Finding Fixed Points in Finite Lattices

Chris Martin

Chris Hankin

Department of Computing,

Imperial College of Science and Technology,

180 Queen's Gate, London SW7,

England.

ABSTRACT

Recently there has been much interest in the abstract interpretation of declarative languages. Abstract interpretation is a semantics-based approach to program analysis that uses compile time evaluation of programs using simplified value domains. This gives information about the run-time properties of programs and provides the basis for significant performance improvements. A particular example of abstract interpretation is strictness analysis which allows the detection of the parameters in which a function is strict; these parameters may be passed by value without compromising the termination properties of the program.

The central, most complex task of an abstract interpreter is finding the fixpoints of recursive functions in the abstract value space. An elegant algorithm, the frontiers algorithm, has been proposed by Simon Peyton-Jones and Chris Clack that performs very well for the strictness analysis of first-order functions. In this paper we extend their algorithm and show how it can be applied to higher-order functions over arbitrary finite lattices. This raises the possibility of using the algorithm as the basis for more general abstract interpretation tools. We describe the algorithm in a modular way that is conducive to proofs of correctness and termination properties.

1. Introduction

Abstract interpretation is a semantics-based method of finding properties of programs at compile time. A major example is strictness analysis, which involves analysing lazy functions to determine which parameters the functions are strict in. For these parameters, the more efficient call-by-value evaluation strategy can be used without affecting the semantics. (A unary function f is strict in its argument if

$$f \perp = \perp$$

with the obvious generalisation to functions of more than one argument.)

A large proportion of the functional language abstract interpretations described in the literature are defined over finite lattices. Strictness Analysis for example, uses the domain **2** ($\{0,1\}$), with 0 representing definite non-termination, and 1 representing possible termination. Arithmetic operators such as "+" and "*" that are strict in both their arguments become boolean **and** in the abstract domain, constants become 1, and the conditional becomes

predicate **and** (consequent **or** alternative)

For example: the "accumulating factorial" function

$$afact = \quad \lambda\, x\, y.\textbf{if}\ (x = 0)\ \textbf{then}\ y\ \textbf{else}\ afact\ (x - 1)\ (x * y)$$

becomes the abstract function:

$$\begin{aligned} afact^{\#} &= \quad \lambda\, x\, y.\ (x\ \textbf{and}\ 1)\ \textbf{and}\ (y\ \textbf{or}\ afact^{\#}\ (x\ \textbf{and}\ 1)\ (x\ \textbf{and}\ y)) \\ &= \quad \lambda\, x\, y.\ x\ \textbf{and}\ (y\ \textbf{or}\ afact^{\#}\ x\ (x\ \textbf{and}\ y)) \end{aligned}$$

The functions analysed are usually recursive and it is therefore necessary to find the fixpoints of abstract functions over finite lattices. Methods of finding fixpoints already exist for first-order functions over the two point domain.

Finding the fixpoint of a recursive function is exponential in the number of arguments and the size of the expression, so we cannot expect to find efficient algorithms in the general case. What we can aim for is an algorithm that works well with the "average" functions that occur in functional programming languages. [Clack85, Abramsky87] contain the background to the frontiers method and show how it is efficient for these "average" functions. We shall not repeat this but concentrate on giving a formal basis for the algorithms. There is much scope for efficient implementation as there are many places in which heuristics can be used to make "optimal" choices to increase the average case efficiency of the algorithm.

We shall outline an improved version of the frontiers algorithm of [Clack85] and we shall show how it can be extended to analyse functions that take functional arguments (higher-order functions). We shall then show how it can be extended further to include functions defined over arbitrary finite lattices. Finally we shall give a brief description of other work in this field, and other approaches that might be possible for finding fixpoints over general domains.

2. Definition of a Frontier

A function over a finite lattice can be represented by labelling all the possible arguments (represented as tuples) with the value of the function at that point. Consider the function

$f = \lambda xy.x$ **and** y

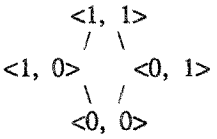
over the abstract domain **2**. Labelling all the possible arguments would give

<x, y>	f x y
<0, 0>	0
<1, 0>	0
<0, 1>	0
<1, 1>	1

We know, however, that there is an ordering on the domain of arguments to f (the argument domain: **A**), the pointwise ordering:

$$< x_1, x_2, \ldots x_n > \ \leq \ < y_1, y_2, \ldots y_n > \quad \textbf{iff} \quad x_i \leq y_i \ \forall i\colon 1 \leq i \leq n.$$

Therefore we have the following order on the argument domain **A**:



We also know that the functions are **monotonic:**

$$x \leq y \ \rightarrow f x \leq f y \qquad \forall x, y \in \mathbf{A}$$

This means that all the nodes in **A** labelled with a 1 (the "1-nodes") will be above all those nodes marked with a 0 (the "0-nodes"). We can use this information to construct a much more compact representation of the function: we need only store those nodes on the boundary between nodes labelled with different results.

