Finding fixed points in non-trivial domains: Proofs of pending analysis and related algorithms

Alan Dix

Full Reference

A. J. Dix (1988). *Finding fixed points in non-trivial domains: proofs of pending analysis and related algorithms*. YCS 107, Dept. of Computer Science, University of York.
`http://alandix.com/academic/papers/fixpts-YCS107-88/`

This version re-typeset March 2017

Abstract

Young and Hudak[1] developed an algorithm called *Pending Analysis* for deriving the fixed points of first order functions over binary domains. They prove a result in their paper which justifies the approach, but does not really prove its correctness. Further they suggest that the analysis may be extended to deeper domains. The structure of pending analysis means that many of the intermediate functions generated are not monotonic, which means that arguments of correctness based on partial orderings tend to fail without great care.

This paper attempts to fill some of the gaps in this work, we will:

1 Give proofs of correctness for the binary case when the defining functional is fully monotonic (i.e. monotonic even for non-monotonic function arguments). This is usually for *non-self applicatory functions*.

2 Develop a slight variant of the algorithm which makes better use of the monotonicity of the function, and which is correct for all functions.

3 Show how the binary analysis can be used to solve more complex domains if the primatives are sufficiently well behaved.

4 Parallel steps 1 and 2 for general domains using an iterative version of the algorithm. In particular proving several *incremental fixed point theorems*. We will also discus representation and algorithms for the partial functions used in this algorithm.

5 Show that pending analysis (and probably other algorithms too) can be extended to higher order functions by considering the *term algebra*.

6 Show that the concept of a fixed point can be extended to *configurations* (partial functions), and that this can be used to generate easy proofs of safeness for pending analysis and similar algorithms.

7 Use these ideas to generate a non-deterministic algorithm which is always correct, and which can be instantiated with different heuristics to yield optimised algorithms. This algorithm includes as special cases pending analysis (with and without memoing) and standard bottom up fixed point calculation.

1. Introduction

1.1. Background - abstract interpretation

Abstract interpretation is widely used technique for analysing programs. Originally developed for flow analysis of imperative languages[2] It has been especially useful in generating information for the efficient compilation of pure functional languages.[3, 4]

Examples of abstract interpretation include:

- - strictness analysis
- - determinism analysis
- - type checking
- - Currying analysis
- - list length analysis

Typically it is the aim of abstract interpretation to completely evaluate the denotation of the program/function/expression in the abstract domain. Again typically, but not necessarily, the abstract domain needs to be finite for this interpretation to succeed for recursive functions. Where the domains are not naturally finite, some adjustment may be necessary. For example, list lengths are unbounded (or even infinite), but analysis can be performed over the domain { 0, 1, *many* }. Similarly, the domains for Milner type checking are unbounded, but well typed programs have finite types, and careful analysis can prevent infinite recursion for incorrect programs.

Strictness analysis of first order functions is performed over the domain { $\bot$, $\top$ }, where $\bot$ means definite non-termination and $\top$ means possible termination. Higher order functions have finite domains too, although they grow in size rapidly.

1.2. Solution by fixed points

Having obtained semantic equations for the abstract domain, it is easy to solve those which are non-recursive. Thus abstract interpretation reduces to the finding of fixed points of an abstracted generating functional. The simple solution to this is to iterate, starting at $\bot$. The problem with this is the large number of argument values we may be solving for. In the first order case there are 2^n possible arguments to a function of arity n, and for higher order functions this increases rapidly. Clack and Peyton-Jones use *frontier analysis*[5] which is a compact way of storing the values of monotonic functions. This significantly improves the space and time complexity in "average" cases.

1.3. Pending analysis

Unfortunately, even when there are only a few points at which we want to know the functions value, we must iterate until the whole function is stable, as there may be "plateaus" for particular value which may later start to climb again.

Young and Hudak[1] suggest an algorithm called *pending analysis* which can take advantage of the few values that the fixed point is required at. They suggest that when evaluating the abstracted function f at the argument value a, we keep a note that $f(\ a\ )$ is "pending", then when we encounter during the expansion of $f(\ a\ )$ another instance of the same application, we substitute $\bot$ for this. They differentiate this from standard iteration in that it is an "inside-out" strategy whereas pending analysis is "outside-in".

They only present their algorithm for the case of binary domains (with strictness analysis in mind), but do suggest that the technique could be extended to deeper domains by iterating. They also limit themselves to first order functions.

1.4. Purpose and content of paper

Young and Hudak proposed their algorithm after practical experience suggested its correctness. They include a proof, however this is slightly flawed. The purpose of this paper is to give correct proofs of pending analysis for the binary case with conditions for its correctness, and to extend the algorithm to more complex domains, both deeper domains and higher order functions.

Although I have not found any counter examples where simple pending analysis fails, it is only possible to prove its correctness for a limited class of functions, namely *non-self applicatory* functions. A slightly different algorithm is proposed which is correct for all functions, and which makes better use of the monotonicity of the target function.

Because the flaws in the original proof are quite subtle, and because many obvious properties turn out to be false (or far more difficult than at first glance) the proofs are given in detail, however I hope the progression

from simple to more complex proofs makes this more acceptable. Proofs will give special attention to the monotonicity of intermediate functions, as this is where most problems arise. They are typically concerned with two factors:

Safety - giving a result no lower in the domain than is correct.

Optimality - giving a result no higher than necessary

The safety result is most important in strictness analysis, as it is a more dangerous error to consider an argument strict that is not than to leave a strict argument lazy. In general we can assume that the domain is oriented such that $\top$ is safe but $\bot$ is most desirable. This is in fact only important when we consider higher order functions. Young and Hudak make use of the relative importance of safety when they suggest "pessimizing", that is returning results deliberately to high in the domain. However "obviously" safe options such as this must be viewed with extreme care as because of non-monotonicity they may be incorrect (unsafe).

We will consider first of all, in Section 2, the case of binary domains. The problems in Young and Hudak's proof are discussed and correct proofs given, together with the improved algorithm. After this is a short section giving conditions when simple uniterated pending analysis can be applied to more complex domains.

Section 4 introduces the representation of partial functions as *configurations* which will be needed in the analysis of non-binary domains. Section 5 and 6 give proofs of algorithms for non-binary domains, Section 5 is concerned with non-self applicatory functions, and proves simple pending analysis with iteration correct for these. Section 6 extends the analysis to all functions using the improved algorithm.

The representation of configurations is crucial to the efficiency of the improved algorithm, and Section 7 discusses ways of structuring and processing these. Higher order functions are discussed in Section 8, and it is suggested that these may be represented by terms (a *reified* interpretation) yet still preserve the correctness of the algorithm. Thus saving on the otherwise complex representation of these.

The proofs given are all expressed for evaluation in purely functional terms, and thus optimisations like memoing require slight reworkings of these, especially to ensure optimality. Instead of doing these directly Section 9 and 10 take a different approach. Section 9 looks at the fixed point properties of configurations considered in their own right. These properties suggest a new non-deterministic (or generic) algorithm which considers configurations as whole. Pending analysis, pending analysis with memoing and standard iterative fixed point calculation are all instances of this algorithm, and because the non-deterministic algorithm has been proved correct, all instances of it are also. In addition, new instances with more intelligent heuristics can be formulated.

2. Proofs for binary case

2.1. Notation

For internal consistency with the rest of this paper the notation will differ slightly from that of Young and Hudak. we are trying to solve for a function I with arguments in a domain *Appl* and results in a domain D. That is $I \in Interp$ where

$$Interp \;:\; Appl \;\rightarrow\; D$$

Typically I represents several user level functions, and its arguments are of the form $f(\, d_1 \,..\, d_n \,)$ (hence *Appl* - application). Elements of the domain *Interp* we will also refer to as interpretations. The desired function is defined recursively using a functional G

$$G \;:\; Interp \;\rightarrow\; Interp$$

and is the least fixed point of G

$$fix\ G \;=\; \bigvee I_i \qquad \textbf{where} \quad I_0 = \bot \qquad I_{i+1} = G\ I_i$$

Young and Hudak use the notation $h[\, v/\bar{x}\,]$ for the function that is equal to h everywhere except at $\bar{x}$ whence it equals v. For this section we will only use this when the substituted value is $\bot$, in addition it is

always a function resulting from and application of G, thus we will use the notation G_a for the functional such that for all I, $G_a\ I$ equals $G\ I$ everywhere except at a where it is $\perp$. This is the constrained iterator. We will also use the more general notation G_A with the obvious meaning:

$$\begin{array}{lcll} G_A\ I\ |_a & = & \perp & a \in A \\ & = & G\ I|_a & \textbf{otherwise} \end{array}$$

$$G_a \quad = \quad G_{\{\ a\ \}}$$

The notation $f|_x$ is used above to mean the value of f at x (i.e. $f(\ x\ )$), and is used to reduce the number of brackets that would otherwise appear. It is consistent with the standard notation (also used) $f|_S$ meaning the function identical to f except restricted to the set S.

2.2. One pending argument

Young and Hudak give a proof that for arbitrary positive integers, m, n (in their notation)

$$(\ G^m\ (\ G^n(\ \perp\ )\ )\ )(\ a\ ) \quad = \quad (\ G^m\ (\ G^n(\ \perp\ )[\ \perp/a\ ]\ )\ )(\ a\ )$$

or in my notation

$$G^m\ G^n(\ \perp\ )|_a \quad = \quad G^m\ G_a\ G^{n\text{-}1}(\ \perp\ )|_a$$

They give as an informal statement of the theorem

Theorem

> If, while evaluating $f(\ a\ )$ we find that it depends on the value of $f(\ a\ )$ again, returning $\perp$ as the result of the second (nested) call to $f(\ a\ )$ is correct with respect to the semantics of recursive monotonic boolean functions.

It should be noted that the formal theorem shows that for *any* such nested call (or to be precise all calls at a certain depth in the call graph) we can substitute bottom, it does not show that we can substitute for *all* such calls. The formal theorem is thus a justification of pending analysis, rather than a proof of it. Whether the informal and formal statements of the theorem say the same thing is a matter of interpretation.

To prove the stronger case we would need to show

$$G^m\ G^n(\ \perp\ )|_a \quad = \quad G^m\ G_a^n(\ \perp\ )|_a$$

At first sight we could follow exactly the lines of Young and Hudak's proof for this slightly stronger result:

Proof

(1) show $G_a^n\ I \quad \leqslant \quad G^n\ I$
  not trivial
(1.1) $G_a^0\ I \quad = \quad I \quad = \quad G^0\ I$
(1.2) induction, assume $G_a^i\ I \quad \leqslant \quad G^i\ I$
(1.3) $\Rightarrow \quad G\ G_a^i\ I \quad \leqslant \quad G\ G^i\ I$
(1.4) $\Rightarrow \quad G_a\ G_a^i\ I \quad \leqslant \quad G\ G^i\ I$
  QED 1
(2) $\Rightarrow$ LHS $\geqslant$ RHS
(3) case LHS = $\perp$ *QED*

(4) case LHS = $\top$
(5) let r be the largest integer such that $G^r(\perp)|_a = \perp$
show $G^r_a(\perp) = G^r(\perp)$
(5.1) $G^0_a(\perp) = \perp = G^0(\perp)$
(5.2) induction, assume $G^i_a(\perp) = G^i(\perp) \quad i < r$
(5.3) $\Rightarrow \quad G\ G^i_a(\perp) = G\ G^i(\perp)$
(5.4) $\Rightarrow \quad G\ G^i_a(\perp)|_a = \perp = G\ G^i(\perp)|_a$
(5.5) $\Rightarrow \quad G_a\ G^i_a(\perp) = G\ G^i_a(\perp) = G\ G^i(\perp)|_a$
QED 5

(6) show $G^r_a(\perp) \leqslant G^n_a(\perp)$
same r as above
(6.1) $G^0_a(\perp) = \perp \leqslant G^k(\perp) \quad \forall k$
(6.2) induction, assume $G^i_a(\perp) \leqslant G^j_a(\perp) \quad i < j$ **and** $i < r$
(6.3) $\Rightarrow \quad G\ G^i_a(\perp) \leqslant G\ G^j_a(\perp)$
(6.4) $\Rightarrow \quad G_a\ G^i_a(\perp) \leqslant G_a\ G^j_a(\perp)$
QED 6

(7) $\Rightarrow \quad G\ G^r_a(\perp) \leqslant G\ G^n_a(\perp)$
(8) $\Rightarrow \quad \top = G\ G^r(\perp) = G\ G^r_a(\perp) \leqslant G\ G^n_a(\perp)$
$\Rightarrow \quad G\ G^n_a(\perp) = \top$
QED case LHS = $\top$

QED theorem

Now this proof looks fairly convincing, but it is dubious at two places, steps 1.3 and 6.3. Both these steps depend on a monotonicity property of G of the form:

$$f \leqslant f' \quad \Rightarrow \quad G\ f \leqslant G\ f'$$

Now this of course is correct for monotonic functions f and f', but in buth cases we are dealing with the results of G_a which are of course in general not monotonic. In fact the proof does hold so long as G is defined using terms (as is the case in Young and Hudak's work). In this case it satisfies a stronger property it is *pseudo-monotonic*. That is the above identity hold so long as one or other of f or f' is monotonic.

Definition - *pseudo-monotonic*

G is pseudo-monotonic if
either f or f' monotonic **and** $f \leqslant f' \quad \Rightarrow \quad G\ f \leqslant G\ f'$

If we reexamine the two dubious steps, at 1.3 we find that we are comparing $G^i_a\ I$ and $G^i\ I$, the later of which is of course monotonic (so long as the seed I is). In the second case we were comparing $G^i_a(\perp)$ and $G^j_a(\perp)$ however we stipulated that $i \leqslant r$, so $G^i_a(\perp) = G^i(\perp)$ which is of course monotonic.

