
x1     c(0,2)  []     [x2]
x2     c(0,2)  [x1]   []
```

Only 2 out of 2 possible evaluations actually performed!

Second Function Lattice

```
x3     c(1,2)  []     [x4]
x4     c(0,2)  [x3]   []
```

---

Fixed-Point Expression

```
letrec([(v(f,(2,2)),l(v($1,2),v($1,2)))])
```

---

Pretty Fixed-Point Expression

```
letrec
      f = ^ $1 . $1


yes
| ?- halt.

[ Prolog execution halted ]
grads(2)% exit
grads(3)%
script done on Thu Jul  7 15:48:06 1988
```

## A.1.2. *Factorial* function

```
Script started on Thu Jul  7 15:48:12 1988
grads(1)% pp
C-Prolog version 1.5
| ?- [main].
main consulted 142368 bytes 7.2 sec.

yes
| ?- ft4(9).
```

---

Reordered Semantics

```
letrec([(v(f,(int,int)),l(v($1,int),@(@(@(v(if,(bool,int,int,int)),@(@(v(eq,(int,int,int)),v($1,int)),c(0,int))),c(
1,int)),@(@(v(mul,(int,int,int)),v($1,int)),@(v(f,(int,int)),@(@(v(sub,(int,int,int)),v($1,int)),c(1,int)))))))])
```

---

Reprinted Text

```
letrec
      f = ^ $1 . if (eq $1 0) 1 (mul $1 (f (sub $1 1)))
```

---

Abstract Expression

```
letrec([(v(f,(2,2)),l(v($1,2),@(@(@(v(if_a,(2,2,2,2)),@(@(v(glb_a,(2,2,2)),v($1,2)),c(1,2))),c(1,2)),@(@(v(
glb_a,(2,2,2)),v($1,2)),@(v(f,(2,2)),@(@(v(glb_a,(2,2,2)),v($1,2)),c(1,2)))))))])
```

---

Reduced Abstract Expression

```
letrec([(v(f,(2,2)),l(v($1,2),v($1,2)))])
```

---

Pretty Reduced Abstract Expression

```
letrec
      f = ^ $1 . $1
```

---

Previous Definitions

```
letrec
      f = ^ $5 . 0
```

---

Latest Definitions

```
letrec
```

```
    f = ^ $1 . $1


Only 2 out of 2 possible evaluations actually performed!
```

---

First Function Lattice

```
x1    c(0,2)  []    [x2]
x2    c(0,2)  [x1]  []

Only 2 out of 2 possible evaluations actually performed!
```

Second Function Lattice

```
x3    c(1,2)  []    [x4]
x4    c(0,2)  [x3]  []
```

---

Fixed-Point Expression

```
letrec([(v(f,(2,2)),l(v($1,2),v($1,2)))])
```

---

Pretty Fixed-Point Expression

```
letrec
    f = ^ $1 . $1


yes
| ?- halt.

[ Prolog execution halted ]
grads(2)% exit
grads(3)%
script done on Thu Jul  7 15:48:45 1988
```

## A.1.3. *Accumulating factorial* function

```
Script started on Thu Jul  7 15:49:58 1988
grads(1)% pp
C-Prolog version 1.5
| ?- [main].
main consulted 142368 bytes 7.26667 sec.

yes
| ?- ft4(112).
```

---

Reordered Semantics

```
letrec([(v(f,(int,int,int)),l(v($1,int),l(v($2,int),@(@(@(v(if,(bool,int,int,int)),@(@(v(eq,(int,int,bool)),v($1,
int)),c(0,int))),v($2,int)),@(@(v(f,(int,int,int)),@(@(v(sub,(int,int,int)),v($1,int)),c(1,int))),@(@(v(mul,(int
,int,int)),v($1,int)),v($2,int)))))))])
```

---

Reprinted Text

```
letrec
     f = ^ $1 . ^ $2 . if (eq $1 0) $2 (f (sub $1 1) (mul $1 $2))
```

---

Abstract Expression

```
letrec([(v(f,(2,2,2)),l(v($1,2),l(v($2,2),@(@(@(v(if_a,(2,2,2,2)),@(@(v(glb_a,(2,2,2)),v($1,2)),c(1,2))),v($2
,2)),@(@(v(f,(2,2,2)),@(@(v(glb_a,(2,2,2)),v($1,2)),c(1,2))),@(@(v(glb_a,(2,2,2)),v($1,2)),v($2,2)))))))])
```

---

Reduced Abstract Expression

```
letrec([(v(f,(2,2,2)),l(v($1,2),l(v($2,2),@(@(v(glb_a,(2,2,2)),v($1,2)),@(@(v(lub_a,(2,2,2)),v($2,2)),@(@(v
(f,(2,2,2)),v($1,2)),@(@(v(glb_a,(2,2,2)),v($1,2)),v($2,2)))))))])
```

---

Pretty Reduced Abstract Expression

```
letrec
     f = ^ $1 . ^ $2 . glb_a $1 (lub_a $2 (f $1 (glb_a $1 $2)))
```

---

Previous Definitions

```
letrec
     f = ^ $6 . ^ $7 . 0
```

---

Latest Definitions

```
letrec
     f = ^ $1 . ^ $2 . glb_a $1 $2
```

Only 2 out of 2 possible evaluations actually performed!

Only 4 out of 4 possible evaluations actually performed!

---

First Function Lattice

```
x1:
*****
x3    c(0,2) []    [x4]
x4    c(0,2) [x3]  []
*****
                       []    [x2]
x2:
*****
x5    c(0,2) []    [x6]
x6    c(0,2) [x5]  []
*****
                       [x1]  []
```

Only 2 out of 2 possible evaluations actually performed!

Only 2 out of 2 possible evaluations actually performed!

Only 3 out of 3 possible evaluations actually performed!

Second Function Lattice

```
x7:
*****
x9    c(1,2) []    [x10]
x10   c(0,2) [x9]  []
*****
                       []    [x8]
x8:
*****
x11   c(0,2) []    [x12]
x12   c(0,2) [x11] []
*****
                       [x7]  []
```

---

Previous Definitions