Such sets of nodes on the boundary are called **frontiers.** Note that there are two choices for a frontier. We can either store the nodes on the "0" side of the boundary or the nodes on the "1" side of the boundary. We shall call the former **maximum-0-frontiers** and the latter **minimum-1-frontiers**, as they record the maximum 0-nodes, and minimum 1-nodes respectively.

Clearly none of the nodes in a frontier are comparable - a frontier contains the maximum (dually: minimum) nodes that evaluate to the same value - if any two elements were comparable, then one would not be a maximum (dually: minimum) node. We say that frontiers are **irredundant.**

Frontiers have the following properties:

The maximum-0-frontier of a function f from an argument domain A to a base domain **2**

$$f : A \rightarrow 2$$

is a set $F0 \in 2^A$ such that:

1. $\forall x \in A, f\ x = 0 \rightarrow \exists y \in F0 : x \leq y.$

   (All 0-nodes are less than at least one frontier node)

2. $\forall x \in F0, f\ x = 0.$

   (All the nodes on the frontier are 0-nodes)

3. $\forall x, y \in F0, x \leq y \rightarrow x = y.$

   (the frontier is irredundant)

A dual definition exists for the minimum-1-frontier.

There is a one-to-one correspondence between functions and maximum-0-frontiers (dually: minimum-1-frontiers) : each frontier defines a unique function, and vice-versa.

2.1. Definitions

The frontiers algorithm does not examine the structure of the functions under analysis, and we shall only assume the following:

Definition: a function f maps a point x in the argument domain **A** to a point r in the base domain **B**. We say that **"f x evaluates to r"** and that point x in the argument domain **A** is an **"r-node".**

For an n-ary function, each node X in the argument domain **A** will be a tuple :

$$X = < x_1, \ldots x_n >$$

Definition: We call the ith component of an argument node X the **"ith element"** of that node, and each element is in the **"element domain" E.**

Example: with first-order strictness analysis, the base domain **B** and the element domain **E** are both **2** ($\{0,1\}$), and the argument domain is the n-wise cartesian product of the element domain; **2** with itself. Moving to higher-order strictness analysis, the base domain **B** is still **2** but the element domain **E** may now contain functions over **2.**

Definition: We shall say that a point x in the argument domain **"is contained in"** a maximum frontier MAX-F

iff $\exists f \in MAX\text{-}F: x \leq f.$

Similarly, a point x **"is contained in"** a minimum frontier MIN-F

iff $\exists f \in MIN\text{-}F: x \geq f.$

2.2. Finding Fixpoints

In finding the fixpoint corresponding to a function f of type $(A \rightarrow B)$:

$$f = \lambda x_1, \ldots x_n.\text{body}$$

we actually need to find the fixpoint of the functional G of type $(A \rightarrow B) \rightarrow (A \rightarrow B)$, which is defined as

$$G = \lambda f, x_1, \ldots x_n.\text{body}$$

What we are looking for is the **least fixed point** (fixpoint) of the functional G, which will give us a function equal to f but without the recursion. The frontiers algorithm finds the fixpoint by iterating to find successive approximations to f, starting with the function that maps every argument to the bottom value of the base domain ($\perp$).

$$\begin{aligned} f_0 &= \lambda\, x_1, x_2, \ldots x_n.\, \perp \\ f_1 &= G\,(f_0) \\ f_2 &= G\,(f_1) \\ &\vdots \\ f_m &= G\,(f_{m-1}) \end{aligned}$$

This sequence (the Ascending Kleene Chain : AKC) clearly has a limit as the functions are monotonic and there are only a finite number of functions over any finite domain.

3. Finding Frontiers for First-Order Strictness Analysis

At each step in the AKC, we need to find the frontier representation of the next approximation. We can then compare this with that obtained for the previous iteration, and if they are equal we have found the fixpoint. The next approximation is the function f with the result of the previous iteration f_{n-1} used to evaluate any recursive calls to f.

3.1. Evaluating Recursive Calls

When we encounter a recursive call during evaluation of a function, the result is that which would be returned by the previous approximation. This approximation will be represented by a maximum-0-frontier, and to determine the value of the function it represents with the given argument, we test if the actual argument is contained in the frontier. If it is, then the argument is less than at least one of 0-nodes on the 0-frontier, and, by monotonicity, the function must return 0 at that point. Conversely, if the argument is not contained in the 0-frontier the function returns 1.

3.2. The Frontiers Algorithm

As in [Clack85] we search the argument domain in parallel from the top and the bottom picking arguments to evaluate alternately from the upwards and downwards searches.