2.3. Further pending arguments

Of course, pending analysis would not be very useful if you could only use it for one argument. It is expected that we would recursively calculate the values of applications using pending analysis as they arise. We can describe this algorithm using the function X_A defined recursively as:

$$\begin{array}{lll} X_A|_a & = \perp & a \in A \\ & = (\ G\ X_{A+\{a\}}\ )|_a & \textbf{otherwise} \end{array}$$

The final result of pending analysis at a point a is $X_{\{\}}|_a$, and the function X_A is the result of pending analysis when we are part way through with A pending.

At first glance this seems to hardly merit further proof, if we imagine an example of the form:

```
f ( x, y )  =  y  and  ( f ( x, x )  or  f ( x, y ) )
```

Imagine evaluating this at $\bot, \top$, we would fill in the value of `f`$(\bot, \top)$ as $\bot$, then would require the true value of `f`$(\bot, \bot)$, this could clearly be obtained by pending analysis itself (giving eventually $\bot$), hence eventually yielding the correct value.

If we chose a slightly different example however

```
f ( x, y )   =   y  and  ( f ( x, x )  or  f ( y, x ) )
```

This time, we when evaluating at $\bot, \top$, we would get a call of `f`$(\top, \top)$ within the recursive call to `f(y,x)`, in this case we don't really want the correct value of `f`$(\top, \top)$ so much as the correct value when were assuming `f`$(\bot, \top)$ is pending. It seems that it is at least safe to return the correct value, since if anything this is (surely?) as large as the value with an application pending. But is it optimal?

In fact, we shall see that there are more fundamental problems than that, and that the obvious proof, following the line of the above fails in certain cases. We will uncover these by attempting the proof, which turns out to be valid for many cases including the previous examples. By doing this we will uncover an important sub-class of functions that also have a simpler proof for the case of non-binary domains. Further the shape of the proof is very similar to the progressively more complex proofs in the paper and is a useful introduction.

2.4. Proof of binary pending analysis for non-self applicatory functions

We can repeat the proof for single application pending analysis over G, but this time for an embedded case with some applications pending (A).

Theorem

$$G_A\ {G_A}^n(\bot)|_a \quad = \quad G_A\ {G_{A+\{a\}}}^n(\bot)|_a$$

At first sight we could follow exactly the lines of Young and Hudak's proof for this slightly stronger result:

Proof

(1) show ${G_{A+\{a\}}}^n\ I \quad \leqslant \quad {G_A}^n\ I$
not trivial
(1.1) ${G_{A+\{a\}}}^0\ I \quad = \quad I \quad = \quad {G_A}^0\ I$
(1.2) induction, assume ${G_{A+\{a\}}}^i\ I \quad \leqslant \quad {G_A}^i\ I$
(1.3) $\Rightarrow \quad G\ {G_{A+\{a\}}}^i\ I \quad \leqslant \quad G\ {G_A}^i\ I$
(1.4) $\Rightarrow \quad G_{A+\{a\}}\ {G_{A+\{a\}}}^i\ I \quad \leqslant \quad G_A\ {G_A}^i\ I$
QED 1
(2) $\Rightarrow$ LHS $\geqslant$ RHS
(3) case LHS $= \bot$ *QED*
(4) case LHS $= \top$
(5) let r be the largest integer such that ${G_A}^r(\bot)|_a \ = \ \bot$
show ${G_{A+\{a\}}}^r(\bot) \quad = \quad {G_A}^r(\bot)$
--- works as before ---
(6) show ${G_{A+\{a\}}}^r(\bot) \quad \leqslant \quad {G_{A+\{a\}}}^n(\bot)$
same r as above
(6.1) ${G_{A+\{a\}}}^0(\bot) \quad = \quad \bot \quad \leqslant \quad {G_A}^k(\bot) \quad \forall k$
(6.2) induction, assume ${G_{A+\{a\}}}^i(\bot) \quad \leqslant \quad {G_{A+\{a\}}}^j(\bot) \quad i < j$ **and** $i < r$
(6.3) $\Rightarrow \quad G\ {G_{A+\{a\}}}^i(\bot) \quad \leqslant \quad G\ {G_{A+\{a\}}}^j(\bot)$
(6.4) $\Rightarrow \quad G_{A+\{a\}}\ {G_{A+\{a\}}}^i(\bot) \quad \leqslant \quad G_{A+\{a\}}\ {G_{A+\{a\}}}^j(\bot)$
QED 6
(7) $\Rightarrow \quad G_A\ {G_{A+\{a\}}}^r(\bot) \quad \leqslant \quad G_A\ {G_{A+\{a\}}}^n(\bot)$
(8) $\Rightarrow \quad \top \ = \ G_A\ {G_A}^r(\bot) \ = \ G_A\ {G_{A+\{a\}}}^r(\bot) \quad \leqslant \quad G_A\ {G_{A+\{a\}}}^n(\bot)$
$\Rightarrow \quad G_A\ {G_{A+\{a\}}}^n(\bot) \quad = \quad \top$
QED case LHS $= \top$

QED theorem

Step 5 is an argument about equality and is fine, however steps 1.3 and 6.3 are there to worry us again. The pseudo-monotonicity of G (and likewise G_A) is of no avail, as in general neither of ${G_{A+\{a\}}}^i\ I$ and ${G_A}^i\ I$ are monotonic even when $I = \bot$ (the particular case of interest). Similarly in step 6.3 neither ${G_{A+\{a\}}}^i(\bot)$ nor ${G_{A+\{a\}}}^j(\bot)$ will be monotonic. In fact it is quite on the cards that if a functional F does not yield monotonic results then we can have $F^{n+1}(\bot) \leqslant F^n(\bot)$!

If we try out various examples (such as those in the previous section) we find that in many cases these properties do hold, and the temporary non-monotonic functions are acceptable.

If we imagine iterating for a typical function, say f

$$f^*(x, y) = y \quad \textbf{and} \quad (f(x, x) \ \textbf{or} \ f(y, x))$$

where f^* means the next iteration for f. We find that if $f \leqslant f'$ then $f^* \leqslant f'^*$ *whether or not* f and f' are monotonic. That is the generating functional for f is monotonic for all functions whether monotonic or not. We will call such a functional *fully monotonic*.

Definition - *fully monotonic*

G is fully monotonic if
$\forall f, f' \quad f \leqslant f' \quad \Rightarrow \quad G\ f \leqslant G\ f'$
f and f' not necessarily monotonic

When do the "general" cases, where the proof fails, arise? The answer is *self applicatory* functions. That is functions defined so that an argument to the function may contain an expression dependent on the function. For instance, `fib` is non-self applicatory

```
fib (n)  =   if n < 2   then   1
                        else   fib ( n-1 )  +  fib ( n-2 )
```

In the definition of `fib`, the arguments to it are `n-1` and `n-2`, which do not contain references to `fib`.

On the other hand `tak` is self applicatory

```
tak ( x, y, z )   =
     if  x < y  then   z
                else   tak ( tak ( x-1, y, z ),
                             tak ( y-1, z, x ),
                             tak ( z-1, x, y ) )
```

The resulting functional is not fully monotonic (although of course still pseudo-monotonic).

It is easy to see where a loss of monotonicity can appear. If we increase the input value of f to an expression like $f(f(\top, \top), \bot)$, then the result of the inner application of f will clearly increase, but if f was not monotonic then the outer application may decrease. For example:

$$\begin{array}{llll} f(\bot,\bot) = \top & & f'(\bot,\bot) = \top \\ f(\top,\bot) = \bot & & f'(\top,\bot) = \bot \\ f(\top,\top) = \bot & & f'(\top,\top) = \top \end{array}$$

so $f \leqslant f'$ but
$f(f(\top, \top), \bot) \rightarrow f(\bot, \bot) \rightarrow \top$
$f'(f'(\top, \top), \bot) \rightarrow f'(\top, \bot) \rightarrow \bot$

Formally, a first-order term is self-applicatory if every application of a user defined function is a constant term (that is a term involving primative operations and arguments only). When we are defining several user defined functions, we can treat non-mutually recursive functions as primative. The higher order case is slightly more complex as we need also be careful of non-fully monotonic primatives (like application). Self application is easy to check statically, and could be done as a pre-pass before more extensive analysis, in order to select appropriate algorithms.

If we restrict ourselves to functionals defined by non-self applicatory terms, we can show that they are fully monotonic (see appendix) and thus the above proof holds. We can thus prove binary pending analysis

correct for this case.

The general result proved is that $X_A = fix\ G_A$, which of course includes as a special case $X_{\{\}} = fix\ G$. Because the G_A are fully monotonic, the fixed point construction makes sense even though the resulting functions are not monotonic. It is useful to restate the result of the above theorem as:

Incremental fixed point theorem for binary functions

$$fix\ G_A|_a = G_A\ fix\ G_{A+\{a\}}|_a$$

Further on we shall prove two similar results, one for non-binary but self applicatory functionals, and one for general functionals. These results will however become progressively more complex.

We can thus prove the correctness. When A is complete then X_A, G_A and hence $fix\ G_A$ are all identically $\perp$. The recursive case is proved using the above theorem.

Theorem

$$X_A = fix\ G_A$$

Proof

Induction on size A.
Base case A complete
$X_A = \perp$
$G_A = \perp \Rightarrow fix\ G_A = \perp$
QED base case
Inductive case.
Sub-cases
case $a \in A$
$X_A|_a = \perp = G_A\ X_{A+\{a\}}|_a$
QED $a \in A$
case $a \notin A$
$X_A|_a = G_A\ X_{A+\{a\}}|_a$
$= G_A\ fix\ G_{A+\{a\}}|_a$ - induction
$= fix\ G_A|_a$ - incremental fixed point theorem
QED $a \notin A$
QED inductive cases

QED Theorem

Again we shall see similar proofs later on for non-binary self applicatory and for general functionals.

2.5. General, binary functionals

Simple pending analysis does appear to still work in these cases, but proofs break down. In order to produce a proof, we must alter the algorithm slightly. The problem is having intermediate non-monotonic functions. For the non-self applicatory case this did not matter as the functionals were fully monotonic. In the general case we cannot rely on this and must therefore force the intermediate results to be monotonic. The simplest way to do this is to always extend the pending values in A so as to make it downwards complete, that is:

$$a \in A \quad \textbf{and} \quad a' < a \quad \Rightarrow \quad a' \in A$$

If this is so, then G_A will take monotonic functions to monotonic functions, hence all the intermediate results are monotonic. The only extra problem is that we will add more then one element to A, does this mean we have to iterate several times? In fact not, but we have to modify the proofs very slightly to take account of this.

The new algorithm is:

$$\begin{array}{lcll} Z_A|_a & = & \bot & a \in A \\ & = & (\ G\ Z_{A \oplus \{a\}}\ )|_a & \textbf{otherwise} \end{array}$$

where $A \oplus \{a\}$ is A extended by A and then downwards closed. That is:

$$A \oplus \{a\} \quad = \quad A \cup \{\ a' \mid a' \leqslant a\ \}$$

We proceed by proving and incremental fixed point theorem, and then showing that $Z_A = fix\ H_A$. As all the sets A considered are downwards closed, the fixed point construction is meaningful.

Theorem - Incremental fixed point theorem for general binary functions

$$fix\ G_A|_a \quad = \quad G_A\ fix\ G_{A \oplus \{a\}}|_a \qquad \text{- } A \text{ downwards closed}$$

Proof

Steps -
(1) $G_A \geqslant G_{A \oplus \{a\}}$
(2) $\Rightarrow$ LHS $\geqslant$ RHS
(3) case LHS = $\bot$ *QED*
(4) case LHS = $\top$
(5) let r be largest such that ${G_A}^r(\bot)|_a = \bot$
show ${G_{A \oplus \{a\}}}^r(\bot) = {G_A}^r(\bot)$
(5.1) ${G_{A \oplus \{a\}}}^0(\bot) = \bot = {G_A}^0(\bot)$
(5.2) induction, assume ${G_{A \oplus \{a\}}}^i(\bot) = {G_A}^i(\bot) \quad i < r$
(5.3) $\Rightarrow$ $G\ {G_{A \oplus \{a\}}}^i(\bot) = G\ {G_A}^i(\bot)$
(5.4) $\Rightarrow$ $G\ {G_{A \oplus \{a\}}}^i(\bot)|_a = \bot = G\ {G_A}^i(\bot)|_a$
(5.5) $\Rightarrow$ $\forall a' \leqslant a \quad G\ {G_{A \oplus \{a\}}}^i(\bot)|_{a'} = \bot$ - monotonicity
(5.6) $\Rightarrow$ $G_{A \oplus \{a\}}\ {G_{A \oplus \{a\}}}^i(\bot) = G_A\ {G_{A \oplus \{a\}}}^i(\bot) = G_A\ {G_A}^i(\bot)$
QED 5
(6) show ${G_{A \oplus \{a\}}}^r(\bot) \leqslant {G_{A \oplus \{a\}}}^n(\bot)$
same r as above
---- proof exactly as for $G_{A + \{a\}}$ ----
(7) $\Rightarrow$ $G_A\ {G_{A \oplus \{a\}}}^r(\bot) \leqslant G_A\ {G_{A \oplus \{a\}}}^n(\bot)$
(8) $\Rightarrow$ $\top = G_A\ {G_A}^r(\bot) = G_A\ {G_{A \oplus \{a\}}}^r(\bot) \leqslant G_A\ {G_{A \oplus \{a\}}}^n(\bot)$
$\Rightarrow$ $G_A\ {G_{A \oplus \{a\}}}^n(\bot) = \top$
(9) $\Rightarrow$ $G_A\ fix\ G_{A \oplus \{a\}} = \top$
QED case LHS = $\top$

QED theorem

The only difference from the previous proof was step 5.5, where we had to be careful that all the additions to $A \oplus \{a\}$ were already $\bot$, but the monotonicity of the intermediate results ensured this. The final correctness of the algorithm, namely:

Theorem - correctness of pending analysis for general binary functions

$$Z_A \quad = \quad fix\ G_A \qquad \text{- } A \text{ downward closed}$$

can be proved exactly as the non-self applicatory case, except that the recursive case has $A \oplus \{a\}$ rather than $A + \{a\}$. This proof is omitted.