```
letrec
     f = ^ $1 . ^ $2 . glb_a $1 $2
```

---

Latest Definitions

```
letrec
     f = ^ $1 . ^ $2 . glb_a $1 (lub_a $2 (glb_a $1 (glb_a $1 $2)))
```

---

First Function Lattice

```
x13:
*****
x15   c(1,2) []    [x16]
x16   c(0,2) [x15] []
*****
                     []    [x14]
x14:
*****
x17   c(0,2) []    [x18]
x18   c(0,2) [x17] []
*****
                     [x13] []
```

Only 2 out of 2 possible evaluations actually performed!

Only 4 out of 4 possible evaluations actually performed!

Second Function Lattice

```
x19:
*****
x21   c(1,2) []    [x22]
x22   c(0,2) [x21] []
*****
                     []    [x20]
x20:
*****
x23   c(0,2) []    [x24]
x24   c(0,2) [x23] []
*****
                     [x19] []
```

They are equal

---

Fixed-Point Expression

```
letrec([(v(f,(2,2,2)),l(v($1,2),l(v($2,2),@(@(v(glb_a,(2,2,2)),v($1,2)),v($2,2)))))])
```

---

Pretty Fixed-Point Expression

```
letrec
     f = ^ $1 . ^ $2 . glb_a $1 $2
```

```
yes
| ?- halt.

[ Prolog execution halted ]
grads(2)% exit
grads(3)%
script done on Thu Jul  7 15:50:35 1988
```

## A.1.4. *Length* function

```
Script started on Thu Jul  7 15:51:04 1988
grads(1)% pp
C-Prolog version 1.5
| ?- [main].
main consulted 142368 bytes 7.21666 sec.

yes
| ?- ft4(14).
```

---

Reordered Semantics

```
letrec([(v(length,(list,int)),l(v($1,list),@(@(@(v(if_null,(list,int,int,int)),v($1,list)),c(0,int)),@(@(v(add,(int
,int,int)),c(1,int)),@(v(length,(list,int)),@(v(tl,(list,list)),v($1,list)))))))])
```

---

Reprinted Text

```
letrec
      length = ^ $1 . if_null $1 0 (add 1 (length (tl $1)))
```

---

Abstract Expression

```
letrec([(v(length,(4,2)),l(v($1,4),@(@(@(v(if_null_a,(4,2,2,2)),v($1,4)),c(1,2)),@(@(v(glb_a,(2,2,2)),c(1,2)
),@(v(length,(4,2)),@(v(tl_a,(4,4)),v($1,4)))))))])
```

---

Reduced Abstract Expression

```
letrec([(v(length,(4,2)),l(v($1,4),@(@(@(v(if_null_a,(4,2,2,2)),v($1,4)),c(1,2)),@(v(length,(4,2)),@(v(tl_a,(
4,4)),v($1,4))))))])
```

---

Pretty Reduced Abstract Expression

```
letrec
      length = ^ $1 . if_null_a $1 1 (length (tl_a $1))
```

---

Previous Definitions

```
letrec
      length = ^ $2 . 0
```

---

Latest Definitions

```
letrec
    length = ^ $1 . if_null_a $1 1 0
```

Only 2 out of 4 possible evaluations actually performed!

---

First Function Lattice

```
x1    c(0,2)  []    [x2]
x2    c(0,2)  [x1]  [x3]
x3    c(0,2)  [x2]  [x4]
x4    c(0,2)  [x3]  []
```

Only 3 out of 4 possible evaluations actually performed!

Second Function Lattice

```
x5    c(1,2)  []    [x6]
x6    c(0,2)  [x5]  [x7]
x7    c(0,2)  [x6]  [x8]
x8    c(0,2)  [x7]  []
```

---

Previous Definitions

```
letrec
    length = ^ $1 . if_null_a $1 1 0
```

---

Latest Definitions

```
letrec
    length = ^ $1 . if_null_a $1 1 (if_null_a (tl_a $1) 1 0)
```

---

First Function Lattice

```
x9    c(1,2)  []    [x10]
x10   c(0,2)  [x9]  [x11]
x11   c(0,2)  [x10] [x12]
x12   c(0,2)  [x11] []
```

Only 4 out of 4 possible evaluations actually performed!

Second Function Lattice

```
x13   c(1,2)  []    [x14]
x14   c(1,2)  [x13] [x15]
x15   c(0,2)  [x14] [x16]
x16   c(0,2)  [x15] []
```

---

Previous Definitions

```
letrec
      length = ^ $1 . if_null_a $1 1 (if_null_a (tl_a $1) 1 0)
```

---

Latest Definitions

```
letrec
      length = ^ $1 . if_null_a $1 1 (if_null_a (tl_a $1) 1 (if_null_a (tl_a (tl_a $1)) 1 0))
```

---

First Function Lattice

```
x17    c(1,2)  []     [x18]
x18    c(1,2)  [x17]  [x19]
x19    c(0,2)  [x18]  [x20]
x20    c(0,2)  [x19]  []
```

Only 4 out of 4 possible evaluations actually performed!

Second Function Lattice

```
x21    c(1,2)  []     [x22]
x22    c(1,2)  [x21]  [x23]
x23    c(0,2)  [x22]  [x24]
x24    c(0,2)  [x23]  []
```

They are equal

---

Fixed-Point Expression

```
letrec([(v(length,(4,2)),l(v($1,4),@(@(@(v(if_null_a,(4,2,2,2)),v($1,4)),c(1,2)),@(@(@(v(if_null_a,(4,2,2,2
)),@(v(tl_a,(4,4)),v($1,4))),c(1,2)),c(0,2)))))])
```

---

Pretty Fixed-Point Expression

```
letrec
      length = ^ $1 . if_null_a $1 1 (if_null_a (tl_a $1) 1 0)


yes
| ?- halt.