The [Clack85] algorithm uses the upwards search to look for the minimum-1-frontier. When it encounters a 1-node, it adds that node to the minimum-1-frontier it is constructing. The downwards search similarly looks for the maximum-0-frontier. When the two searches meet in the middle, the upwards search needs to evaluate the lowest "1-nodes" to know it has reached the

frontier. Similarly, the downward search needs to evaluate the highest "0-nodes" to realise it has found it's goal. It is possible, therefore, that certain nodes in the middle of the argument domain are evaluated twice. The main objective of the algorithm is to reduce as much as possible the number of evaluations needed to determine the fixpoint, and so we should avoid evaluating nodes at points in the argument domain where we can determine the value of that node by other means.

Our algorithm is different in that it uses the upwards search to build up the 0-frontier - every time a 0-node is found by either search, it is added to the 0-frontier (with the appropriate checks to ensure irredundancy). Once the algorithm has terminated this frontier will be the required 0-frontier.

This frontier will always contain the maximum of the 0-nodes found so far, we can check any node picked for evaluation in the downward search against this frontier; if it contains the node, we know the point in the argument domain is a 0-node and need not evaluate it.

Dually, a similar test is carried out before evaluating any node picked from the upwards search with the minimum-1-frontier being constructed by the downwards search.

3.3. Searching The Argument Domain

Each search is represented by two sets. For the upwards search building up the 0-frontier, these are:

NewFr: This contains all the points that have evaluated to 0 so far. This is a maximum-0-frontier.

TrialFr: This is a minimum-1-frontier, which contains within it all the points that have yet to be evaluated during the upwards search that still might evaluate to 0. The elements of this set are the next values to be evaluated by the upwards search.

As the upwards search progresses, 0-nodes are added to NewFr, and TrialFr moves up the argument domain. Once TrialFr is empty, the whole domain has been searched and the algorithm terminates.

Notation: call the NewFr and TrialFr searching upwards building up the maximum-0-frontier "NewFr-0" and "TrialFr-0" respectively. Call their duals in the downward search "NewFr-1" and "TrialFr-1".

The algorithm for finding the frontiers then becomes:

```
TrialFr-0   = {<0, 0, ... 0> }
NewFr-0     = {}

TrialFr-1   = NewFr-0 from the previous iteration: f_{n-1} – initially {<1, 1, ... 1>}
NewFr-1     = NewFr-1 from the previous iteration: f_{n-1} – initially {}

while (TrialFr-0 ∪ TrialFr-1 ≠ {})

    if TrialFr-0 ≠ {}

        pick x from TrialFr-0

        if    CONTAINS-MIN (New-Fr1, x)
        then  result = 1
        else  evaluate f x to give result.
        endif

        Search-Upwards (x, result)
        Search-Downwards (x, result)

    endif

    if TrialFr-1 ≠ {}

        pick x from TrialFr-1

        if    CONTAINS-MAX (New-Fr0, x)
        then  result = 0
        else  evaluate f x to give result.
        endif

    Search-Upwards (x, result)
    Search-Downwards (x, result)

    endif
endwhile
```

3.4. The Search Actions

The "Search-Upwards" and "Search-Downwards" routines need to update the "NewFr" and "TrialFr" frontiers each time they are passed information about a point in the domain. The upwards search is the exact dual of the downwards search and so we will only consider the downwards instance.

The action taken by the downwards search on the receipt of a

(point, value)

pair, indicating that "point" in the argument domain evaluates to "value" in the base domain, is as follows:

```
Search-Downwards (point, value) :

    if (value = 1)
        NewFr1    :=  ADD-MIN (New-Fr1, {point})
        TrialFr1  :=  REMOVE-MAX (TrialFr1, point)
    else
        TrialFr1  :=  REMOVE-BELOW (TrialFr1, point)
    endif
```

3.5. Frontier Manipulation

We will now define the operations we have used in the above algorithm. Note that where a particular function (ADD-MIN for example) is defined, we shall also define it's dual (ADD-MAX) for the search in the other direction.

3.5.1. CONTAINS-MAX and CONTAINS-MIN

CONTAINS-MAX indicates whether a particular node is contained within a given MAX-FR:

$$\begin{aligned} &\text{CONTAINS-MAX} &&: \text{MAX-FR} \times \text{NODE} &&\rightarrow \quad \{\text{True, False}\} \\ &\text{CONTAINS-MIN} &&: \text{MAX-FR} \times \text{NODE} &&\rightarrow \quad \{\text{True, False}\} \end{aligned}$$

defined by:

$$\begin{aligned} \text{CONTAINS-MAX (Max-Fr, Node)} &= \text{True} \quad \textbf{iff} \quad \exists x \in \text{Max-Fr: Node} \leq x \\ &= \text{False} \quad \text{otherwise} \end{aligned}$$

where $\leq$ is the comparison operation on the argument domain **A**, which in turn is defined pointwise using $\leq$ on the element domain **E**. Similarly CONTAINS-MIN is defined using $\geq$.