2.6. Discussion of binary case

The proof given in Young and Hudak's paper is useful in convincing one of the correctness of pending analysis, but does not prove it entirely. It is possible to use the form of their proof to give a full proof of correctness of pending analysis applied to one application, but even this uses a property of functionals defined using terms, pseudo-monotonicity, the normal monotonicity of the functional being insufficient to complete the proof. The case of multiple pending applications is more difficult and we needed to restrict ourselves to non-self applicatory functions when the defining functional would be fully-monotonic and hence the non-

monotonicity of the intermediate terms acceptable. Finally, we proved the case of general binary functions by modifying the basic algorithm slightly in order to ensure monotonicity of intermediate results.

We have *not* proved the general case using simple pending analysis, but equally I have not found any counter examples, although there are functionals that do not satisfy apparently sensible properties like:

$$G\ G_{A+\{a\}}\ fix\ G \quad \leqslant \quad G\ G_A\ fix\ G$$

3. Extending binary analysis by abstract interpretation

If we are working over a non-binary domain D we may still be able to use the results for the binary case if the primative operations have a particular form. For any y in D define:

$$\begin{array}{lll} Abs_y : D \rightarrow \mathbf{2} & & \\ d_y = \top & \textbf{if} & d \geqslant y \\ \quad = \bot & \textbf{otherwise} & \end{array}$$

Now if for each y and each primative operator μ we can define an abstract version μ_y such that:

$$\forall\ a \in \textbf{dom}\ \mu \quad \mu_y(\ a_y\ ) \quad = \quad (\ \mu(\ a\ )\ )_y$$

Then this identity is also satisfied for the functional G defined in terms of them, and we can prove that simple pending analysis also works for this non-binary case.

Sketch of proof

Assume that $fix\ G\ (\ a\ ) = y$. We use Abs_y and find that the result of the abstract pending analysis is $\top$, and hence show that the result of pending analysis is at least as big as y. We then by consider $Abs_{y'}$ for all y' above y, and similarly show that the abstract pending analysis yields $\bot$ and hence that the result of pending analysis is less than all these y'.

In the following, let $Pend(\ F, x\ )$ be the result of applying pending analysis to obtain the value of the fixed point of F at the point x in the appropriate domain.

Proof

$fix\ G\ (\ a\ ) = y$
Step 1 - prove $Pend(\ G, a\ ) \geqslant y$
$fix\ G\ (\ a\ ) = y$
$fix\ G_y\ (\ a_y\ ) = \top$
$Pend(\ G_y, a_y\ ) = \top$ - correctness of pending analysis for binary domain
$(\ Pend(\ G, a\ )\ )_y = \top$ - pending analysis uses only primative operations
$Pend(\ G, a\ ) \geqslant y$ - def'n of Abs_y
QED 1
Step 2 - prove $y < y' \Rightarrow Pend(\ G, a\ ) < y'$
$fix\ G\ (\ a\ ) = y$
$fix\ G_{y'}\ (\ a_{y'}\ ) = \bot$ - $y < y'$
$Pend(\ G_{y'}, a_{y'}\ ) = \bot$
$(\ Pend(\ G, a\ )\ )_{y'} = \top$
$Pend(\ G, a\ ) < y'$ - def'n of Abs_y
QED 2
QED

4. Configurations

Frontier algorithms and the like are based around producing successive interpretations approximating $fix\ G$. That is they find values for all possible function applications simultaneously. Pending analysis seeks to only look at sufficient points to find the value of $fix\ G$ for a particular function at a particular point. We will therefore want to consider partial interpretations. We will call a partial interpretation a *configuration*.

The set of configurations is then a subset of the partial functions between applications and results.

$$Config \quad \subseteq \quad Appl \twoheadrightarrow D$$

We will use the notation $C + (a \rightarrow d)$ for the extension of C to map an application a to a result d. It is assumed that $a \notin \mathbf{dom}\ C$. That is:

Definition - $C + (a \rightarrow d)$

$$\mathbf{dom}\ C + (a \rightarrow d) \quad = \quad \mathbf{dom}\ C \ + \ \{ a \}$$

$$C + (a \rightarrow d)|_{\mathbf{dom}\ C} \quad = \quad C$$
$$C + (a \rightarrow d)|_a \quad = \quad d$$

We will use several partial orderings on configurations.

We can compare configurations using a pointwise comparison:

$$C \leqslant C' \quad \equiv \quad \mathbf{dom}\ C = \mathbf{dom}\ C'$$
$$\forall\, a \in \mathbf{dom}\ C \quad C (a) \leqslant C' (a)$$

We will also need a domain oriented ordering -

$$C \subseteq C' \quad \equiv \quad \mathbf{dom}\ C \subseteq \mathbf{dom}\ C'$$
$$\forall\, a \in \mathbf{dom}\ C \quad C (a) = C' (a)$$
$$i.e. \quad C'|_{\mathbf{dom}\ C} = C$$

Also we want a mixture of the two :

$$C \ll C' \quad \equiv \quad \mathbf{dom}\ C \subseteq \mathbf{dom}\ C'$$
$$\forall\, a \in \mathbf{dom}\ C \quad C (a) \leqslant C' (a)$$
$$C \gg C' \quad \equiv \quad \mathbf{dom}\ C \subseteq \mathbf{dom}\ C'$$
$$\forall\, a \in \mathbf{dom}\ C \quad C (a) \geqslant C' (a)$$

These last two relations are weaker than the above two:

$$C \leqslant C' \quad \Leftrightarrow \quad C \ll C' \ \textbf{and} \ C' \gg C$$
$$C \geqslant C' \quad \Leftrightarrow \quad C \gg C' \ \textbf{and} \ C' \ll C$$
$$C \subseteq C' \quad \Leftrightarrow \quad C \ll C' \ \textbf{and} \ C \gg C'$$

A configuration is *total* if its domain is all of $Appl$.

A configuration (partial or total) is said to be *monotonic* if:

$$\forall\, a, a' \in Appl \quad a \leqslant a' \ \Rightarrow \ C_a \leqslant C_{a'}$$

A total configuration can be regarded as an interpretation and this definition of monotonicity is then standard monotonicity.

We can express the recursion as a function H over total monotonic configurations:

$$H \quad : \quad Config \rightarrow Config$$

$$HC \quad = \quad \cap\{ \ G\ I \mid C \subseteq I \ \}$$

Note that this is the intersection of the $G\ I$, that is the result is a configuration which is only defined at the points where *all* the I agree. An equivalent definition would be.

$$\mathbf{dom}\ (HC) \quad = \quad \{ \ a \mid \forall I, I' \ \ C \subseteq I \ \textbf{and} \ C \subseteq I' \ \Rightarrow \ G\ I(a) = G\ I'(a) \ \}$$

$$(HC)|_a \quad = \quad G\ I(a) \text{ for any } I \ \textbf{st} \ C \subseteq I$$

H is thus clearly a monotonic (wrt. $\ll$, $\gg$, $\leqslant$ and $\subseteq$) continuous functional over monotonic configurations and equal to G for total configurations. In most of the following we will only apply H to total configurations, but we will use H consistently whether the configuration is total (and G would do) or not. Typically

when a configuration is total we will refer to it as I or J, possibly primed, whereas configurations will be C, C' etc.

5. Pending analysis for non-self-applicatory functions

Young and Hudak suggest that their algorithm can be extended to non-binary domains by iterating the pending analysis at each stage. This brings to mind numerical optimisation algorithms, where when optimising several quantities, one fixes one of them and optimises over the rest under that assumption. Finally one optimisises the chosen "pivot" value using the result of the recursive optimisation as ones target function.

In the binary case, we had the simple recursion equation for X_A, the function "given" that the elements of A are $\perp$.

$$\begin{array}{lcll} X_A|_a & = & \perp & a \in A \\ X_A|_a & = & H\ X_{A+a}|_a & \text{otherwise} \end{array}$$

This is more complex when the domain is deeper. Whereas the in the binary case we get the right value after one application of H we must either expand the function deeper, or iterate. By expanding several calls before we apply the pending value, we would do too much work if the value were quite low in the domain, and thus the second course seems better. Iterating requires relatively minor changes to the binary algorithm. In the binary case, an application was either not pending or $\perp$, in the non-binary case an application may be pending with various values and hence we use a configuration to capture these pending values. When we come to expand the value of a function, and the application is pending, we return the pending value as held in the configuration:

$$X^L_C|_a \quad = \quad C(a) \qquad a \in \textbf{dom}\ C$$

When we come to expand the value of a function, we must change the rule when the application is not pending. For binary analysis this was:

$$X_A|_a \quad = \quad H\ X_{A+a}|_a$$

this must now include a fixed point calculation:

$$\begin{array}{l} X^L_C|_a \quad = \quad fix\ h \\ \textbf{where} \\ \qquad h(d) \ = \ (H\ X^L_{C+(a\to d)})|_a \end{array}$$

Clearly we must again be careful about non-monotonic intermediate results, as otherwise the fixed point calculation may not be valid and we cannot argue sensibly about $\leqslant$ conditions. Thus we will first confine ourselves to the case of non-self applicatory functionals as we did for binary functions. We will return to the general case in the next section with a slightly modified algorithm. Non-self applicatory functions are ok because their generating functionals are fully monotonic. Although intermediate results will be non-monotonic the final function is monotonic however.

In order to prove the algorithm correct we will need to prove an *incremental fixed point theorem* that says we can calculate the fixed point of H at a point a by first working out a fixed point of a function using H fixed at various values. In general we will have that the fixed point of H^L_C (H fixed to take the values of C) can be expressed in terms of $H^L_{C+(a\to d)}$:

Incremental fixed point theorem for fully monotonic functionals

$$\begin{array}{l} fix\ H^L_C|_a \quad = \quad fix\ g \\ \textbf{where} \\ \qquad g(d) \ = \ H^L_C\ (\ fix\ H^L_{C+(a\to d)}\ )\ |\ a \end{array}$$

H^L_C can be defined in terms of a limiting functional L_C:

$$
\begin{array}{lcll}
H^L_C & = & L_C\ H & \\
L_C\ F \mid a & = & C(a) & a \in \textbf{dom}\ C \\
 & = & F(a) & \textbf{otherwise}
\end{array}
$$

Clearly $L_{\{\}}$ is the identity and thus $fix\ H^L_{\{\}} = fix\ H$. Further L_C is fully monotonic so all the H^L_C, bt composition with H, will be also.

Given this we can prove the correctness of the evaluation algorithm.

$$
\begin{array}{lcll}
X^L_C & : & Appl \rightarrow D & \\
 & & & \\
X^L_C|_a & = & C(a) & a \in \textbf{dom}\ C \\
X^L_C|_a & = & fix\ h & \text{otherwise} \\
\textbf{where} & & & \\
\quad h(d) & = & (H\ X^L_{C+(a\rightarrow d)})|_a &
\end{array}
$$

We want $X^L_{\{\}}$ to be equal to $fix\ H$, so we prove the general case that for all C, $X^L_C = fix\ H^L_C$. The incremental fixed point theorem is used to prove the recursive case. We proceed by induction on the size of C.

5.1. Correctness of algorithm for non-self applicatory terms

Theorem

$$X^L_C = fix\ H^L_C$$

Proof

Induction on size of C
We assume that $X^L_C = fix\ H^L_C$, for any C such that $\| C \| > C$

Base case C total
In this case H^L_C is constantly C and thus $fix\ H^L_C = C$
Similarly X^L_C is constantly C
QED C total

Prove $\forall\ a \quad X^L_C(a) = (fix\ H^L_C)_a$
there are two cases :
either $a \in \textbf{dom}\ C$
$X^L_C(a) \rightarrow C_a$
but $fix\ H^L_C|_{\textbf{dom}\ C} = C$
QED $a \in \textbf{dom}\ C$
or $a \notin \textbf{dom}\ C$
$X^L_C(a) \rightarrow fix\ h$
where $h(d) = H\ X^L_{C+(a\rightarrow d)}|_a$
now $\| C + (a \rightarrow d) \| \geqslant \| C \|$
$\Rightarrow H\ X^L_{C+(a\rightarrow d)} = H\ fix\ H^L_{C+(a\rightarrow d)}$ - by induction
That is h here is equal to g of the incremental fixed point theorem,
and as g is monotonic the $fix\ h$ is the minimal fixed point of g.
We thus apply the theorem :-
$X^L_C(a) = fix\ g = (fix\ H^L_C)_a$
QED $a \notin \textbf{dom}\ C$
QED inductive cases

QED theorem

5.2. Incremental fixed point theorem for fully monotonic functionals.

As the theorem includes the fixed point of g it is slightly different from the proofs for the binary case. First of all we prove that $d_{fix} \leqslant (fix\ H^L_C)_a$. Rather than trying to prove the reverse equality directly, we use

this result to prove more generaly that and then use this to show that in general $fix\ H^L_C \geqslant fix\ H^L_{C_{fix}}$. Finally, we prove the reverse inequality for the general case by showing that $fix\ H^L_{C_{fix}}$ is a fixed point of H^L_C.