[ Prolog execution halted ]
grads(2)% exit
grads(3)%
```

```
script done on Thu Jul  7 15:51:43 1988
```

## A.1.5. *Sum* function

```
Script started on Thu Jul  7 15:51:51 1988
grads(1)% pp
C-Prolog version 1.5
| ?- [main].
main consulted 142368 bytes 7.35 sec.

yes
| ?- ft4(15).
```

---

Reordered Semantics

```
letrec([(v(sumlist,(list,int)),l(v($1,list),@(@(@(v(if_null,(list,int,int,int)),v($1,list)),c(0,int)),@(@(v(add,(i
nt,int,int)),@(v(hd,(list,int)),v($1,list))),@(v(sumlist,(list,int)),@(v(tl,(list,list)),v($1,list)))))))])
```

---

Reprinted Text

```
letrec
      sumlist = ^ $1 . if_null $1 0 (add (hd $1) (sumlist (tl $1)))
```

---

Abstract Expression

```
letrec([(v(sumlist,(4,2)),l(v($1,4),@(@(@(v(if_null_a,(4,2,2,2)),v($1,4)),c(1,2)),@(@(v(glb_a,(2,2,2)),@(v(
hd_a,(4,2)),v($1,4))),@(v(sumlist,(4,2)),@(v(tl_a,(4,4)),v($1,4)))))))])
```

---

Reduced Abstract Expression

```
letrec([(v(sumlist,(4,2)),l(v($1,4),@(@(@(v(if_null_a,(4,2,2,2)),v($1,4)),c(1,2)),@(@(v(glb_a,(2,2,2)),@(v(
hd_a,(4,2)),v($1,4))),@(v(sumlist,(4,2)),@(v(tl_a,(4,4)),v($1,4)))))))])
```

---

Pretty Reduced Abstract Expression

```
letrec
      sumlist = ^ $1 . if_null_a $1 1 (glb_a (hd_a $1) (sumlist (tl_a $1)))
```

---

Previous Definitions

```
letrec
      sumlist = ^ $2 . 0
```

---

Latest Definitions

```
letrec
      sumlist = ^ $1 . if_null_a $1 1 0
```

Only 2 out of 4 possible evaluations actually performed!

---

First Function Lattice

```
x1    c(0,2) []     [x2]
x2    c(0,2) [x1]   [x3]
x3    c(0,2) [x2]   [x4]
x4    c(0,2) [x3]   []
```

Only 3 out of 4 possible evaluations actually performed!

Second Function Lattice

```
x5    c(1,2) []     [x6]
x6    c(0,2) [x5]   [x7]
x7    c(0,2) [x6]   [x8]
x8    c(0,2) [x7]   []
```

---

Previous Definitions

```
letrec
      sumlist = ^ $1 . if_null_a $1 1 0
```

---

Latest Definitions

```
letrec
      sumlist = ^ $1 . if_null_a $1 1 (glb_a (hd_a $1) (if_null_a (tl_a $1) 1 0))
```

---

First Function Lattice

```
x9    c(1,2) []     [x10]
x10   c(0,2) [x9]   [x11]
x11   c(0,2) [x10]  [x12]
x12   c(0,2) [x11]  []
```

Only 4 out of 4 possible evaluations actually performed!

Second Function Lattice

```
x13   c(1,2) []     [x14]
x14   c(1,2) [x13]  [x15]
x15   c(0,2) [x14]  [x16]
x16   c(0,2) [x15]  []
```

---

Previous Definitions

```
letrec
    sumlist = ^ $1 . if_null_a $1 1 (glb_a (hd_a $1) (if_null_a (tl_a $1) 1 0))
```

---

Latest Definitions

```
letrec
    sumlist = ^ $1 . if_null_a $1 1 (glb_a (hd_a $1) (if_null_a (tl_a $1) 1 (glb_a (hd_a (tl_a $1))
(if_null_a (tl_a (tl_a $1)) 1 0))))
```

---

First Function Lattice

```
x17   c(1,2)  []    [x18]
x18   c(1,2)  [x17]  [x19]
x19   c(0,2)  [x18]  [x20]
x20   c(0,2)  [x19]  []
```

Only 4 out of 4 possible evaluations actually performed!

Second Function Lattice

```
x21   c(1,2)  []    [x22]
x22   c(1,2)  [x21]  [x23]
x23   c(0,2)  [x22]  [x24]
x24   c(0,2)  [x23]  []
```

They are equal

---

Fixed-Point Expression

```
letrec([(v(sumlist,(4,2)),l(v($1,4),@(@(@(v(if_null_a,(4,2,2,2)),v($1,4)),c(1,2)),@(@(v(glb_a,(2,2,2)),@(v(
hd_a,(4,2)),v($1,4))),@(@(@(v(if_null_a,(4,2,2,2)),@(v(tl_a,(4,4)),v($1,4))),c(1,2)),c(0,2)))))))])
```

---

Pretty Fixed-Point Expression

```
letrec
    sumlist = ^ $1 . if_null_a $1 1 (glb_a (hd_a $1) (if_null_a (tl_a $1) 1 0))
```

```
yes
| ?- halt.