3.5.2. ADD-MIN and ADD-MAX

We define a union operation on minimum-frontiers called:

$$\text{ADD-MIN} : \quad \text{MIN-FR} \times \text{MIN-FR} \rightarrow \text{MIN-FR}$$

returning the Union of the two argument frontiers, keeping only the minimum of any two comparable points (checking using CONTAINS-MIN). Clearly any point above either of the two component frontiers will also be above the union of them. The dual:

$$\text{ADD-MAX}: \quad \text{MAX-FR} \times \text{MAX-FR} \rightarrow \text{MAX-FR}$$

keeps the maximum of any two comparable points using CONTAINS-MAX.

Clearly

ADD-MIN (Min-Fr, {point})

returns the minimum-frontier "Min-Fr" but with "point" (and any points above it) as 1-nodes.

The dual, ADD-MAX (Max-Fr, {point}), returns the maximum-frontier "Max-Fr" but with any points below "point" as 0-nodes.

3.5.3. REMOVE-MAX and REMOVE-MIN

We use REMOVE-MAX to advance TrialFr-1 down the lattice when we encounter a 1-node:

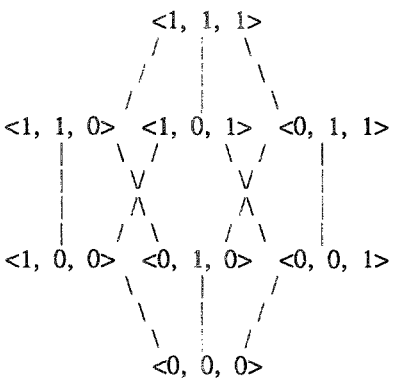
REMOVE-MAX : MAX-FR × Node → MAX-FR
REMOVE-MIN : MIN-FR × Node → MIN-FR

Then:

REMOVE-MAX (Max-Fr, point)

returns the maximum frontier "Max-Fr", but any nodes ≥ "point" become 1-nodes. Intuitively, we need to remove any points in "Max-Fr" that are ≥ "point" and replace them with the highest points below "Max-Fr" that are ≱ "point". Thus any points in the domain ≥ "point" would not be in the resulting frontier, but the highest points that are ≱ "point" would be.

Example: Consider functions over the domain **2** → **2** → **2** → **2** (functions with three base-domain arguments). The argument domain is:



Consider the 0-frontier Fr = {<110>, <101>, <011>}, and a point x: <001>. All the points in the frontier above <001>, that is <101> and <011>, need replacing with the highest points below them that are not above <001>. These points are <100> (below <101>) and <010> (below <011>):

Then Fr is the set of highest 0-nodes (with point "x" marked, the 0-Fr labelled with "*" and representing the domain just by the result labelling):

```
    1
0*  0*  0*
0   0   0 ← point "x".
    0
```

and the frontier we need to return is:

```
    1
0*  1   1
0   0*  1
    0
```

We require that any points above x in Fr are replaced by the highest points below Fr that are not above x. Assume we have a function

NOT-ABOVE : Node × Node → 2^{Node}

which carries out this operation pointwise, i.e.

NOT-ABOVE (x, y) = {g: g is below y but not above x}

then we can define

REMOVE-MAX : MAX-FR × Node → MAX-FR

such that

REMOVE-MAX (Max-Fr, Point)

is the union (using "ADD-MAX") of those points not above "point" in Max-Fr and

NOT-ABOVE (point, f_i)

for those points $f_i \in$ Max-Fr that are above x.

We can prove that [Martin86]

NOT-ABOVE ($<x_1, \ldots x_n>, <y_1, .. y_n>$)

is defined by the set

$\cup \{P_j\}$ $\quad 1 \leq j \leq n,$

where

$P_j = \{\}$ $\quad$ PREDS$(x_j) = \{\}$

$P_j = \cup \{G_{j,k}\}$ $\quad 1 \leq k \leq$ Card (PREDS (x_j))

for every i: $1 \leq i \leq n$, PREDS $(x_j) \neq \{\}$, $G_{j,k} = \{<g_1, g_2, \ldots g_n>\}$ is given by

$g_i = y_i$ $\quad i \neq j,$

$g_i \in$ PREDS (x_j) $\quad i = j.$

Note that that there as many values in the resulting set as there are in the Union of all the "PREDS (x_j)".

PREDS : Element → $2^{Element}$

is defined as the predecessor operation on the element domain, for **2** it is defined by:

PREDS (1) = {0}

PREDS (0) = {}

For higher-order functions "PREDS" potentially has non-singleton results.