Theorem - incremental fixed point theorem

let $g(\ d\ ) = (\ H^L_C\ (\ fix\ H^L_{C+(\ a \to d\ )}\ )\ )_a$
and $d_{fix} = fix\ g$
and $C_{fix} = C + (\ a \to d_{fix}\ )$
then
(i) $(\ fix\ H^L_C\ )_a = d_{fix}$
(ii) $fix\ H^L_C = fix\ H^L_{C_{fix}}$

Proof

Steps -
(1) show that $d \leqslant d_{lhs} \Rightarrow g(\ d\ ) \leqslant d_{lhs}$
where $d_{lhs} = (\ fix\ H^L_C\ )_a$
Proof step 1
(1.1) $d \leqslant d_{lhs} \Rightarrow fix\ H^L_C|_{\mathbf{dom}\ C+(\ a \to d\ )} \geqslant C + (\ a \to d\ )$
(1.2) $\Rightarrow H^L_{C+(\ a \to d\ )}\ fix\ H^L_C \leqslant fix\ H^L_C$
(1.3) $\Rightarrow fix\ H^L_{C+(\ a \to d\ )} \leqslant fix\ H^L_C$
(1.4) $\Rightarrow H^L_C\ fix\ H^L_{C+(\ a \to d\ )} \leqslant H^L_C\ fix\ H^L_C = fix\ H^L_C$
(1.5) $\Rightarrow g(\ d\ ) \leqslant d_{lhs}$
QED (1)
(2) $\Rightarrow d_{fix} = fix\ g \leqslant d_{lhs}$
(3) $\Rightarrow fix\ H^L_{C_{fix}} \leqslant fix\ H^L_C$

(4) **let** $I = fix\ H^L_{C_{fix}}$
prove that $H^L_C\ I = I$
(4.1) $(\ H^L_C\ I\ )_a = d_{fix}$
– immediate from def'n of g and $g(\ d_{fix}\ ) = d_{fix}$
(4.2) $a' \neq a \Rightarrow (\ H^L_C\ I\ )_{a'} = (\ H^L_{C_{fix}}\ I\ )_{a'}$
QED (4)
(5) thus $fix\ H^L_{C_{fix}} = I \geqslant fix\ H^L_C$ – minimality of $fix\ H^L_C$
QED (ii)
(6) $fix\ H^L_C = H^L_C\ (\ fix\ H^L_C\ ) = H^L_C\ (\ fix\ H^L_{C_{fix}}\ )$
$\Rightarrow (\ fix\ H^L_C\ )_a = g(\ d_{fix}\ ) = d_{fix}$
QED (i)

QED theorem

Note the various properties of H^L_C that have been used through this proof.

Step 1.2 uses the fixing property of L_C, that is:

$$I|_{\mathbf{dom}\ C} \geqslant (=, =<)\ C \Rightarrow I \geqslant (=, =<)\ L_C\ I$$

and the absorption property of L_C and H^L_C

$$L_{C+(\ a \to d\ )} \circ L_C = L_{C+(\ a \to d\ )} = L_C \circ L_{C+(\ a \to d\ )}$$
$$\Rightarrow H^L_{C+(\ a \to d\ )} = L_{C+(\ a \to d\ )} \circ H^L_C$$

We will need a similar result for the slightly different iterating function which we will use later for general functions.

Step 1.3 uses the property of fixed points

$$F\ f \leqslant f \Rightarrow fix\ F \leqslant f$$

and requires that $H^L_{C+(\ a \to d\ )}$ be monotonic.

6. Pending analysis for general functions

6.1. Need for a bounding function

As we've said above, not all function definitions are free of self application hence we must produce a more complex analysis to deal with this case. We will need to force interpretations to agree with the assumed configuration in a way that leaves them monotonic. We must however be very careful in the way we do this however. We might consider using the *ceiling* function to normalise interpretations (where *ceiling*(I) is the least monotonic interpretation greater then or equal to I), This however implies a global analysis of the interpretation and is not suitable for a pending analysis algorithm. So we must beware on two points

- Montonicity - intermediate results must be monotonic
- Locality - applying the result of a functional at a point must depend on just a few values of the functionals argument.

The incremental fixed point theorem below instead makes use of a "bounding" function that constrains interpretations to agree with a configuration but does not use any information except the value of the interpretation at the point of interest, and the configuration itself, satisfying both constraints. This leads to an effective (but inefficient) algorithm. However, having proved the basic algorithm correct, various improvements are apparent.

We will have to prove properties of the bounding function B_C similar to those required for the limiting function L_C in the previous section, such as the functions which are its fixed points, absorption properties. These will then be used to prove properties of the constrained iterator $H_C = B_C\, H$ necessary to prove an iterative fixed point theorem which should by now be familiar. This will be used to prove the correctness of a variant of pending analysis in a proof which again should be familiar!

In general all the results proved seem obviously true, however some need surprising care, hence the slow build up from properties of configurations, to those of the bounding function, those the iterator and then finally the fixed point theorem itself.

6.2. Configuration operators

Before describing the bounding function itself, we need a few new operators on configurations. For any configuration C we define $C^\top$ and $C^\bot$ as follows:

$C^\top$ and $C^\bot$ are total

$$C^\top(a) = \bigwedge_{a' \geqslant a} C(a')$$

$$C^\bot(a) = \bigvee_{a' \leqslant a} C(a')$$

We can think of $C^\top$ as the greatest total monotonic configuration below C, and similarly $C^\bot$ is the least above C. That is:

for all total and monotonic C',

$$C \gg C' \quad \Rightarrow \quad C^\top \geqslant C'$$

$$C \ll C' \quad \Rightarrow \quad C^\bot \leqslant C'$$

These operators preserve some of the ordering properties of the configurations.

Lemma

(i) $C \ll C' \Rightarrow C^{\perp} \leqslant C'^{\perp}$
(ii) $C \gg C' \Rightarrow C^{\top} \geqslant C'^{\top}$

Proof

The two results are duals so we prove the former

$$C^{\perp} = \bigvee_{a' \leqslant a} C(a')$$
$$\leqslant \bigvee_{a' \leqslant a \text{ and } a' \in \mathbf{dom}\, C} C'(a')$$
$$\leqslant \bigvee_{a' \leqslant a} C'(a')$$
$$= C'^{\perp}$$

QED (i)

QED lemma

Lemma

$C \leqslant C' \Rightarrow C^{\perp} \leqslant C'^{\perp}$ **and** $C^{\top} \leqslant C'^{\top}$

Proof

$C \leqslant C' \;\square\; C \ll C'$ **and** $C' \gg C$
$\Rightarrow C^{\perp} \leqslant C'^{\perp}$ **and** $C'^{\top} \geqslant C^{\top}$
– applying (i) and (ii) above

QED lemma

Lemma

$C \subseteq C' \Rightarrow C^{\perp} \leqslant C'^{\perp}$ **and** $C^{\top} \geqslant C'^{\top}$

Proof

$C \subseteq C' \;\square\; C \ll C'$ **and** $C \gg C'$
$\Rightarrow C^{\perp} \leqslant C'^{\perp}$ **and** $C^{\top} \geqslant C'^{\top}$
– applying (i) and (ii) above

QED lemma

Lemma

C monotonic $\Rightarrow C^{\top}|_{\mathbf{dom}C} = C$

$C^{\perp}|_{\mathbf{dom}C} = C$

$C^{\perp} \leqslant C^{\top}$

Proof

first two trivial.
last simple too -
$\forall a, a', a'' \in Appl$ **st** $a' \leqslant a \leqslant a''$

$a', a'' \in \mathbf{dom}\, C \Rightarrow C(a') \leqslant C(a'')$ – monotonicity of C

thus $\bigwedge_{a' \leqslant a} C(a') \leqslant \bigvee_{a'' \geqslant a} C(a'')$

that is $C^{\perp}(a) \leqslant C^{\top}(a)$ as required

QED lemma

The eventual algorithm will work by incrementally adding to the set of assumptions. As we need to retain a monotonic configuration we need to add in such a way as to preserve monotonicity.

Definition - $C \oplus (a{-}{>} d)$

$$\begin{aligned}
&\mathbf{dom}\ C \oplus (a{-}{>} d) &&= (\mathbf{dom}\ C) + \{ a \} \\
&C \oplus (a{-}{>} d)|_{\mathbf{dom}\ C} &&= C \\
&C \oplus (a{-}{>} d)(a) &&= (d \wedge C^{\top}(a)) \vee C^{\perp}(a)
\end{aligned}$$

Lemma

C monotonic $\Rightarrow$ $C \oplus (a{-}{>} d)$ also monotonic
$C \subseteq C \oplus (a{-}{>} d)$
$C \oplus (a{-}{>} d)$ is monotonic ($\leqslant$) as a function of d

Proof - trivial

6.3. The bounding function

Now we can define the bounding function in terms of $C^{\top}$ and $C^{\perp}$:

Definition

$$B_C (I) = (I \wedge C^{\top}) \vee C^{\perp}$$

For monotonic C the order of bounding is not important as (the above lemma) $C^{\perp} \leqslant C^{\top}$. Note also that B_C has good locality properties, in order to work out $(B_C\ I)|_a$ one only needs to know the value of I at a and the value of the configuration.

We prove several lemmas, the first three regarding the fixed points of B_C, and after that an absorption property. These will be used in the same way as the corresponding properties for L_C in the non-self applicatory case. Finally we will prove the necessary monotonicity properties of B_C.

Lemma - fixing

$$C \text{ monotonic} \Rightarrow \forall I \quad B_C (I)|_{\mathbf{dom}\ C} = C$$

Proof

let $I' = B_C (I)$
$\forall\ a \in \mathbf{dom}\ C$
$\quad C^{\top}(a) = C (a)$ **and** $C^{\perp}(a) = C (a)$
$\Rightarrow$
$\quad I'(a) = (I(a) \wedge C^{\top}(a)) \vee C^{\perp}(a)$
$\quad = C (a)$

QED lemma

Lemma - fixed points of B_C

$\forall$ C monotonic and I monotonic

(i) $I|_{\mathbf{dom}\ C} \geqslant C \quad \Rightarrow \quad B_C(I) \leqslant I$

(ii) $I|_{\mathbf{dom}\ C} \leqslant C \quad \Rightarrow \quad B_C(I) \geqslant I$

(iii) $I|_{\mathbf{dom}\ C} = C \quad \Rightarrow \quad B_C(I) = I$

Proof

Clearly (i) + (ii) $\Rightarrow$ (iii)

Also (i) and (ii) are duals of one another, hence we only need to prove (i)

$I|_{\mathbf{dom}\ C} \geqslant C \quad \square \quad C \ll I$

$\Rightarrow \quad C^{\perp} \leqslant I^{\perp} = \perp$ lemma and I total and monotonic

$\Rightarrow \quad B_C(I) = (I \wedge C^{\top}) \vee C^{\perp}$

$= (I \vee C^{\perp}) \wedge C^{\top}$ – C monotonic

$= I \wedge C^{\top} \leqslant I$

QED (i)

QED lemma

Lemma - identity

$B_{\{\}} = id$

Proof - special case of the above

Lemma - absorption

C' monotonic **and** $C \subseteq C' \quad \Rightarrow \quad B_{C'} \circ B_C = B_{C'} = B_C \circ B_{C'}$

Proof

By previous lemmas $C^{\top} \geqslant C'^{\top}$ **and** $C^{\perp} \leqslant C'^{\perp}$

The proof of both results then follows by juggling the order of $\wedge$s and $\vee$s

We will prove the first result only as the proofs are so similar.

$$
\begin{aligned}
B_{C'} \circ B_C(I) &= (((I \wedge C^{\top}) \vee C^{\perp}) \wedge C'^{\top}) \vee C'^{\perp} \\
&= ((I \wedge C^{\top}) \vee C^{\perp} \vee C'^{\perp}) \wedge C'^{\top} && - C' \text{ monotonic} \\
&= ((I \wedge C^{\top}) \vee C'^{\perp}) \wedge C'^{\top} && - C^{\perp} \leqslant C'^{\perp} \\
&= (I \wedge C^{\top} \wedge C'^{\top}) \vee C'^{\perp} && - C' \text{ monotonic} \\
&= (I \wedge C'^{\top}) \vee C'^{\perp} && - C^{\top} \geqslant C'^{\top} \\
&= B_{C'}(I)
\end{aligned}
$$

QED lemma

In particular if C is monotonic we have :

$$B_{C \oplus (a \to d)} \circ B_C = B_{C \oplus (a \to d)}$$

If we are to use B_C in fixed point constructions we also need monotonicity:

Lemma

B_C is monotonic for any C

Proof

say $I_1 \geqslant I_2$
let $I_1' = B_C(I_1)$, $I_2' = B_C(I_2)$
$\forall a \in Appl \quad I_1(a) \geqslant I_2(a) \Rightarrow$
$I_1'(a) = (I_1(a) \wedge C^{\top}(a)) \vee C^{\perp}(a)$
$\geqslant (I_2(a) \wedge C^{\top}(a)) \vee C^{\perp}(a) = I_2(a)$

QED lemma

Note that this proof applies to all C and I *not just* monotonic ones. However, if C is not monotonic then $B_C(I)$ will not be monotonic, even when I is.

Lemma

B_C is monotonic as a function of C (with the $\leqslant$ ordering on *Config*) - again trivial

Note that B_C is *not* monotonic with respect to the $\subseteq$ and $\ll$ orderings on *Config*.

6.4. Constrained iterator

To find the fixed point of H we can iterate using it and starting at $\perp$. Pending analysis uses a constrained iterator, and we obtain this using the bounding function.