[ Prolog execution halted ]
grads(2)% exit
```

```
grads(3)%
script done on Thu Jul  7 15:52:40 1988
```

## A.2. Examples of Termination Analysis

### A.2.1. *Double* function

```
Script started on Thu Jul  7 15:31:14 1988
grads(1)% pp
C-Prolog version 1.5
| ?- [main].
main consulted 141132 bytes 7.68333 sec.

yes
| ?- ft34(8).
```

---

Reordered Semantics

```
letrec([(v(f,(int,int)),l(v($1,int),@(@(v(mul,(int,int,int)),v($1,int)),c(2,int))))])
```

---

Reprinted Text

```
letrec
      f = ^ $1 . mul $1 2
```

---

Abstract Expression

```
letrec([(v(f,(3,3)),l(v($1,3),@(@(v(f_a,(3,3,3)),v($1,3)),c(1,3))))])
```

---

Reduced Abstract Expression

```
letrec([(v(f,(3,3)),l(v($1,3),@(@(@(v(if,(bool,3,3,3)),@(@(v(eq,(3,3,bool)),@(@(v(glb_a,(3,3,3)),v($1,3)),c
(1,3))),c(1,3))),@(@(v(lub_a,(3,3,3)),v($1,3)),c(1,3))),@(@(v(glb_a,(3,3,3)),v($1,3)),c(1,3)))))])
```

---

Pretty Reduced Abstract Expression

```
letrec
      f = ^ $1 . if (eq (glb_a $1 1) 1) (lub_a $1 1) (glb_a $1 1)
```

---

Previous Definitions

```
letrec
      f = ^ $4 . 0
```

---

Latest Definitions

```
letrec
     f = ^ $1 . if (eq (glb_a $1 1) 1) (lub_a $1 1) (glb_a $1 1)
```

Only 2 out of 3 possible evaluations actually performed!

---

First Function Lattice

```
x1    c(0,3)  []    [x2]
x2    c(0,3)  [x1]  [x3]
x3    c(0,3)  [x2]  []
```

Only 3 out of 3 possible evaluations actually performed!

Second Function Lattice

```
x4    c(?,3)  []    [x5]
x5    c(1,3)  [x4]  [x6]
x6    c(0,3)  [x5]  []
```

---

Fixed-Point Expression

```
letrec([(v(f,(3,3)),l(v($1,3),@(@(@(v(if,(bool,3,3,3)),@(@(v(eq,(3,3,bool)),@(@(v(glb_a,(3,3,3)),v($1,3)),c
(1,3))),c(1,3))),@(@(v(lub_a,(3,3,3)),v($1,3)),c(1,3))),@(@(v(glb_a,(3,3,3)),v($1,3)),c(1,3)))))])
```

---

Pretty Fixed-Point Expression

```
letrec
     f = ^ $1 . if (eq (glb_a $1 1) 1) (lub_a $1 1) (glb_a $1 1)
```

```
yes
| ?- halt.

[ Prolog execution halted ]
grads(2)% exit
grads(3)%
script done on Thu Jul  7 15:32:06 1988
```

## A.2.2. *Factorial* function

Script started on Thu Jul 7 15:04:29 1988
grads(1)% pp
C-Prolog version 1.5
| ?- [main].
main consulted 141132 bytes 7.21666 sec.

yes
| ?- ft34(9).

---

Reordered Semantics

letrec([(v(f,(int,int)),l(v($1,int),@(@(@(v(if,(bool,int,int,int)),@(@(v(eq,(int,int,int)),v($1,int)),c(0,int))),c(
1,int)),@(@(v(mul,(int,int,int)),v($1,int)),@(v(f,(int,int)),@(@(v(sub,(int,int,int)),v($1,int)),c(1,int)))))))])

---

Reprinted Text

```
letrec
    f = ^ $1 . if (eq $1 0) 1 (mul $1 (f (sub $1 1)))
```

---

Abstract Expression

letrec([(v(f,(3,3)),l(v($1,3),@(@(@(v(if_a,(3,3,3,3)),@(@(v(f_a,(3,3,3)),v($1,3)),c(1,3))),c(1,3)),@(@(v(f_a
,(3,3,3)),v($1,3)),@(v(f,(3,3)),@(@(v(f_a,(3,3,3)),v($1,3)),c(1,3)))))))])

---

Reduced Abstract Expression

letrec([(v(f,(3,3)),l(v($1,3),@(@(@(v(if,(bool,3,3,3)),@(@(v(eq,(3,3,bool)),@(@(v(glb_a,(3,3,3)),@(@(@(
v(if,(bool,3,3,3)),@(@(v(eq,(3,3,bool)),@(@(v(glb_a,(3,3,3)),v($1,3)),c(1,3))),c(1,3))),@(@(v(lub_a,(3,3,3
)),v($1,3)),c(1,3))),@(@(v(glb_a,(3,3,3)),v($1,3)),c(1,3)))),@(@(v(lub_a,(3,3,3)),c(1,3)),@(@(@(v(if,(bool,
3,3,3)),@(@(v(eq,(3,3,bool)),@(@(v(glb_a,(3,3,3)),v($1,3)),@(v(f,(3,3)),@(@(@(v(if,(bool,3,3,3)),@(@(v
(eq,(3,3,bool)),@(@(v(glb_a,(3,3,3)),v($1,3)),c(1,3))),c(1,3))),@(@(v(lub_a,(3,3,3)),v($1,3)),c(1,3))),@(@(
v(glb_a,(3,3,3)),v($1,3)),c(1,3)))))),c(1,3))),@(@(v(lub_a,(3,3,3)),v($1,3)),@(v(f,(3,3)),@(@(@(v(if,(bool,3
,3,3)),@(@(v(eq,(3,3,bool)),@(@(v(glb_a,(3,3,3)),v($1,3)),c(1,3))),c(1,3))),@(@(v(lub_a,(3,3,3)),v($1,3)),c
(1,3))),@(@(v(glb_a,(3,3,3)),v($1,3)),c(1,3)))))),@(@(v(glb_a,(3,3,3)),v($1,3)),@(v(f,(3,3)),@(@(@(v(if,(b
ool,3,3,3)),@(@(v(eq,(3,3,bool)),@(@(v(glb_a,(3,3,3)),v($1,3)),c(1,3))),c(1,3))),@(@(v(lub_a,(3,3,3)),v($1
,3)),c(1,3))),@(@(v(glb_a,(3,3,3)),v($1,3)),c(1,3)))))))),c(0,3))),c(0,3)),@(@(@(v(if,(bool,3,3,3)),@(@(v(a
nd,(bool,bool,bool)),@(@(v(eq,(3,3,bool)),@(@(@(v(if,(bool,3,3,3)),@(@(v(eq,(3,3,bool)),@(@(v(glb_a,(
3,3,3)),v($1,3)),c(1,3))),c(1,3))),@(@(v(lub_a,(3,3,3)),v($1,3)),c(1,3))),@(@(v(glb_a,(3,3,3)),v($1,3)),c(1,3
))))),c(1,3))),@(@(v(eq,(3,3,bool)),@(@(@(v(if,(bool,3,3,3)),@(@(v(eq,(3,3,bool)),@(@(v(glb_a,(3,3,3)),v(
$1,3)),@(v(f,(3,3)),@(@(@(v(if,(bool,3,3,3)),@(@(v(eq,(3,3,bool)),@(@(v(glb_a,(3,3,3)),v($1,3)),c(1,3))),
c(1,3))),@(@(v(lub_a,(3,3,3)),v($1,3)),c(1,3))),@(@(v(glb_a,(3,3,3)),v($1,3)),c(1,3)))))),c(1,3))),@(@(v(lub
_a,(3,3,3)),v($1,3)),@(v(f,(3,3)),@(@(@(v(if,(bool,3,3,3)),@(@(v(eq,(3,3,bool)),@(@(v(glb_a,(3,3,3)),v($
1,3)),c(1,3))),c(1,3))),@(@(v(lub_a,(3,3,3)),v($1,3)),c(1,3))),@(@(v(glb_a,(3,3,3)),v($1,3)),c(1,3)))))),@(@
(v(glb_a,(3,3,3)),v($1,3)),@(v(f,(3,3)),@(@(@(v(if,(bool,3,3,3)),@(@(v(eq,(3,3,bool)),@(@(v(glb_a,(3,3,3
)),v($1,3)),c(1,3))),c(1,3))),@(@(v(lub_a,(3,3,3)),v($1,3)),c(1,3))),@(@(v(glb_a,(3,3,3)),v($1,3)),c(1,3)))))))
,c(1,3)))),c(1,3)),c(?,3))))])

---

Pretty Reduced Abstract Expression