The dual, "REMOVE-MIN" is defined in an exactly similar way, with dual types, but using "NOT-BELOW" and "ADD-MIN".

3.5.4. REMOVE-BELOW and REMOVE-ABOVE

REMOVE-BELOW removes all the nodes in a minimum-frontier that are below a particular point. We use this when a 0-node is found during the downwards search; clearly all the nodes below it must also be 0-nodes (by monotonicity) and so they can be removed from the search.

REMOVE-BELOW	:	MIN-FR	×	Node →	MIN-FR
REMOVE-ABOVE	:	MAX-FR	×	Node →	MAX-FR

This is implemented in the obvious way, testing each element in turn.

3.6. Basic Routines

Note that the only operations needed to prove (and implement) all these operations are

1. The compare operations ("≤" and "≥") on the element domain **E**.
   - for comparing two iterations for equality and implementing "CONTAINS-MAX" and "CONTAINS-MIN".
2. A "PREDS" (and it's dual, "SUCCS") operation on the element domain.
   - for the "NOT-ABOVE" operation (and it's dual: "NOT-BELOW")

This is useful when we come to extend the method to higher-order functions, as we only have to define these two operations to be able to use the same algorithm and proofs.

We have proved that [Martin86]

1. The algorithm is correct in that it does calculate the frontiers we are looking for.
2. It terminates in all cases.
   This proof relies on all the domains being finite, and shows the "TrialFr" sets are continually moving up this finite domain - they must eventually reach the top, and then the algorithm terminates.
3. Every node in the argument domain is evaluated in the worst case at most once.
   Normally we would expect better behaviour than this.

4. Strictness Analysis and Higher-Order Functions

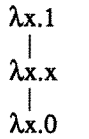
In introducing higher-order functions into the element domain **E**, we follow the approach taken by [Burn85, Hankin86], and we will be working with a strongly typed language. To analyse functions we must expand the element domain to include all the possible functions that can be passed as arguments.

Consider the function Apply of type $(\mathbf{2} \to \mathbf{2}) \to \mathbf{2} \to \mathbf{2}$:

```
Apply f x = f x
```

The element lattice for the first argument must contain all the possible functions of one

argument, ordered by monotonicity:



To use the frontiers algorithm, as we mentioned above, we need to include such functions over **2** in the element domain **E**, and therefore need to be able to define the following operations on these functions:

1. "≤" and "≥" - we need to be able to compare two elements for equality and approximation.
2. SUCCS and PREDS - given a member of the element domain we need to be able to find the elements that are above and below it.

4.1. A Representation for Functions in the Element Domain

We can use frontiers to represent the functions in the element domain, just as they were used to represent the approximations in the main algorithm.

Comparison of two frontiers is easily implemented; it just involves evaluating the node of each frontier in the function represented by the other frontier. For Maximum-frontiers:

$$F1 \leq F2 \quad \textbf{iff} \quad \forall x \in F2\text{: CONTAINS-MAX}(F1, x)$$

if $F1 \not\leq F2$ and $F2 \not\leq F1$, then the two functions are incomparable and:

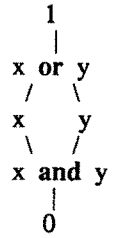
$$\exists x \in F1 \text{ : CONTAINS-MAX } (F2, x) \quad \textbf{and}$$
$$\exists y \in F2 \text{ : CONTAINS-MAX } (F1, y)$$

The operation on minimum-frontiers is the dual using CONTAINS-MIN.

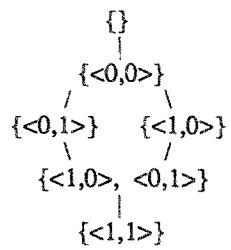
The SUCCS/PREDS operations are more complex, although successor is easy if we are using 0-maximum frontiers (dually, predecessor is easy if we are using 1-minimum frontiers)

Each successor to a maximum-0-frontier is obtained by adding the predecessors of one of the nodes in the original frontier back to this frontier.

For example: Consider the element domain, of type: $\mathbf{2} \rightarrow \mathbf{2} \rightarrow \mathbf{2}$ - the functions of two base-domain arguments, ordered by the usual function ordering and omitting the initial "λxy.":



If we replace each function with its corresponding maximum-0-frontier, we get:



Then we would expect

SUCCS (λxy.x **and** y) = {λxy.x, λxy.y}

using the frontiers representations:

SUCCS ({<1, 0>, <0, 1>}) = { {<1, 0>}, {<0, 1>} }

This set can be generated by taking each node in turn from the original frontier: For every node, there is an entry in the resulting set obtained by adding its predecessors to the original set (less the element itself) and keeping the maximum of any comparable elements.

Define

PREDECESSOR-NODE : Node → 2^{Node}

to return all the nodes directly below the node passed as arguments. This is the pointwise application of the base domain "PREDS" to each element in the node in turn. Note that PREDS still has the same type as with the first order case:

PREDS : Element → $2^{Element}$

but the element domain 2 now can contain frontiers itself.