Definition

$$H_C = B_C \circ H$$

This inherits obvious monotonicity properties:

Lemma

$H_C(I)$ is monotonic both as a function of I and of C
$H_{\{\}} = H$
$C \leqslant C' \Rightarrow H_C \leqslant H_{C'}$
$C \subseteq C' \Rightarrow B_{C'} \circ H_C = H_{C'} = B_C \circ H_{C'}$

Proof

all follow straight-forwardly from the corresponding properties of B_C

QED lemma

The second two of these properties are reflected in their fixed points:

$fix\ H_{\{\}} = fix\ H$
$C \leqslant C' \Rightarrow fix\ H_C \leqslant fix\ H_{C'}$

It seems obvious that if we add an extra assumption to a constrained iterator that is bigger or smaller than the actual value at the fixed point, then the resulting fixed point of the new iterator will be likewise too big or small. The following lemmas prove this, first by proving a similar property for single applications, and then for the fixed points. Because it is the minimal fixed points we are interested in the proofs are not duals, and have a very different form.

Lemma

(i) $(H_C(I))_a \geqslant d \Rightarrow H_{C\oplus(a->d)}(I) \leqslant H_C(I)$
(ii) $(H_C(I))_a \leqslant d \Rightarrow H_{C\oplus(a->d)}(I) \geqslant H_C(I)$
(iii) $(H_C(I))_a = d \Rightarrow H_{C\oplus(a->d)}(I) = H_C(I)$

Proof

(i) + (ii) $\Rightarrow$ (iii)
Also (i) and (ii) are duals, so we need only prove (i)
let $d' = (H(I))_a$, $d'' = (H_C(I))_a$
then $d'' = (d' \wedge C^{\top}(a)) \vee C^{\perp}(a)$
$\Rightarrow d'' = (d'' \wedge C^{\top}(a)) \vee C^{\perp}(a)$
so $d \leqslant d'' \Rightarrow C\oplus(a->d)|_a \leqslant d''$ - monotonicity of $\vee$ and $\wedge$
Also $(H_C I)|_{\mathbf{dom}\, C} = C$
$\Rightarrow (H_C I)|_{\mathbf{dom}\, C\oplus(a->d)} \geqslant C\oplus(a->d)$
$\Rightarrow B_{C\oplus(a->d)}(H_C I) \leqslant H_C I$ – fixed points of $B_{C\oplus(a->d)}$
That is $H_{C\oplus(a->d)}(I) \leqslant H_C(I)$ – absorption
QED (i)

QED lemma

Lemma

$(fix\ H_C)_a \geqslant d \Rightarrow fix\ H_{C\oplus(a->d)} \leqslant fix\ H_C$

Proof

let $I = fix\ H_C$
then
$(H_C(I))_a \geqslant d$
$\Rightarrow H_{C\oplus(a->d)}(I) \leqslant H_C(I) = I$ – above lemma
$\Rightarrow fix\ H_{C\oplus(a->d)} \leqslant I = fix\ H_C$

QED lemma

Lemma

$(fix\ H_C)_a \leqslant d \Rightarrow fix\ H_{C\oplus(a->d)} \geqslant fix\ H_C$

Proof

Not a dual of the above!! - because *fix* is minimal fixed point.
Consider any J such that -
$J \leqslant fix\ H_{C\oplus(a->d)}$ **and** $J \leqslant fix\ H_C$
we want to prove that -
$H_C(J) \leqslant H_{C\oplus(a->d)}(J)$
now $J \leqslant fix\ H_C \Rightarrow (H_C J)_a \leqslant d$
$\Rightarrow H_{C\oplus(a->d)} J \geqslant H_C J$
– above lemma
QED $H_C(J) \leqslant H_{C\oplus(a->d)}(J)$
so starting with $J = \perp$ we can prove inductively that
$fix\ H_{C\oplus(a->d)} \geqslant fix\ H_C$ as required

QED lemma

Lemma

$$(fix\ H_C)_a = d \Rightarrow fix\ H_{C \oplus (a \to d)}(I) = fix\ H_C (I)$$

Proof

By applying the above two lemmas.

QED lemma

6.5. Incremental fixed point theorem

Theorem - incremental fixed point theorem

let $g(d) = (H_C (fix\ H_{C \oplus (a \to d)}))_a$
and $d_{fix} = fix\ g$
and $C_{fix} = C \oplus (a \to d_{fix})$
then
(i) $(fix\ H_C)_a = d_{fix}$
(ii) $fix\ H_C = fix\ H_{C_{fix}}$

Proof

First note that (ii) follows from (i) and the above lemma, therefore we only strictly need to prove (i), in fact we proceed with a little of each.
Steps -
(1) show that $d \leqslant d_{lhs} \Rightarrow g(d) \leqslant d_{lhs}$
where $d_{lhs} = (fix\ H_C)_a$
Proof step 1
$d \leqslant d_{lhs} \Rightarrow fix\ H_{C \oplus (a \to d)} \leqslant fix\ H_C$
– by above lemma
$\Rightarrow H_C \circ fix\ H_{C \oplus (a \to d)} \leqslant H_C \circ fix\ H_C = fix\ H_C$
– monotonicity of H_C
$\Rightarrow g(d) = (H_C \circ fix\ H_{C \oplus (a \to d)})_a \leqslant (fix\ H_C)_a$
QED (1)
(2) $\Rightarrow d_{fix} = fix\ g \leqslant d_{lhs}$
(3) $\Rightarrow fix\ H_{C_{fix}} \leqslant fix\ H_C$
– by above lemmas

(4) **let** $I = fix\ H_{C_{fix}}$
prove that $H_C\ I = I$
(4.1) show $(H_C\ I)_a = d_{fix}$
– immediate from def'n of g and $g(d_{fix}) = d_{fix}$
(4.2) $\Rightarrow H_{C_{fix}}\ I = H_C\ I$
– from previous lemma
(4.3) That is $I = (H_C\ I)$
QED (4)
(5) thus $fix\ H_{C_{fix}} = I \geqslant fix\ H_C$ – minimality of $fix\ H_C$
QED (ii)
(6) $fix\ H_C = H_C (fix\ H_C) = H_C (fix\ H_{C_{fix}})$
$\Rightarrow (fix\ H_C)_a = g(d_{fix}) = d_{fix}$
QED (i)

QED theorem

6.6. The algorithm

The algorithm is virtually identical to that for X_C^L -

$$X_C \;:\; Appl \rightarrow D$$

$$\begin{array}{lll} X_C|_a = C|_a & & a \in \mathbf{dom}\ C \\ X_C|_a = fix\ h & & \mathbf{otherwise} \end{array}$$

where

$$h(\ d\ ) = Bnd_C(\ a\ )(\ H\ X_{C \oplus (\ a \rightarrow d\ )}\ )_a$$

fix is used here to mean the computationally effective function -

$$fix \;:\; (\ D \rightarrow D\ ) \rightarrow D$$

$$fix(\ g\ ) \rightarrow fixit_g(\ \bot\ )$$

$$\begin{array}{l} fixit_g(\ d\ ) \rightarrow \\ \quad \mathbf{let}\ \ d_{new} = g(\ d\ ) \\ \quad \mathbf{if}\ \ d_{new} = d \\ \qquad \mathbf{then}\ \ d \\ \qquad \mathbf{else}\ \ fixit_g(\ d_{new}\ ) \end{array}$$

If the domain is finite and g always terminates then $fix(\ g\ )$ terminates. Further if g is monotonic then this function yields the minimum fixed point of g, and the formula for d_{new} can be simplified to $g(\ d\ )$. Note also that if D is a binary domain then the first application of g yields the fixed point. That is -

$$D \text{ binary} \Rightarrow fix\ g = g(\ \bot\ )$$

With this observation the algorithm collapses to simple pending analysis.

Bnd_C is the one point equivalent of B_C, that is -

$$Bnd_C(\ a\ )(\ d\ ) = (\ d \wedge C^{\top}(\ a\ )\ ) \vee C^{\perp}(\ a\ )$$

It is used to emphasise the locality of the calculation (it only involves C and a).

We now prove termination and correctness of X_C

6.7. Termination

Theorem

X_C terminates

Proof

Induction on size of C
We assume that X_C terminates for any C when $\|\ C\ \| > C$

Base case C total
In this case $X_C = C$ everywhere, and thus terminates.
QED base case

Inductive case, prove $X_C|_a$ terminates for all a,
there are two cases
either $a \in \mathbf{dom}\ C$
whence $X_C\ (\ a\ )\ \rightarrow\ C_a$ and terminates
or $X_C\ (\ a\ )\ \rightarrow\ fix\ h$
where $h(\ d\ )\ =\ Bnd_C\ (\ a\ )\ H\ X_{C \oplus (\ a -> d\ )}$
now $\| C \oplus (\ a -> d\ ) \| \geqslant \| C \|$
so $X_{C \oplus (\ a -> d\ )}$ terminates by induction, H terminates as given,
also $Bnd_C\ (\ a\ )$ terminates as it has only to search a finite
configuration and perform $\bigvee$s and $\bigwedge$s.
thus h terminates.
and *fix* always terminates on finite domains.
(Strictly need to prove h monotonic, this falls out of the correctness proof)
Thus $X_C\ (\ a\ )$ terminates.
QED inductive cases

QED theorem

6.8. Correctness

By correctness, we mean that X_C yields the same result as $fix\ H_C$, and in particular for any application of interest $X_{\{\}}(\ a\ )\ =\ (\ fix\ H\ )_a$. The proof is nearly identical to previous proofs.

Theorem

$$X_C \quad = \quad fix\ H_C$$

Proof

Induction on size of C
We assume that $X_C \quad = \quad fix\ H_C$ for any C when $\| C \| > C$

Base case C total
In this case H_C is constantly C and thus $fix\ H_C\ =\ C$
Similarly X_C is constantly C
QED base case

Induction
W want to prove $X_C\ (\ a\ )\ =\ (\ fix\ H_C\ )_a$
there are two cases :
either $a \in \mathbf{dom}\ C$
$X_C\ (\ a\ )\ \rightarrow\ C_a$
but $fix\ H_C|_{\mathbf{dom}\ C}\ =\ C$
QED $a \in \mathbf{dom}\ C$
or $a \notin \mathbf{dom}\ C$
$X_C\ (\ a\ )\ \rightarrow\ fix\ h$
where $h(\ d\ )\ =\ Bnd_C\ (\ a\ ) \circ (\ H\ X_{C \oplus (\ a->d\ )}\ )_a$
now $\|\ C \oplus (\ a->d\ )\ \|\ \geqslant\ \|\ C\ \|$
$\Rightarrow\ H\ X_{C \oplus (\ a->d\ )}\ =\ H\ fix\ H_{C \oplus (\ a->d\ )}$
and $Bnd_C\ (\ a\ )\ (\ (\ H(\ fix\ H_{C \oplus (\ a->d\ )}\ )\ )_a\ )\ =\ (\ B_C\ (\ H(\ fix\ H_{C \oplus (\ a->d\ )}\ )\ )_a$
That is h here is equal to g of the incremental fixed point theorem,
and as g is monotonic the $fix\ h$ is the minimal fixed point of g.
We thus apply part (i) of the theorem :-
$X_C\ (\ a\ )\ =\ fix\ g\ =\ (\ fix\ H_C\ )_a$
QED $a \notin \mathbf{dom}\ C$
QED inductive cases

QED theorem

7. An efficient algorithm for configuration searching

The speed of the above algorithm depends critically on the speed of the *Bnd* function and (equivalently) augmentation $C \oplus (\ a->d\ )$. A simple linear search of the configuration for all elements greater than or less than a given element will be too slow for large configurations. The simple limiting function L_C is easy to implement as this involves only looking for an exact match, it is the finding of related elements that is of difficulty. An algorithm is proposed that uses the lattice structure of the parameter domain as a means of structuring the configuration storage. Searching for elements above or below a certain element then involves a walk through the lattice starting at the top or bottom. If the lattice were complete, corresponding to a complete configuration, we could choose any path when there were several elements less than the target (assuming a walk down) as they would all eventually lead to the same target. If the configuration is not complete then we need to explore all paths (although some would later meet) in order to find all elements above the one of interest.

We can clearly index the configuration directly by the particular function, hence we can assume we are dealing with a configuration that only represents one function, and the *Appl* domain is identical to the argument domain D_a.

$$C\ =\ D_a\ \twoheadrightarrow\ D_d$$

The argument and algorithm is in no way dependent on this assumption as the application domain *Appl* has a perfectly good lattice structure of its own, however this would probably be a less efficient use of the algorithm. For other specific domains there may be similar specialisations that will improve performance.

We will look for a structure (call it *Env*) that represents the configuration. We will need several operations from this structure.

(i) Find $C^{\top}\ =\ \bigvee_{a' \geqslant a} C\ (\ a'\ )$

(ii) Find $C^{\perp}\ =\ \bigwedge_{a' \leqslant a} C\ (\ a'\ )$

(iii) Find $C \oplus (\ a->d\ )\ =\ (d \wedge C^{\top}(\ a\ )\ ) \vee C^{\perp}\ (\ a\ )$

That is we will want functions:

$$
\begin{aligned}
&\mathit{upper\_bound} \;:\; \mathit{Env} \times D_a \to D_d \\
&\mathit{lower\_bound} \;:\; \mathit{Env} \times D_a \to D_d \\
&\mathit{augment} \;:\; \mathit{Env} \times D_a \times D_d \to \mathit{Env}
\end{aligned}
$$

We will want these to mirror the actual operations, that is, if *env* represents C then:

$$
\begin{aligned}
&\mathit{upper\_bound}(\ \mathit{env}, a\ ) \;=\; C^{\top}\ (\ a\ ) \\
&\mathit{lower\_bound}(\ \mathit{env}, a\ ) \;=\; C^{\bot}\ (\ a\ ) \\
&\mathit{augment}(\ \mathit{env}, a, d\ ) \text{ represents } C \oplus (\ a -> d\ )
\end{aligned}
$$

Env will both represent the configuration itself, but also it will contain information specifically to improve the speed of the searches implied by the $\bigwedge$s and $\bigvee$s.