```
letrec
    f = ^ $1 . if (eq (glb_a (if (eq (glb_a $1 1) 1) (lub_a $1 1) (glb_a $1 1)) (lub_a 1 (if (eq (glb_a $1 (f (if
(eq (glb_a $1 1) 1) (lub_a $1 1) (glb_a $1 1)))) 1) (lub_a $1 (f (if (eq (glb_a $1 1) 1) (lub_a $1 1) (glb_a $1
1)))) (glb_a $1 (f (if (eq (glb_a $1 1) 1) (lub_a $1 1) (glb_a $1 1)))))))) 0) 0 (if (and (eq (if (eq (glb_a $1 1) 1)
(lub_a $1 1) (glb_a $1 1)) 1) (eq (if (eq (glb_a $1 (f (if (eq (glb_a $1 1) 1) (lub_a $1 1) (glb_a $1 1)))) 1)
(lub_a $1 (f (if (eq (glb_a $1 1) 1) (lub_a $1 1) (glb_a $1 1)))) (glb_a $1 (f (if (eq (glb_a $1 1) 1) (lub_a $1
1) (glb_a $1 1))))) 1)) 1 ?)
```

---

Previous Definitions

```
letrec
    f = ^ $25 . 0
```

---

Latest Definitions

```
letrec
    f = ^ $1 . if (eq (glb_a (if (eq (glb_a $1 1) 1) (lub_a $1 1) (glb_a $1 1)) 1) 0) 0 ?
```

Only 2 out of 3 possible evaluations actually performed!

---

First Function Lattice

```
x1    c(0,3)  []    [x2]
x2    c(0,3)  [x1]  [x3]
x3    c(0,3)  [x2]  []
```

Only 3 out of 3 possible evaluations actually performed!

Second Function Lattice

```
x4    c(?,3)  []    [x5]
x5    c(?,3)  [x4]  [x6]
x6    c(0,3)  [x5]  []
```

---

Previous Definitions

```
letrec
    f = ^ $1 . if (eq (glb_a (if (eq (glb_a $1 1) 1) (lub_a $1 1) (glb_a $1 1)) 1) 0) 0 ?
```

---

Latest Definitions

```
letrec
    f = ^ $1 . if (eq (glb_a (if (eq (glb_a $1 1) 1) (lub_a $1 1) (glb_a $1 1)) (lub_a 1 (if (eq (glb_a $1 (if
(eq (glb_a (if (eq (glb_a (if (eq (glb_a $1 1) 1) (lub_a $1 1) (glb_a $1 1)) 1) 1) (lub_a (if (eq (glb_a $1 1) 1)
(lub_a $1 1) (glb_a $1 1)) 1) (glb_a (if (eq (glb_a $1 1) 1) (lub_a $1 1) (glb_a $1 1)) 1)) 1) 0) 0 ?)) 1)
(lub_a $1 (if (eq (glb_a (if (eq (glb_a (if (eq (glb_a $1 1) 1) (lub_a $1 1) (glb_a $1 1)) 1) 1) (lub_a (if (eq
(glb_a $1 1) 1) (lub_a $1 1) (glb_a $1 1)) 1) (glb_a (if (eq (glb_a $1 1) 1) (lub_a $1 1) (glb_a $1 1)) 1)) 1)
0) 0 ?)) (glb_a $1 (if (eq (glb_a (if (eq (glb_a (if (eq (glb_a $1 1) 1) (lub_a $1 1) (glb_a $1 1)) 1) 1) (lub_a
(if (eq (glb_a $1 1) 1) (lub_a $1 1) (glb_a $1 1)) 1) (glb_a (if (eq (glb_a $1 1) 1) (lub_a $1 1) (glb_a $1 1))
1)) 1) 0) 0 ?))))) 0) 0 (if (and (eq (if (eq (glb_a $1 1) 1) (lub_a $1 1) (glb_a $1 1)) 1) (eq (if (eq (glb_a $1 (if
(eq (glb_a (if (eq (glb_a (if (eq (glb_a $1 1) 1) (lub_a $1 1) (glb_a $1 1)) 1) 1) (lub_a (if (eq (glb_a $1 1) 1)
(lub_a $1 1) (glb_a $1 1)) 1) (glb_a (if (eq (glb_a $1 1) 1) (lub_a $1 1) (glb_a $1 1)) 1)) 1) 0) 0 ?)) 1)
(lub_a $1 (if (eq (glb_a (if (eq (glb_a (if (eq (glb_a $1 1) 1) (lub_a $1 1) (glb_a $1 1)) 1) 1) (lub_a (if (eq
(glb_a $1 1) 1) (lub_a $1 1) (glb_a $1 1)) 1) (glb_a (if (eq (glb_a $1 1) 1) (lub_a $1 1) (glb_a $1 1)) 1)) 1)
0) 0 ?)) (glb_a $1 (if (eq (glb_a (if (eq (glb_a (if (eq (glb_a $1 1) 1) (lub_a $1 1) (glb_a $1 1)) 1) 1) (lub_a
(if (eq (glb_a $1 1) 1) (lub_a $1 1) (glb_a $1 1)) 1) (glb_a (if (eq (glb_a $1 1) 1) (lub_a $1 1) (glb_a $1 1))
1)) 1) 0) 0 ?))) 1)) 1 ?)
```

---

First Function Lattice