Then

SUCCS ({<1, 0> , <0, 1>})

has two members, one is given by:

ADD-MAX ({<0, 1>}, PREDECESSOR-NODE (<1, 0>)) =
ADD-MAX ({<0, 1>}, {<0, 0>}) =
{<0, 1>}

Whenever moving up the element domain using maximum-0-frontiers, we have shown that each successor can be generated by applying "PREDECESSOR-NODE" to the nodes already there.

The operation:

PREDS (λxy.x)

which we expect to give the frontier representation of

λxy.x **and** y

is more difficult. Using the frontiers, we expect that

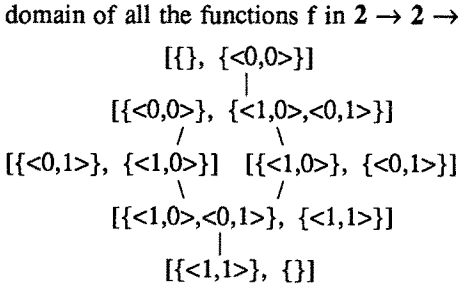
PREDS ({<0 ,1>}) = {<0, 1>, <1, 0>}

To carry out this operation it is necessary to add the element "<1, 0>" to the frontier, and there is no clear way of knowing when values like this have to be added to implement the operation without recalculating every node in the entire domain.

We are able to implement the "SUCCS" operation easily if we use maximum-0-frontiers to represent the functions in the element domain, and "PREDS" easily if we use a minimum-1-frontier representation. To implement the other operations (SUCCS with minimum-1-frontiers and PREDS with maximum-0-frontiers) we can use both the maximum-1 and the minimum-0 frontiers. A function in the element domain is now represented by

[maximum-0-frontier, minimum-1-frontier]

and the domain of all the functions f in $\mathbf{2} \rightarrow \mathbf{2} \rightarrow \mathbf{2}$ now is:



Consider the "PREDS" operation using this representation (the "SUCCS" operation is its dual). Each time we apply the operation to an argument [F0, F1], we need to return the maximum-0 and minimum-1 frontiers [NF0, NF1] for all the element(s) directly below [F0, F1] in the lattice.

NF1 is defined, as above, by taking each node in F1 in turn, and replacing it in F1 with its pointwise successors keeping the minimum elements. For each node, there is a separate NF1. We add the value removed from F1 to F0 to construct NF0, keeping the maximum of any comparable points.

For example: Consider the operation

PREDS ([{<1,0>}, {<0,1>}])

First we find any pointwise successors to the minimum-1-frontier using "SUCCESSOR-NODE", defined in an analogous way to "PREDECESSOR-NODE". We have that

SUCCESSOR-NODE (<0,1>) = {<1, 1>}

there is only one such successor, and so we have only one element in the result. The minimum-1-frontier in the result is given by replacing <0,1> by its successor: <1, 1>. Since there is only one element we do not have any comparable elements: if we did we would keep the minimum. To find the maximum-0-frontier in the result, we add the value we took from the minimum-1-frontier, <0, 1> to the old maximum-0-frontier, giving:

{<0, 1>, <1, 0>}

These two values are incomparable (if they were not we would keep the maximum). This gives the set

{ [{<0,1>, <1, 0>}, {<1, 1>}] }

as the result. Now consider it's dual, "SUCCS", moving up the function domain:

SUCCS ([{<0,1>, <1,0>}, {<1,1>}])

We first find the pointwise predecessors to the elements in the maximum-0-frontier and from these generate the new maximum-0-frontiers:

ADD-MAX ({<0, 1>}, {<0, 0>}) = {<0, 1>} (1)
ADD-MAX ({<1, 0>}, {<0, 0>}) = {<1, 0>} (2)

Consider the element in the result set given by (1) - the other is very similar. The new minimum-1-frontier is given by adding the value removed from the maximum-0-frontier set in (1): <1, 0>:

ADD-MIN ({<1, 1>}, {<1, 0>}) = {<1, 0>}

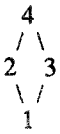
as required. We have proved in [Martin86] that this implementation of successor and predecessor in the functional domain does give the correct results.

5. General Finite Lattices

The ordering of any finite base domain is carried over, by monotonicity, to the labelling of the argument domain. In a three point chain domain, for example, all the 2-nodes must be above any 1-nodes, but below any 3-nodes.