We represent the configuration value by a set of nodes, each one corresponding to an argument in the domain of C.

$$
\begin{aligned}
&\mathit{Env} \;\equiv\; \mathit{nodes} : \mathbb{P}\ \mathit{Node} \\
&\qquad\quad \mathit{arg} : \mathit{Node} \to D_a \\
&\qquad\quad \mathit{val} : \mathit{Node} \to D_d \\
&\textbf{where} \\
&\quad \textbf{dom}\ \mathit{arg} = \textbf{dom}\ \mathit{val} = \mathit{nodes} \\
&\quad \textbf{range}\ \mathit{arg} = \textbf{dom}\ C \\
&\quad \mathit{arg} \text{ is injective} \\
&\quad \forall\ n \in \mathit{nodes} \quad \mathit{val}(\ n\ ) = C\ (\ \mathit{arg}(\ n\ )\ )
\end{aligned}
$$

In order to search this structure, we will need to cross index these nodes giving for each node the set of nodes with arguments "just above" and "just below". This will give the nodes a structure like the lattice **dom** C.

$$
\begin{aligned}
&\mathit{Env} \;+\!=\; \mathit{up\_nodes} : \mathit{Node} \to \mathbb{P}(\ \mathit{Node}\ ) \\
&\qquad\quad \mathit{down\_nodes} : \mathit{Node} \to \mathbb{P}(\ \mathit{Node}\ ) \\
&\textbf{where} \\
&\quad \textbf{dom}\ \mathit{up\_nodes} = \textbf{dom}\ \mathit{down\_nodes} = \mathit{nodes} \\
&\quad \forall\ n \in \mathit{nodes} \quad \mathit{up\_nodes}(\ n\ ) \subseteq \mathit{nodes} \\
&\qquad\qquad\quad \mathit{down\_nodes}(\ n\ ) \subseteq \mathit{nodes} \\
&\quad \forall\ n, n' \in \mathit{nodes} \quad n' \in \mathit{up\_nodes}(\ n\ ) \;\Rightarrow \\
&\qquad\qquad\qquad \textbf{not}\ \exists\ n'' \in \mathit{nodes}\ \ \textbf{st}\ \ \mathit{arg}(\ n\ ) < \mathit{arg}(\ n''\ ) < \mathit{arg}(\ n'\ ) \\
&\quad \forall\ n, n' \in \mathit{nodes} \quad n' \in \mathit{down\_nodes}(\ n\ ) \;\Rightarrow \\
&\qquad\qquad\qquad \textbf{not}\ \exists\ n'' \in \mathit{nodes}\ \ \textbf{st}\ \ \mathit{arg}(\ n'\ ) < \mathit{arg}(\ n''\ ) < \mathit{arg}(\ n\ )
\end{aligned}
$$

We will need to record the topmost and bottommost nodes in order to start searches. We can do this explicitly :

$$
\begin{aligned}
&\mathit{Env} \;+\!=\; \mathit{top\_nodes} : \mathbb{P}(\ \mathit{Node}\ ) \\
&\qquad\quad \mathit{bottom\_nodes} : \mathbb{P}(\ \mathit{Node}\ ) \\
&\textbf{where} \\
&\quad \mathit{top\_nodes} = \{\ \mathit{node} \in \mathit{nodes} \mid \mathit{up\_nodes}(\ \mathit{node}\ ) = \{\}\ \} \\
&\quad \mathit{bottom\_nodes} = \{\ \mathit{node} \in \mathit{nodes} \mid \mathit{down\_nodes}(\ \mathit{node}\ ) = \{\}\ \}
\end{aligned}
$$

However this will lead to special cases when we are dealing with elements at the top or bottom of *Env*. An alternative is to add special elements to D_a and D_d, a "super top" ($\overline{\top}$) and "super bottom" ($\underline{\bot}$). These will sit above and below the normal top and bottom and thus any real argument will sit below the top node and above the bottom. So instead of the sets *top_nodes* and *bottom_nodes* we will have:

$Env \quad += \quad top\_node \ : \ Node$
$\qquad\qquad\quad bottom\_node \ : \ Node$
where
$\quad top\_node, bottom\_node \ \in \ nodes$
$\quad arg(\ top\_node\ ) \ = \ \overline{\top}_a$
$\quad arg(\ bottom\_node\ ) \ = \ \underline{\bot}_a$
$\quad val(\ top\_node\ ) \ = \ \overline{\top}_d$
$\quad val(\ bottom\_node\ ) \ = \ \underline{\bot}_d$

All three operations we wish to carry out depend on first finding the nodes directly above and below the target argument. That is we want functions :

$above \ : \ Env \times D_a \rightarrow \mathbb{P}(\ node\ )$
$below \ : \ Env \times D_a \rightarrow \mathbb{P}(\ node\ )$
where
$\quad above(\ env, a\ ) \subseteq nodes$
$\quad below(\ env, a\ ) \subseteq nodes$
$\quad$ **not** $\exists\ n \in nodes, n' \in above(\ env, a\ )$ **st** $a < arg(\ n\ ) < arg(\ n'\ )$
$\quad$ **not** $\exists\ n \in nodes, n' \in below(\ env, a\ )$ **st** $arg(\ n'\ ) < arg(\ n\ ) < a$

Given such functions we can easily specify the operations we want in terms of them.

$upper\_bound \ : \ Env \times D_a \rightarrow D_d$
$upper\_bound(\ env, a\ ) \ = \ \bigwedge_{n \in above(\ a\ )} val(\ n\ )$

$lower\_bound \ : \ Env \times D_a \rightarrow D_d$
$lower\_bound(\ env, a\ ) \ = \ \bigvee_{n \in below(\ a\ )} val(\ n\ )$

$augment \ : \ Env \times D_a \times D_d \rightarrow Env$
let $A = above(\ a\ ), \ B = below(\ a\ )$
if $A = B = \{\ n\ \}$ – that is $a \in$ **dom** C
$\quad augment(\ env, a, d\ ) \ = \ env$
otherwise – adding a new element
$\quad augment(\ env, a, d\ ) \ = \ env'$
$\quad$ **where**
$\quad\quad$ **dom** $env' \ = \ $ **dom** $env \ + \ \{\ n\ \}$
$\quad\quad arg'(\ n\ ) \ = \ a$
$\quad\quad val'(\ n\ ) \ = \ d$
$\quad\quad up\_nodes(\ n\ ) \ = \ A$
$\quad\quad down\_nodes(\ n\ ) \ = \ B$
$\quad\quad arg'(\ n'\ ) \ = \ arg(\ n'\ ) \qquad n' \neq n$
$\quad\quad val'(\ n'\ ) \ = \ val(\ n'\ ) \qquad n' \neq n$
$\quad\quad up\_nodes'(\ n'\ ) \ = \ up\_nodes(\ n'\ ) \qquad n' \neq n$ **and** $n' \notin B$
$\quad\quad up\_nodes'(\ n'\ ) \ = \ up\_nodes(\ n'\ ) \ - \ A \ + \ \{\ n\ \} \qquad n' \in B$
$\quad\quad down\_nodes'(\ n'\ ) \ = \ down\_nodes(\ n'\ ) \qquad n' \neq n$ **and** $n' \notin A$
$\quad\quad down\_nodes'(\ n'\ ) \ = \ down\_nodes(\ n'\ ) \ - \ B \ + \ \{\ n\ \} \qquad n' \in A$

It only remains to define effective algorithms for *above* and *below*. In fact, we can define two algorithms for each, depending on when we start at the top or the bottom.

$$
\begin{array}{lcl}
above^{\downarrow}_{Env}(\ a\ ) & = & above^{\downarrow}_{Env}(\ a, top\_node\ ) \\
above^{\downarrow}_{Env}(\ a, b\ ) & = & \textbf{let}\ \ S = \cup\{\ above^{\downarrow}_{Env}(\ a, c\ )\ |\ c \in down\_nodes(\ b\ )\ \textbf{and}\ \ a \leqslant arg(\ c\ )\ \} \\
 & & \textbf{if}\ \ S = \{\} \\
 & & \textbf{then}\ \ b \\
 & & \textbf{else}\ \ S
\end{array}
$$

$$
\begin{array}{lcl}
below^{\downarrow}_{Env}(\ a\ ) & = & below^{\downarrow}_{Env}(\ a, top\_node\ )\ \text{or}\ = \bigcup_{b \in above^{\downarrow}_{Env}(\ a\ )} below^{\downarrow}_{Env}(\ a, b\ ) \\
below^{\downarrow}_{Env}(\ a, b\ ) & = & \textbf{if}\ \ arg(\ b\ ) \leqslant a\ \ \textbf{and}\ \ \forall c \in up\_nodes(\ b\ )\ \ \textbf{not}\ \ arg(\ c\ ) \leqslant a \\
 & & \textbf{then}\ \ b \qquad\qquad \text{- that is}\ \ a \in \textbf{dom}\ C \\
 & & \textbf{else if}\ \ arg(\ b\ ) \leqslant a \\
 & & \textbf{then}\ \ \{\} \\
 & & \textbf{else}\ \ \bigcup_{c \in down\_nodes(\ b\ )} below^{\downarrow}_{Env}(\ a, c\ )
\end{array}
$$

$below^{\uparrow}_{Env}$ is the dual of $above^{\downarrow}_{Env}$, working from the bottom, and $above^{\uparrow}_{Env}$ is the dual of $below^{\downarrow}_{Env}$. The choice between using upward or downward searches could be made once and for all, or could be made depending on the exact value of a.

The cost of an application of $above^{\downarrow}_{Env}$ is proportional to the number of nodes with $arg(\ node\ ) \geqslant a$. In the worst case (looking for $\bot$) this is $\|\ nodes\ \|$ which is as bad as the simple search (but with a worse constant of proportionality). Typical cases are far better. The worst kind of domain is a simple linear one, when one visits half the nodes on average. However the argument domains will usually be product domains and here the behaviour is much better. For a product of n linear domains one would visit on average $n\,\frac{1}{2}^n$ of the nodes. Similarly the cost of $below^{\uparrow}_{Env}$ is proportional to the number of nodes with $arg(\ node\ ) \leqslant a$ with a worst case looking for $\top$.

This suggests that by examining where the target lies we could protect ourselves against worst case behaviour, but because the exact cost depends not only on a but also **dom** C it may not be worth it.

The above cost assumes that the unions can be performed in linear time implying a complex set representation (eg. hashing), but an imperative version of the algorithm would run down the paths in turn and mark nodes encountered which could then be avoided by subsequent paths.

In the incremental fixed point algorithm, the applications of Bnd_C and $C \oplus (\ a{-}{>}\ d\ )$ come in the expression *fix h* where h is defined as :

$$h(\ d\ ) = Bnd_C(\ a\ ) \circ (\ H\ X_{C \oplus (\ a->d\ )}\ )_a$$

Now on each iteration of *fix* we need to find out $above(\ a\ )$ and $below(\ a\ )$ to work out $C^{\top}(\ a\ )$ and $C^{\bot}(\ a\ )$ in $Bnd(\ a\ )$ and also to calculate $C \oplus (\ a{-}{>}\ d\ )$. These are the same both for these three uses in each iteration, but also between iterations. Therefore these can be calculated once and then reused on each iteration. That is we only need one configuration lookup for each fixed point calculation. Further as the number of iterations of the fixed point is itself on average half the domain height, cost of configuration search is likely to be far less than the cost of iteration. If you include the reduction in the number of iterations because we start above bottom (typically very near the fixed point), the algorithm looks very respectable.

If the height if the configuration is greater than the height of the result domain, it may be worth using frontier analysis techniques to compress "flat" chains. However care must be taken to keep note of those applications that are in the configuration but which are holophrasted for efficiency. For some algorithms even this may not be necessary.

In this section we have seen how configurations can be stored and accessed efficiently with a worst case behaviour comparable with simple searching, but with typical performance far better. However, it is also noted that we can reuse most of the work so that searches are relatively infrequent in the incremental fixed point algorithm, hence even worst case behaviour is likely to be acceptable.

8. Higher order analysis using term algebras

The two weaknesses of pending analysis were the limitation on the domain, and that it is a first order analysis. We have seen that it can be modified to cope with large domains, and thus of course if necessary function domains. However, these are large and comparisons over the domains, either of equality (for X_C^L) or inequality (for X_C) will be very expensive. In general, the number of distinct partial applications will be small compared to the total number of functions available. It would be nice therefore to use the terms describing the functions, instead of the functions themselves, and use these as the pending functions. For example, in the analysis of `map f` where `f` is some function, we would pend the application of `map` to the *identifier* `f`. This is equivalent to doing pending analysis over a domain of terms, and as the analysis is correct for all domains it is correct in particular for term algebras.

The only misgiving one might have, is that we are distinguishing terms that may be the same. One doubt arising from this is the impact on the efficiency of the algorithm as we miss some pending calls, however this is likely to be far less than the cost of full function comparison. More problematical is that it might affect the correctness of the algorithm. We know that it is correct within its own domain (term algebras), but is it correct in the original higher order domain. In fact this is ok, as the term algebra is a reified interpretation (opposite of abstract interpretation) of the original domain and functions. That is if *Term* is the evaluation operation from terms to values we have:

$$\forall\ f, x \qquad Term(f)\ \ Term(x) \ = \ Term(\ f\ x\ )$$

Of course the pending analysis only works if the domains are finite, the term domain is in principle infinite. Does this invalidate the procedure? In many cases it does not, as the terms generated are strictly bounded, the cases where pending analysis will produce unbounded terms are rare. They can only occur where a function can be applied to an irreducible term which includes a partial application of itself. The possibility of this can be detected by static analysis (yet more abstract interpretation) but this is not essential as dynamic measures are just as effective. When a term is generated which exceeds a certain complexity, we can substitute a "super-top" element ($\overline{\top}$) that has the property that any irreducible term containing it becomes super-top too, and when it is applied to anything, or is the argument to a primative operation, the result is larger than any other such application. That is:

$$\begin{aligned} \overline{\top}(\ a\ ) \ &= \ \bigvee_{f\ \leqslant\ \overline{\top}} f(\ a\ ) \\ f(\ \overline{\top}\ ) \ &= \ \bigvee_{a\ \leqslant\ \overline{\top}} f(\ a\ ) \end{aligned}$$

Or alternatively and cheaper, but less exact:

$$\begin{aligned} \overline{\top}(\ a\ ) \ &= \ \overline{\top} \\ f(\ \overline{\top}\ ) \ &= \ \overline{\top} \end{aligned}$$

With either of these we loose the exact correspondence between the normal interpretation and the reified term algebra, however both of these yield a relationship

$$\forall\ f, x \qquad Term(f)\ \ Term(x) \ \leqslant \ Term(\ f\ x\ )$$

Which is sufficient to yield a safe, but inexact, result of pending analysis. This is no worse in its effect than the pessimizing suggested by Young and Hudak to reduce evaluation time, and can similarly be varied by adjusting the acceptable complexity of terms before "topping out".