```
x7    c(?,3) []    [x8]
x8    c(?,3) [x7]  [x9]
x9    c(0,3) [x8]  []
```

Only 3 out of 3 possible evaluations actually performed!

Second Function Lattice

```
x10   c(?,3) []    [x11]
x11   c(?,3) [x10] [x12]
x12   c(0,3) [x11] []
```

They are equal

---

Fixed-Point Expression

```
letrec([(v(f,(3,3)),l(v($1,3),@(@(@(v(if,(bool,3,3,3)),@(@(v(eq,(3,3,bool)),@(@(v(glb_a,(3,3,3)),@(@(@(
v(if,(bool,3,3,3)),@(@(v(eq,(3,3,bool)),@(@(v(glb_a,(3,3,3)),v($1,3)),c(1,3))),c(1,3))),@(@(v(lub_a,(3,3,3
)),v($1,3)),c(1,3))),@(@(v(glb_a,(3,3,3)),v($1,3)),c(1,3)))),c(1,3))),c(0,3))),c(0,3)),c(?,3)))))])
```

---

Pretty Fixed-Point Expression

```
letrec
    f = ^ $1 . if (eq (glb_a (if (eq (glb_a $1 1) 1) (lub_a $1 1) (glb_a $1 1)) 1) 0) 0 ?
```

```
yes
| ?- halt.

[ Prolog execution halted ]
```

```
grads(2)% exit
grads(3)%
script done on Thu Jul  7 15:07:49 1988
```

## A.2.3. *Length* function

```
Script started on Thu Jul  7 15:08:13 1988
grads(1)% pp
C-Prolog version 1.5
| ?- [main].
main consulted 141132 bytes 7.21666 sec.

yes
| ?- ft34(14).
```

---

Reordered Semantics

```
letrec([(v(length,(list,int)),l(v($1,list),@(@(@(v(if_null,(list,int,int,int)),v($1,list)),c(0,int)),@(@(v(add,(int
,int,int)),c(1,int)),@(v(length,(list,int)),@(v(tl,(list,list)),v($1,list)))))))])
```

---

Reprinted Text

```
letrec
    length = ^ $1 . if_null $1 0 (add 1 (length (tl $1)))
```

---

Abstract Expression

```
letrec([(v(length,(4,3)),l(v($1,4),@(@(@(v(if_null_a,(4,3,3,3)),v($1,4)),c(1,3)),@(@(v(f_a,(3,3,3)),c(1,3)),
@(v(length,(4,3)),@(v(tl_a,(4,4)),v($1,4)))))))])
```

---

Reduced Abstract Expression

```
letrec([(v(length,(4,3)),l(v($1,4),@(@(@(v(if_null_a,(4,3,3,3)),v($1,4)),c(1,3)),@(@(@(v(if,(bool,3,3,3)),
@(@(v(eq,(3,3,bool)),@(@(v(glb_a,(3,3,3)),c(1,3)),@(v(length,(4,3)),@(v(tl_a,(4,4)),v($1,4))))),c(1,3))),@(
@(v(lub_a,(3,3,3)),c(1,3)),@(v(length,(4,3)),@(v(tl_a,(4,4)),v($1,4))))),@(@(v(glb_a,(3,3,3)),c(1,3)),@(v(l
ength,(4,3)),@(v(tl_a,(4,4)),v($1,4)))))))])
```

---

Pretty Reduced Abstract Expression

```
letrec
    length = ^ $1 . if_null_a $1 1 (if (eq (glb_a 1 (length (tl_a $1))) 1) (lub_a 1 (length (tl_a $1))) (glb_a
1 (length (tl_a $1))))
```

---

Previous Definitions