A function over a lattice of n elements: a_1 to a_n, therefore, can be represented by n-1 maximum frontiers, one for each element of the base lattice. No frontier is needed for the top element because any node above all the other frontiers must be an a_n-node (in a complete lattice). Each maximum-a_m-frontier contains all the a_i-nodes contained in the argument lattice for all a_i : $a_i \leq a_m$. Every maximum-a_m-frontier is above the maximum frontier for any elements a_i where $a_i \leq a_m$, but below the maximum frontier for any elements a_j where $a_m \leq a_j$. Hence functions over general complete lattices can be represented using the frontiers technique.

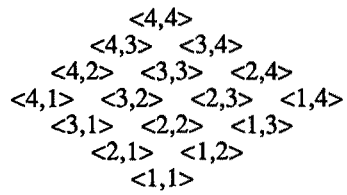
For example: 4 = {1, 2, 3, 4}, ordered by:



Consider a function f : 4 → 4 → 4 defined by:

$f = \lambda x_1\ x_2.x_1$

The argument lattice is :



and the maximum-frontiers are given by

max-1-frontier : {<1,4>}
max-2-frontier : {<2,4>}
max-3-frontier : {<3,4>}

These three frontiers completely describe the function f. To evaluate any point in the frontier, we find the smallest m, such that the m-frontier contains the argument, and return m for the result. If the argument is not contained in any of the frontiers, we return the top element in the base lattice.

Frontiers can be the same for different elements; Consider

λx y.3

where the frontiers are :

max-1-frontier : {}
max-2-frontier : {}
max-3-frontier : {<4,4>}

There are no elements that evaluate to either 1 or 2, and so both the frontiers are empty.

5.1. Implementing the Search Algorithm For Higher-Order Functions

To extend the algorithm to general lattices, we must extend both the element domain operations and the searching algorithm. The searching algorithm only requires two operations, comparison and successor/predecessor.

Two functions are compared by comparing all their corresponding frontiers: for one function to be less than another every corresponding frontier would have to be lower. If some frontiers are higher, and some are lower, then the two are incomparable.

The predecessors to a particular function are obtained by increasing any of the frontiers that can be increased. It must be ensured, however, that a lower frontier does not "overtake" a higher one: the maximum-a_i-frontier must never be greater than the maximum-a_j-frontier where $a_i \leq a_j$, or the function would not be monotonic.

For example: (represent a function by :[1-Fr, 2-Fr, 3-Fr]):

PREDS ([{<1,4>}, {<2,4>}, {<3,4>}]) =

{ [{<1,4>}, {<2,4>}, {<4,4>}], (increasing the 3-Fr)
[{<1,4>}, {<3,4>}, {<3,4>}], (increasing the 2-Fr)
[{<2,4>}, {<2,4>}, {<3,4>}] } (increasing the 1-Fr)

5.2. Searching the Argument Lattice

During the upward search of the argument lattice, we may encounter any a_m-node (unless we have already found the a_m frontier). Therefore we need to keep a "NewFr" set for every frontier we have yet to find. Each "a_m-TrialFr" contains all the possible nodes that could yet evaluate to a_m or below, and so there must be as many of these as the maximum width of the base domain: Imagine a lattice consisting of the coalesced sum of "n" chain domains: we need at least a separate Trial Frontier for each distinct chain domain.

Conceptually, we have a separate pair of search processes (one up, one down) for each frontier we are looking for. When a point in the argument lattice is evaluated, that information is passed to all the search processes. Initially all the "NewFr" and "TrialFr" sets, apart from TrialFr-Upwards-1 and TrialFr-Downwards-n, are empty. These two contain the bottom and top elements in the argument lattice.

On receipt of a "(point, val)" pair, there are three possible actions for each search process. Assume this is an upwards search building the maximum-m-frontier: then if

1. val $\leq$ m.

   TrialFr-m := REMOVE-MIN(TrialFr-m, point) ;
   NewFr-m := ADD-MAX (NewFr, {point}) ;

   Action: "point" should be added (keeping the highest points) to the "NewFr", and any successors to "point" should be added to the "TrialFr", keeping the smallest nodes after removing any nodes $\leq$ point.

2. val > m.

   TrialFr-m := REMOVE-ABOVE (TrialFr-m, point) ;

   Action: Any points in "TrialFr-m" that are $\geq$ point should be removed, as they could never evaluate to "m-nodes". Note that the successors to "point" will be added to the TrialFr of the search for a_{val}, as clearly the value can not evaluate to any of the elements between a_m and a_{val}.

3. val and m are incomparable.

   TrialFr-m := REMOVE-FROM-MIN (TrialFr-m, point);

   Action: If "point" $\in$ "TrialFr-m" then remove it and replace it with its successors in the argument lattice, otherwise do nothing (since the two elements are incomparable, we can deduce nothing about the m-frontiers from this result, the most we can do is ensure that "point" is not evaluated again).

As the algorithm progresses, the TrialFr's for the "lower" searches will empty as the frontiers they are looking for are found. When choosing a node to evaluate, we will choose a node in the lowest TrialFr that is non-empty. If there is more than one such node, then we can use heuristics, perhaps based on the structure of the function being analysed, to decide which node to use.