We do not need to change our proofs at all, except that proofs of properties of functionals defined using terms must include an *apply* operator, which is in fact monotonic on these domains and does not alter their correctness.

9. Fixed point properties of configurations

Young and Hudak suggest memoing the results of pending analysis as an optimisation. The resulting algorithm no longer has the simple functional form used previously, and it needs additional proof. However the memoing does suggest that one is iterating estimates to a whole configuration, rather than just at a point. This section looks at the fixed point properties of configurations in order to establish a general method of

proving correct optimisations of basic pending analysis.

C is a fixed configuration of H if
$\forall I \quad C \subseteq I \quad \Rightarrow \quad C \subseteq H\ I$

Equivalently (and easier to prove!)

C is a fixed configuration of H if
$C \subseteq H\ C$

Theorem

if C is a fixed configuration
$\exists\ I$ a fixed point of H (not necessarily minimal)
st $C \subseteq I$

Proof

let $I_0|_{\mathbf{dom}\ C} = C$
$I_0|_a = \bot$ elsewhere
let $I_{i+1} = H\ I_i$
then
$C \subseteq I_0$
$\forall i \quad C \subseteq I_i \quad \Rightarrow \quad C \subseteq I_{i+1}$
$\Rightarrow \quad C \subseteq \bigvee I_i$
and $\bigvee I_i$ is a fixed point of H

QED

So if C is a fixed configuration and $C \subseteq fix\ H$ then we can conclude that $C = fix\ H|_{\mathbf{dom}\ C}$.

This is very useful for proving algorithms correct. We need only prove that any value obtained is part of a fixed configuration to know that the value is safe (i.e. above the fixed point). This is an end state result and is not dependent on the internal workings of the algorithm. To prove the algorithm optimal (that is equal to the minimum fixed point) will typically involve a *process* argument that the minimal fixed point is never exceeded, however since it is a weak (i.e. $\leqslant$) argument it is easier to work with than an exact argument and it is thus easier to involve clever heuristics.

If we are happy with safe results, we need only prove that

$\forall I \quad C \gg I \quad \Rightarrow \quad C \gg H\ I$
or
$C \gg H\ C$

whence by a similar proof to the above we have $C \gg fix\ H$

Fixed configurations can be related to *minimal function graphs*.[6] Jones and Mycroft describe minimal function graphs in terms of a domain with two "bottom" elements: ! representing non termination and $\bot$ representing "never called". Roughly their ! corresponds to values of $\bot$ in the configuration, and their $\bot$ to values which are undefined.

10. A non-deterministic algorithm based on configurations

The pending analysis algorithms proved have been functional, they have not involved memoising as suggested by Young and Hudak, and have sometimes been conservative (Bnd_C may reduce the value returned). This is because they have relied on proving that intermediate results have been equal to fixed points of constrained iterators. The various optimisations will typically yield results that are not exactly the fixed point of the constrained iterator but lie between it and the actual fixed point required. We could go through all the proofs again and prove the various optimised versions. The above result about fixed configurations would make this process far easier. It is fairly obvious that if we take all the application result pairs from the *last* iteration of all the various embedded fixed point calculations that they would form a fixed configuration. we would thus only need to prove that all the values obtained are below the fixed point. This would

probably require a $\geqslant$ version of the incremental fixed point theorem, but with all the lemmas developed involving B_C this would be easy.

An easy way to prove such optimisations is to prove correct a non-deterministic algorithm which includes the optimisations of interest as special sub-cases. Rather than doing this for top-down pending analysis we examine an alternative non-deterministic algorithm that is based around bottom-up evaluation of configurations.

Define a sequence

let C_i be a sequence of configurations defined thus:
C_0 any configuration such that $C_0 \ll fix\ H$
for each i either
(1) choose $a \notin \mathbf{dom}\ C,\ d \leqslant (\ fix\ H\ )|_a$
$C_{i+1} = C_i \uparrow (\ a \rightarrow d\ )$
$\Rightarrow \quad C_{i+1} \ll fix\ H$
(2) choose $C \subseteq C_i$
then choose $C' \subseteq H\ C$
$C_{i+1} = C_i \uparrow C'$
$\Rightarrow \quad C_{i+1} \ll fix\ H$
(3) choose any monotonic $C_{i+1} \ll C_i$
$\Rightarrow \quad C_{i+1} \ll fix\ H$

Obtain result

for any i choose $C \subseteq C_i$ a fixed configuration
$\Rightarrow \quad C \subseteq fix\ H$

The operator $\uparrow$ is the symmetric upward sum, defined thus:

Define - $C \uparrow C'$

$\mathbf{dom}\ (\ C \uparrow C'\ ) = \mathbf{dom}\ C \cup \mathbf{dom}\ C'$
$(\ C \uparrow C'\ )|_a = C|_a \vee C'|_a$

The various algorithms used can all be seen as variants of this. Consider, for example, general pending analysis with the bounding function. It uses the $\oplus$ operator, however this can be described as step 1 or 2 followed by step 3. To avoid saying this we will describe the variant of the algorithm using the $\uparrow$ operator.

The algorithm starts off with an empty configuration. Each time a new value is encountered step 1 is executed with $d = \bot$ then the fixed point calculation begins. Effectively doing multiples of step 2. As the functional algorithm does not remember internal results of recursive calls, step 3 is effectively executed at each function return. By simply omitting the "forgetful" step 3, we produce pending analysis with memoing.

However, we can imagine far cleverer heuristics based on this algorithm. In general pending analysis can spend effort calculating values that do not contribute to the final result (but are needed for intermediate values). If instead of pushing execution right down the call tree, we could as a first estimate return bottom, for everything, then (if the pending value has been used) iterate on this value alone, only when this has stopped do we go round the other values used to make sure they are correct. This means that some applications low in the domain may not need to be iterated over. We can retain dependency information in order to decide which values may have changed, and also remember critical paths, so that we can first iterate on applications likely to lead to an increase in the target application.

There are clearly many ways of approaching this, but typically algorithms will have one or more target applications that they're actually after. If the call graph of the application involves elements not in the configuration, step 1 will be executed to extend it. Then by examining the call graph an application is chosen as "pivot" and step 2 is iterated until stable or the result of the pivot increases enough to change one of the targets. In order to execute step 2 of course the configuration is typically further extended etc.

Following these heuristics the fixed configuration output will be precisely the minimal function graph associated with the target applications. Other values will have been calculated of course, as it is impossible in general to calculate the minimal function graph without doing some "wasted" work.

A memoised algorithm, that is one without step 3 can be shown to terminate so long as one is reasonably sensible about choices of C'. Algorithms using step 3 need separate proofs, but why bother? Possibly, in some cases we could prove that certain applications will never be needed again and its worth reducing the size of configuration, but its hard to believe that this will frequently be worthwhile.

By defining a non-deterministic algorithm we have left room for many different optimisations to basic pending analysis, some of which may have improved typical behaviour. Strangely enough, we end up with an algorithm far closer to a traditional fixed point calculation.

11. Conclusions

Proofs have been given of Young and Hudak's pending analysis for non-self applicatory functions and of the improved algorithm for general and higher order functions. In addition a new non-deterministic algorithm has been presented and proved correct. This algorithm not only includes both pending analysis and standard algorithms as special cases, but also lays the formal groundwork for more advanced algorithms. This unification of approaches is especially encouraging as it gives a framework for future study.

An open problem is whether simple pending analysis is in fact correct for general functions. Examples, such as in Appendix 7, make one very doubtful, however pending analysis does still work for this. It would be nice to either form a true counter example or find a proof however the examples here show that such a proof would be tortuous. My gut feeling is that perhaps it is correct in the binary case, but it is clear that gut feelings in this area can be very misleading.

Appendix 1 - Recursion equations for multiple functions

We have called the domain of the function sought *Appl* as it is expected that it typically represents the application of a function symbol to some arguments. Below is briefly outlined the formal basis of this. None of the proofs depended on the well typedness of the operations involved, and if the relevant definitions are initially well typed then the result of the various algorithms will themselves be well typed.

The abstract interpretation will be over some set of recursively defined functions mapping into finite lattices. The functions form a signature $< F, T, arity >$:

F : a set of function symbols
T : a set of type symbols

$$arity \;:\; F \rightarrow T^* \times T$$

The domains of interest are labelled by these types, and for convenience we consider also the disjoint sum of these domains.

$$\{ D_\tau \}_{\tau \in T}$$
$$D = \sum_{\tau \in T} D_\tau$$

The set of all type correct applications of function symbols to members of D^* we will call *Appl*:

$$Appl = \{ f d_1 d_2 .. d_n \mid f \in F \textbf{ and } d_i \in D_{\tau_i} \}$$
where
$$arity(f) = (\tau_1, \tau_2 .. \tau_n), \tau$$

This inherits a partial ordering from the domains:

$$f d_1 d_2 .. d_n \leqslant f' d_1' d_2' .. d_n' \quad \equiv \quad f = f' \textbf{ and } d_1 \leqslant d_1' \textbf{ and } d_2 \leqslant d_2' \ \ldots \ d_n \leqslant d_n'$$

Any interpretation is a Σ-algebra over this signature. And the set of all such interpretations we call *Interp*.

$$Interp \;=\; Appl \;\rightarrow\; D$$

subject of course to the well typing condition:

$$\forall a = f d_1 d_2 .. d_n \in Appl,\; I \in Interp$$
$$I(\,a\,) \;\in\; D_\tau$$
$$\textbf{where}\;\; arity(\,f\,) \;=\; (\,\tau_1, \tau_2\, ..\, \tau_n\,), \tau$$

The recursion equations can be represented as a function on interpretations:

$$G \;:\; Interp \;\rightarrow\; Interp$$

These equations are of course usually represented by terms over the function symbols, but we do not need this additional structure for the first part of the analysis. The only properties we require is that G is monotonic and continuous. (We will not need to invoke continuity in our arguments explicitly as all the domains of interest are finite, and it will thus be implicitly true of all monotonic functions.)

The problem we are after solving is therefore finding $fix\ G$, the minimal fixed point of G.

Appendix 2 - Facts used about fixed points

In this paper we have used the fixed point operator (called *fix*) extensively. Its properties are well known, and those used in the paper are summarise here.

The minimal fixed point of a function f is the smallest value x such that $f\ (\,x\,) \;=\; x$. Thus we have the important conclusion that:

$$f\ (\,x\,) \;=\; x \quad\Rightarrow\quad fix\ f \;\leqslant\; x$$

A standard result is that the minimal fixed point of a function f is given by:

$$fix\ f \;=\; \bigvee x_i$$
$$\textbf{where}$$
$$x_0 \;=\; \bot$$
$$x_{i+1} \;=\; f(\,x_i\,)$$

This depends on the function f being *monotonic* and *continuous* however, as the lattices we will consider will always be finite the continuity condition need not concern us.

Two facts about *fix* that result fairly immediately from the above are:

(i) $x \;\leqslant\; fix\ f \quad\Rightarrow\quad f(\,x\,) \;\leqslant\; fix\ f$
(ii) $(\,x \leqslant y \;\Rightarrow\; f\ x \leqslant y\,) \quad\Rightarrow\quad fix\ f \;\leqslant\; y$
(iii) $(\,x \leqslant fix\ f$ **and** $x \leqslant fix\ g \;\Rightarrow\; f\ x \leqslant g\ x\,) \quad\Rightarrow\quad fix\ f \;\leqslant\; fix\ g$
(iv) $x \;\geqslant\; f(\,x\,) \quad\Rightarrow\quad x \;\geqslant\; fix\ f$

Appendix 3 - Proof that non-self applicatory functionals defined by terms are fully monotonic

We want to prove that if a functional G is defined by terms and is not self-applicatory, then it is monotonic. So we need to prove:

$$I \;\leqslant\; I' \quad\Rightarrow\quad G\ I \quad\leqslant\quad G\ I'$$

Now if G is defined by non-self applicatory terms, for any a we have

$$G\ I\ |_a \;=\; Eval_I\ H_a^1$$

Where H_a^1 is a non-self applicatory term. So it suffices to prove the more general theorem below:

Theorem

$I \leqslant I'$ and *term* is non-self applicatory
$\Rightarrow \quad Eval_I(\ term\ ) \leqslant Eval_{I'}(\ term\ )$

Proof

By induction on $depth(\ term\ )$
Base case $depth(\ term) = 0$
$term = d$
$Eval_I(\ term\ ) = d = Eval_{I'}(\ term\ )$
QED base case

Inductive case $depth(\ term\ ) > 0$
there are two subcases
either $term = op\ e_1 \ldots e_n$
where the e_i are themselves non-self applicatory
$depth(\ e_i\ ) < depth(\ term\ )$
$\Rightarrow \quad v_i = Eval_I(\ e_i\ ) \leqslant Eval_{I'}(\ e_i'\ ) = v_i'$
$\Rightarrow \quad Eval_I(\ term\ ) = \square_{op}\{\ v_i\ \} \leqslant \square_{op}\{\ v_i'\ \} = Eval_{I'}(\ term\ )$
QED $op\ e_1 \ldots e_n$
or $term = f\ e_1 \ldots e_n$
where the e_i are constant terms
$\Rightarrow \quad v_i = Eval_I(\ e_i\ ) = Eval_{I'}(\ e_i'\ )$
$\Rightarrow \quad I(\ \{\ v_i\ \}\ ) \leqslant I'(\ \{\ v_i\ \}\ )$
QED $f\ e_1 \ldots e_n$
QED inductive case

QED

Appendix 4 - Proof that all functionals defined by terms are $\succ$ monotonic and pseudo-monotonic

To prove pseudo-monotonicity, we first prove a stronger result, namely monotonicity with respect to a new relation $\succ$. The relation $\succ$ is stronger than normal function comparison, but equivalent to it when either of its arguments are monotonic.