```
letrec
    length = ^ $4 . 0
```

---

Latest Definitions

letrec
    length = ^ $1 . if_null_a $1 1 0

Only 2 out of 4 possible evaluations actually performed!

---

First Function Lattice

x1  c(0,3) []  [x2]
x2  c(0,3) [x1]  [x3]
x3  c(0,3) [x2]  [x4]
x4  c(0,3) [x3]  []

Only 3 out of 4 possible evaluations actually performed!

Second Function Lattice

x5  c(?,3) []  [x6]
x6  c(0,3) [x5]  [x7]
x7  c(0,3) [x6]  [x8]
x8  c(0,3) [x7]  []

---

Previous Definitions

letrec
    length = ^ $1 . if_null_a $1 1 0

---

Latest Definitions

letrec
    length = ^ $1 . if_null_a $1 1 (if (eq (glb_a 1 (if_null_a (tl_a $1) 1 0)) 1) (lub_a 1 (if_null_a (tl_a $1) 1 0)) (glb_a 1 (if_null_a (tl_a $1) 1 0)))

---

First Function Lattice

x9  c(?,3) []  [x10]
x10  c(0,3) [x9]  [x11]
x11  c(0,3) [x10]  [x12]
x12  c(0,3) [x11]  []

Only 4 out of 4 possible evaluations actually performed!

Second Function Lattice

x13 c(?,3) [] [x14]
x14 c(?,3) [x13] [x15]
x15 c(0,3) [x14] [x16]
x16 c(0,3) [x15] []

---

Previous Definitions

letrec
length = ^ $1 . if_null_a $1 1 (if (eq (glb_a 1 (if_null_a (tl_a $1) 1 0)) 1) (lub_a 1 (if_null_a (tl_a $1) 1 0)) (glb_a 1 (if_null_a (tl_a $1) 1 0)))

---

Latest Definitions

letrec
length = ^ $1 . if_null_a $1 1 (if (eq (glb_a 1 (if_null_a (tl_a $1) 1 (if (eq (glb_a 1 (if_null_a (tl_a (tl_a $1)) 1 0)) 1) (lub_a 1 (if_null_a (tl_a (tl_a $1)) 1 0)) (glb_a 1 (if_null_a (tl_a (tl_a $1)) 1 0))))) 1) (lub_a 1 (if_null_a (tl_a $1) 1 (if (eq (glb_a 1 (if_null_a (tl_a (tl_a $1)) 1 0)) 1) (lub_a 1 (if_null_a (tl_a (tl_a $1)) 1 0)) (glb_a 1 (if_null_a (tl_a (tl_a $1)) 1 0))))) (glb_a 1 (if_null_a (tl_a $1) 1 (if (eq (glb_a 1 (if_null_a (tl_a (tl_a $1)) 1 0)) 1) (lub_a 1 (if_null_a (tl_a (tl_a $1)) 1 0)) (glb_a 1 (if_null_a (tl_a (tl_a $1)) 1 0))))))

---

First Function Lattice

x17 c(?,3) [] [x18]
x18 c(?,3) [x17] [x19]
x19 c(0,3) [x18] [x20]
x20 c(0,3) [x19] []

Only 4 out of 4 possible evaluations actually performed!

Second Function Lattice

x21 c(?,3) [] [x22]
x22 c(?,3) [x21] [x23]
x23 c(0,3) [x22] [x24]
x24 c(0,3) [x23] []

They are equal

---

Fixed-Point Expression

letrec([(v(length,(4,3)),l(v($1,4),@(@(@(v(if_null_a,(4,3,3,3)),v($1,4)),c(1,3)),@(@(@(v(if,(bool,3,3,3)),@(@(v(eq,(3,3,bool)),@(@(v(glb_a,(3,3,3)),c(1,3)),@(@(@(v(if_null_a,(4,3,3,3)),@(v(tl_a,(4,4)),v($1,4))),c(1,3)),c(0,3)))),c(1,3))),@(@(v(lub_a,(3,3,3)),c(1,3)),@(@(@(v(if_null_a,(4,3,3,3)),@(v(tl_a,(4,4)),v($1,4))),c(1,3)),c(0,3)))),@(@(v(glb_a,(3,3,3)),c(1,3)),@(@(@(v(if_null_a,(4,3,3,3)),@(v(tl_a,(4,4)),v($1,4))),c(1,3)),c(0,3)))))))])

---

Pretty Fixed-Point Expression

```
letrec
      length = ^ $1 . if_null_a $1 1 (if (eq (glb_a 1 (if_null_a (tl_a $1) 1 0)) 1) (lub_a 1 (if_null_a (tl_a $1)
1 0)) (glb_a 1 (if_null_a (tl_a $1) 1 0)))


yes
| ?- halt.

[ Prolog execution halted ]
grads(2)% exit
grads(3)%
script done on Thu Jul  7 15:26:10 1988
```

## A.2.4. *Sum* function

Script started on Thu Jul 7 15:26:21 1988
grads(1)% pp
C-Prolog version 1.5
| ?- [main].
main consulted 141132 bytes 7.76667 sec.

yes
| ?- ft34(15).

---

Reordered Semantics

```
letrec([(v(sumlist,(list,int)),l(v($1,list),@(@(@(v(if_null,(list,int,int,int)),v($1,list)),c(0,int)),@(@(v(add,(i
nt,int,int)),@(v(hd,(list,int)),v($1,list))),@(v(sumlist,(list,int)),@(v(tl,(list,list)),v($1,list)))))))])
```

---

Reprinted Text

```
letrec
      sumlist = ^ $1 . if_null $1 0 (add (hd $1) (sumlist (tl $1)))
```

---

Abstract Expression

```
letrec([(v(sumlist,(4,3)),l(v($1,4),@(@(@(v(if_null_a,(4,3,3,3)),v($1,4)),c(1,3)),@(@(v(f_a,(3,3,3)),@(v(hd
_a,(4,3)),v($1,4))),@(v(sumlist,(4,3)),@(v(tl_a,(4,4)),v($1,4)))))))])
```

---

Reduced Abstract Expression

```
letrec([(v(sumlist,(4,3)),l(v($1,4),@(@(@(v(if_null_a,(4,3,3,3)),v($1,4)),c(1,3)),@(@(@(v(if,(bool,3,3,3)),
@(@(v(eq,(3,3,bool)),@(@(v(glb_a,(3,3,3)),@(v(hd_a,(4,3)),v($1,4))),@(v(sumlist,(4,3)),@(v(tl_a,(4,4)),v(
$1,4)))))),c(1,3))),@(@(v(lub_a,(3,3,3)),@(v(hd_a,(4,3)),v($1,4))),@(v(sumlist,(4,3)),@(v(tl_a,(4,4)),v($1,4)
))))),@(@(v(glb_a,(3,3,3)),@(v(hd_a,(4,3)),v($1,4))),@(v(sumlist,(4,3)),@(v(tl_a,(4,4)),v($1,4)))))))])
```

---

Pretty Reduced Abstract Expression