Initially only the trial frontier for the node at the bottom of the lattice contains any values. As the search progresses, and the search upwards for the a_i-frontier encounters nodes above a_i, nodes are taken from that trial frontier and added to the trial frontiers of the elements directly above a_i. The union of all the "trial frontier" sets of all the searches moves up the lattice, when a value is removed from one trial frontier, it is added to the search for the next element up in the base domain. The dual minimum-frontier search from the top is similar.

The "NewFr" sets in the searches are being built up all the time, however, as any search for a frontier can encounter a node evaluating to any value, and this must be stored in the "NewFr" set. Note that any time a node evaluates to a value a_i, then this is added to the NewFr for every search from a_1 to a_i. Hence even if no a_m nodes are encountered in the lattice, it will still have a frontier: the same as that for a_{m-1}.

The main algorithm for the search now becomes:

```
Initialise:
        All NewFr's and TrialFr's := {}
        TrialFr-Up-1      := {0, 0, .. 0}        bottom value in the argument lattice
        TrialFr-Down-n    := {1, 1, .. 1}        top value in the argument lattice

while (∪ TrialFr's ≠ {})

        pick a point from lowest non-empty TrialFr-Up
        Check point is not contained within the NewFr of any Downwards-NewFr
        evaluate f at that point to give result
        Call each upwards and downwards search with "(point, result)"

        Similar Downwards-search...

endwhile
```

As the number of incomparable elements in the base domain grows and the argument lattice to search becomes "less connected", the performance of the algorithm drops. It depends on using monotonicity information to remove the necessity to evaluate large parts of the argument lattice, and the less connected the argument lattice is, the less the information about the whole lattice that each point can indicate.

In the worst case, with two pointwise incomparable sub-lattice in the base lattice, evaluating a point in one will indicate no information about the other at all. With the coalesced sum of "n" lattices, the "TrialFr"'s will separate into "n" disjoint sets once the top and bottom values of each sub-lattice have been evaluated.

It can be seen that the searches adapt themselves to the structure of the lattice being searched, only looking in the part of the lattice where they could find nodes comparable with the frontier they are building.

6. Conclusion

We have presented a significant advance in the implementation of strictness analysers, following the advances of the theory in [Burn85]. The algorithms here are based on those given in [Clack85], but we have extended the method to include both higher-order functions and general finite lattices. [Martin86] gives correctness and termination proofs for the algorithms included here.

There are a number of places in the algorithm that we can make choices of which part of the argument lattice to explore next. We could use "heuristics", guided on the structure of the function being analysed, to explore parts of the lattice that give most information about the function first. This will further help the "average-case" performance of the algorithm.

There are other possible approaches to finding fixpoints for first-order functions over **2.** One interesting method is pending analysis [Young86] but there seems to be no obvious method of extending it to higher-order functions and more general domains. However the approach does give good results for first-order strictness analysis.

Another approach that has been pursued [Martin87] is using a syntactic method of finding fixpoints without evaluating the function at all. The problems in proving termination are avoided by a process of substitution that gives safe approximations to the fixpoint sought, while guaranteeing termination. This can be very efficient on some functions, and is generally applicable to higher-order functions and even infinite domains.

7. Acknowledgements

The authors would like to acknowledge the continued interest and encouragement from Sebastian Hunt, and his valuable comments on an earlier draft of this paper.

References

Abramsky87.

S. Abramsky and C. Hankin (Eds), *Abstract Interpretation of Declarative Languages,* Ellis Horwood, 1987.

Burn85.

G. Burn, C. Hankin, and S. Abramsky, *The Theory of Strictness Analysis for Higher Order Functions,* pp. 42-62, Springer Verlag LNCS 217, 1985.

Clack85.

C. Clack and S. Peyton-Jones, "Strictness Analysis - A Practical Approach," in *Functional Programming Languages and Computer Science*, pp. 35-49, Springer Verlag

LNCS 201, Nancy, France, September 1985.

Hankin86.

C. Hankin, G. Burn, and S. Peyton-Jones, *A Safe Approach to Parallel Combinator Reduction,* pp. 99-110, Springer Verlag LNCS 213, 1986.

Martin86.

C.C. Martin, *Extending the Frontiers Algorithm to Higher Order Functions and General Finite Domains,* Unpublished Document, Department Of Computing, Imperial College, 1986.

Martin87.

C.C. Martin, *Finding Fixpoints Using a Syntactic Method,* Unpublished Document, Department of Computing, Imperial College, 1987.

Young86.

J. Young and P. Hudak, *Finding Fixpoints on Function Spaces,* Research Report YALEEU/DCS/RR-505, Yale University Department of Computer Science, December 1986.