```
letrec
      sumlist = ^ $1 . if_null_a $1 1 (if (eq (glb_a (hd_a $1) (sumlist (tl_a $1))) 1) (lub_a (hd_a $1)
(sumlist (tl_a $1))) (glb_a (hd_a $1) (sumlist (tl_a $1))))
```

---

Previous Definitions

```
letrec
      sumlist = ^ $4 . 0
```

---

Latest Definitions

```
letrec
      sumlist = ^ $1 . if_null_a $1 1 0
```

Only 2 out of 4 possible evaluations actually performed!

---

First Function Lattice

```
x1    c(0,3)  []    [x2]
x2    c(0,3)  [x1]  [x3]
x3    c(0,3)  [x2]  [x4]
x4    c(0,3)  [x3]  []
```

Only 3 out of 4 possible evaluations actually performed!

Second Function Lattice

```
x5    c(?,3)  []    [x6]
x6    c(0,3)  [x5]  [x7]
x7    c(0,3)  [x6]  [x8]
x8    c(0,3)  [x7]  []
```

---

Previous Definitions

```
letrec
      sumlist = ^ $1 . if_null_a $1 1 0
```

---

Latest Definitions

```
letrec
      sumlist = ^ $1 . if_null_a $1 1 (if (eq (glb_a (hd_a $1) (if_null_a (tl_a $1) 1 0)) 1) (lub_a (hd_a $1)
(if_null_a (tl_a $1) 1 0)) (glb_a (hd_a $1) (if_null_a (tl_a $1) 1 0)))
```

---

First Function Lattice

```
x9    c(?,3)  []    [x10]
x10   c(0,3)  [x9]  [x11]
x11   c(0,3)  [x10] [x12]
x12   c(0,3)  [x11] []
```

Only 4 out of 4 possible evaluations actually performed!

Second Function Lattice

```
x13   c(?,3) []    [x14]
x14   c(?,3) [x13]  [x15]
x15   c(0,3) [x14]  [x16]
x16   c(0,3) [x15]  []
```

---

Previous Definitions

```
letrec
    sumlist = ^ $1 . if_null_a $1 1 (if (eq (glb_a (hd_a $1) (if_null_a (tl_a $1) 1 0)) 1) (lub_a (hd_a $1)
(if_null_a (tl_a $1) 1 0)) (glb_a (hd_a $1) (if_null_a (tl_a $1) 1 0)))
```

---

Latest Definitions

```
letrec
    sumlist = ^ $1 . if_null_a $1 1 (if (eq (glb_a (hd_a $1) (if_null_a (tl_a $1) 1 (if (eq (glb_a (hd_a (tl_a
$1)) (if_null_a (tl_a (tl_a $1)) 1 0)) 1) (lub_a (hd_a (tl_a $1)) (if_null_a (tl_a (tl_a $1)) 1 0)) (glb_a (hd_a
(tl_a $1)) (if_null_a (tl_a (tl_a $1)) 1 0))))) 1) (lub_a (hd_a $1) (if_null_a (tl_a $1) 1 (if (eq (glb_a (hd_a
(tl_a $1)) (if_null_a (tl_a (tl_a $1)) 1 0)) 1) (lub_a (hd_a (tl_a $1)) (if_null_a (tl_a (tl_a $1)) 1 0)) (glb_a
(hd_a (tl_a $1)) (if_null_a (tl_a (tl_a $1)) 1 0))))) (glb_a (hd_a $1) (if_null_a (tl_a $1) 1 (if (eq (glb_a (hd_a
(tl_a $1)) (if_null_a (tl_a (tl_a $1)) 1 0)) 1) (lub_a (hd_a (tl_a $1)) (if_null_a (tl_a (tl_a $1)) 1 0)) (glb_a
(hd_a (tl_a $1)) (if_null_a (tl_a (tl_a $1)) 1 0))))))
```

---

First Function Lattice

```
x17   c(?,3) []    [x18]
x18   c(?,3) [x17]  [x19]
x19   c(0,3) [x18]  [x20]
x20   c(0,3) [x19]  []
```

Only 4 out of 4 possible evaluations actually performed!

Second Function Lattice

```
x21   c(?,3) []    [x22]
x22   c(?,3) [x21]  [x23]
x23   c(0,3) [x22]  [x24]
x24   c(0,3) [x23]  []
```

They are equal

---

Fixed-Point Expression

```
letrec([(v(sumlist,(4,3)),l(v($1,4),@(@(@(v(if_null_a,(4,3,3,3)),v($1,4)),c(1,3)),@(@(@(v(if,(bool,3,3,3)),
@(@(v(eq,(3,3,bool)),@(@(v(glb_a,(3,3,3)),@(v(hd_a,(4,3)),v($1,4))),@(@(@(v(if_null_a,(4,3,3,3)),@(v(t
l_a,(4,4)),v($1,4))),c(1,3)),c(0,3)))),c(1,3))),@(@(v(lub_a,(3,3,3)),@(v(hd_a,(4,3)),v($1,4))),@(@(@(v(if_n
ull_a,(4,3,3,3)),@(v(tl_a,(4,4)),v($1,4))),c(1,3)),c(0,3)))),@(@(v(glb_a,(3,3,3)),@(v(hd_a,(4,3)),v($1,4))),@
(@(@(v(if_null_a,(4,3,3,3)),@(v(tl_a,(4,4)),v($1,4))),c(1,3)),c(0,3)))))))])
```

---

Pretty Fixed-Point Expression

```
letrec
      sumlist = ^ $1 . if_null_a $1 1 (if (eq (glb_a (hd_a $1) (if_null_a (tl_a $1) 1 0)) 1) (lub_a (hd_a $1)
(if_null_a (tl_a $1) 1 0)) (glb_a (hd_a $1) (if_null_a (tl_a $1) 1 0)))


yes
| ?- halt.

[ Prolog execution halted ]
grads(2)% exit
grads(3)%
script done on Thu Jul  7 15:31:01 1988
```