Definition $\succ$

$$f \succ f' \quad \equiv \quad \forall\, a, a' \quad a \geqslant a' \quad \Rightarrow \quad f(\ a\ ) \geqslant f'(\ a'\ )$$

Note how this differs from normal $\geqslant$ relation which only covers $a = a'$. Similarly note how if either of f or f' is monotonic, we can introduce an intermediate "pivot" term ($f(\ a'\ )$ or $f'(\ a\ )$ respectively) to prove $\succ$ from $\geqslant$.

$\succ$ is not a true partial order in itself, as it is transitive, but not reflexive. The case where $f \geqslant f$ is of interest being exactly when f is monotonic. If we require a true partial order we can always add an "or equal" condition to $\succ$.

$$f \succcurlyeq f' \quad \equiv \quad f \succ f' \quad \textbf{or} \quad f = f'$$

The lattice formed by $\succcurlyeq$ has a rather complicated least upper bound operator:

$$a \bigvee^{\succcurlyeq} b \quad = \quad \textbf{if} \quad a = b \quad \textbf{then} \quad a \qquad \textbf{else} \quad ceil(\ a\ ) \bigvee ceil(\ b\ )$$

The ceiling function means that it is non-local. The fixed point operator is better behaved, and if f is $\succ$ (or $\succcurlyeq$) monotonic and the domain is finite then the standard fixed point operator can be used. Consider the sequence

$$x_0 = \bot, \quad x_{i+1} = f\ x_i$$

This is $\succ$ (or $\succcurlyeq$) increasing and hence will terminate at the minimal fixed point of f. In the case of $\succ$

monotonicity, this means that this fixed point will be a monotonic function, even if the intermediate x_i are not.

As functionals defined by terms are $\succ$ monotonic, I had at first hoped that this would give an easy proof of simple pending analysis for general functions, unfortunately the limiting functional L_C is not $\succ$ monotonic, and changes to make it so would mean something of at least the complexity of B_C however, it does seem an interesting ordering waiting for an application.

We prove $\succ$ monotonicity first and then use this to prove pseudo-monotonicity as a special case.

$\succ$ monotonicity

We want to prove that if a functional G is defined by terms it is $\succ$ monotonic, that is:

$$I \succ I' \Rightarrow G\ I \succ G\ I'$$

Now if G is defined by terms, for any a we have

$$G\ I\ |_a = Eval_I\ H^1_a$$

So it suffices to prove the more general theorem below:

Theorem

$$I \succ I' \Rightarrow \forall\ term \geqslant term' \quad Eval_I(\ term\ ) \geqslant Eval_{I'}(\ term'\ )$$

Proof

By induction on $depth(\ term\ )$
Base case $depth(\ term) = 0$
$term = d,\ term' = d'$
$Eval_I(\ term\ ) = d \geqslant d' = Eval_{I'}(\ term'\ )$
QED base case

Inductive case $depth(\ term\ ) > 0$
there are two subcases
either $term = op\ e_1\ ..\ e_n,\ term' = op\ e_1'\ ..\ e_n',\ e_i \geqslant e_i'$
$depth(\ e_i\ ) < depth(\ term\ )$
$\Rightarrow\ v_i = Eval_I(\ e_i\ ) \geqslant Eval_{I'}(\ e_i'\ ) = v_i'$
$\Rightarrow\ Eval_I(\ term\ ) = \square_{op}\{\ v_i\ \} \geqslant \square_{op}\{\ v_i'\ \} = Eval_{I'}(\ term\ )$
QED $op\ e_1\ ..\ e_n$
or $term = f\ e_1\ ..\ e_n,\ term' = f\ e_1'\ ..\ e_n',\ e_i \geqslant e_i'$
$v_i \geqslant v_i'$ as above
$\Rightarrow\ I(\{\ v_i\ \}) \geqslant I'(\{\ v_i'\ \})$ - as $I \succ I'$
QED $f\ e_1\ ..\ e_n$
QED inductive case

QED

Pseudo-monotonicity

We want to prove that if a functional G is defined by terms it is pseudo-monotonic, that is:

$$I \geqslant I' \text{ and either } I \text{ or } I' \text{ monotonic} \Rightarrow G\ I \geqslant G\ I'$$

But if $I \geqslant I'$ and either I or I' is monotonic then $I \succ I'$ and thus $G\ I \succ G\ I'$ which implies (and is equivalent in this case to) $G\ I \geqslant G\ I'$.

Appendix 5 - Properties of L_C

We define L_C for all configurations by:

Definition L_C

$$\textbf{dom}\ (\ L_C\ C'\ ) \quad = \quad \textbf{dom}\ C'$$

$$\begin{array}{lll} L_C\ C'\ |_a & = & C_a \qquad a \in C \\ & = & C'_a \qquad \textbf{otherwise} \end{array}$$

Lemma

L_C is fully monotonic ($\ll$) as a function *Config* $\rightarrow$ *Config*

Proof

$C_1 \ll C_2$
let $C_1' = L_C\ C_1,\ C_2' = L_C\ C_2$
$\textbf{dom}\ C_1' = \textbf{dom}\ C_1 \subseteq \textbf{dom}\ C_2 = \textbf{dom}\ C_2'$
$a \in \textbf{dom}\ C \quad \Rightarrow \quad C_1'|_a = C|_a = C_2'|_a$
$a \in \textbf{dom}\ C_1 - \textbf{dom}\ C \quad \Rightarrow \quad C_1'|_a = C_1|_a \leqslant C_2|_a = C_2'|_a$

QED

Lemma

L_C is continuous ($\ll$)

Proof

Consider $\{\ C_\lambda\ \}_{\lambda \in \Lambda},\ C_\Lambda = \bigvee C_\lambda$
$\textbf{dom}\ C_\Lambda = \bigcup C_\lambda$
$C_\Lambda|_a = \bigcup \{\ C_\lambda|_a\ \textbf{st}\ a \in \textbf{dom}\ C_\lambda\ \}$
let $C_\lambda' = L_C\ C_\lambda,\ C_\Lambda' = L_C\ C_\Lambda$
$\textbf{dom}\ C_\Lambda' = \textbf{dom}\ L_C\ C_\Lambda = \bigcup \textbf{dom}\ C_\lambda = \bigcup \textbf{dom}\ L_C\ C_\lambda'$
$a \in \textbf{dom}\ C \quad \Rightarrow \quad C_\Lambda'|_a = C|_a = \bigcup C_\lambda'|_a$
$a \in \textbf{dom}\ C_\lambda - \textbf{dom}\ C \quad \Rightarrow \quad C_\Lambda'|_a = C_\Lambda|_a = \bigcup C_\lambda|_a = \bigcup C_\lambda'|_a$

QED

Appendix 6 - Functions defined using self partial application

Higher order pending analysis using terms had to be inexact where there was no bound to the complexity of terms produced. This can occur quite easily (and usefully) in weakly typed functions, for instance `map_n` that takes a function to be applied to a list of lists of depth n and returns a homomorphic list of lists.

```
map_n f 0 l     =     f l
map_n f n l     =     map_n (map f) n-1 l
```

`map` is the normal function applying `f` to each element of a list. An application of `map_n f n l` will lead to a term of the form

```
map_n (map (map ( ... (map f) ... )) 0 l
```

Where the depth of nesting of `map`s is `n`, before it is finally reduced. Without limiting the depth of terms acceptable, pending analysis would not terminate for this example.

`map_n` is of course not Milner typeable, there are however type correct functions which exhibit similar properties. Let `twice` be a function, that takes a function and an argument, and applies the function twice

```
twice   :   ( * -> * ) -> * -> *

twice f x    =    f ( f x )
```

We then use this to define a function `lots` that applies a function to an argument lots of times

```
lots   :   nat -> ( * -> * ) -> * -> *

lots 0 f x    =    f x
lots n f x    =    lots n-1 ( twice f ) x
```

The application `lots n f x` will lead to a term:

```
lots 0 ( twice ( twice  ..  ( twice f ) .. )) x
```

Where there are `n` applications of `twice` (leading to 2^n applications of `f`!), before it eventually reduces. Again pending analysis would fail without a size limit on the term algebra.

Typically functions defined in this manner are rather obtuse. Consider two more such functions:

```
lots_more   :   nat -> ( * -> * ) -> * -> *

lots_more 0 f x    =    f x
lots_more n f x    =    lots_more n-1 ( lots_more n-1 f ) x

strange   :   ( nat -> nat ) -> nat -> nat

strange f x 0    =    f x
strange f x y    =    strange f x y-1                          y odd
strange f x y    =    strange (strange f y-1) x y/2            y even
```

Appendix 7 - a nasty non-self applicatory binary function

We have not proved that simple pending analysis works for simple binary functions works, but here is an example where a "reasonable" intermediate result fails. If pending analysis is applied to this function, it does however work!

The reasonable result is:

$$G\ G_{A+\{a\}}\ fix\ G|_a \quad \leqslant \quad (\ G\ G_A\ fix\ G\ )|_a$$

Consider the function:

$$f(\ x, z\ ) \quad = \quad z\ \vee\ f(\ f(\ x, z\ ), \top\ )$$

Clearly if G is the generator $fix\ G\ =\ \top$, however we find that

$$(\ G\ G_{a'+a}\ fix\ G\ )|_a \quad \geqslant \quad (\ G\ G_{a'}\ fix\ G\ )|_a$$

where

$a \quad = \quad \top, \bot$

$a' \quad = \quad \top, \top$

This is the opposite of our intuition that fixing some values to $\bot$ reduces the final result. The reason for this is of course the self applicatory expression $f(\ f(\ x, z\ ), \top\ )$, on the left hand side this leads to an evaluation of $f(\ a''\ )$, where $a''\ =\ \bot, \top$, which is not pending, hence reduces to $\top$, this value then bubbles up to give a final answer of $\top$. On the right however the recursive call is to $f(\ a'\ )$, which leads to a result of $\bot$.

The exact call chain is below, the function applications are labelled with their level numbers and primed on the right, so that f_1' is an outer most call on the right, whereas f_3 is an innermost ($fix\ G$) call on the left.

$$\begin{aligned}
\text{LHS} \ = \ & f_1(\ a\ ) \ \rightarrow \ \bot\ \vee\ f_2(\ f_2(\ a\ ), \top\ ) \\
& \rightarrow \ \bot\ \vee\ f_2(\ \bot, \top\ ) && \text{- } a \text{ pending} \\
& \rightarrow \ \bot\ \vee\ f_3(\ \bot, \top\ ) && \text{- } a'' \text{ not pending} \\
& \rightarrow \ \bot\ \vee\ \top\ \rightarrow\ \top
\end{aligned}$$

$$
\begin{aligned}
\text{RHS} = \; & f_1'(a) \;\rightarrow\; \bot \vee f_2'(f_2'(a), \top) \\
& \rightarrow\; \bot \vee f_2'(f_3'(a), \top) && \text{- } a \text{ not pending} \\
& \rightarrow\; \bot \vee f_2'(\top, \top) \\
& \rightarrow\; \bot \vee \bot && \text{- } a' \text{ pending} \\
& \rightarrow\; \bot
\end{aligned}
$$

References

1. Jonathon Young and Paul Hudak, "Finding fixpoints on function spaces," YALEU/DCS/RR-505, Yale University, Department of Computer Science (December 1986).
2. P. Cousot and R. Cousot, "Static determination of dynamic properties of programs" in *Proceedings of the 2nd International Symposium on Programming* (1976).
3. A. Mycroft, *Abstract interpretation and optimising transformations for applicative programs,* PhD thesis, University of Edinburgh (1981).
4. S. Abramsky and C. Hankin, *Abstract interpretation of declarative languages,* Ellis Horwood (1987).
5. C. Clack and S. Peyton-Jones, "Finding fixpoints in abstract interpretation" in *Abstract interpretation of declarative languages,* ed. S. Abramsky & C. Hankin, pp. 246-265, Ellis Horwood (1987).
6. N.D. Jones and A. Mycroft, "Data flow analysis of applicative programs using minimal function graphs" in *Proceedings 13th Symposium on Principles of Programming Languages,* pp. 296-306, ACM (January 1986).

Finding fixed points in non-trivial domains: Addendum to York Report 107

Alan Dix

Full Reference

A. J. Dix (1988). *Addendum to York Report 107: Finding fixed points in non-trivial domains: proofs of pending analysis and related algorithms*. YCS 107 (addendum), Dept. of Computer Science, University of York.

`http://alandix.com/academic/papers/fixpts-YCS107-88/`.NH 1 Introduction

In York Report 107, I proved the correctness of pending analysis for non-self-applicatory functions, over both binary and general finite (height) domains. I also proved slight variants of basic pending analysis, that make better use of the monotonicity of functions correct for all functions defined using monotonic primatives. This later algorithm could be seen as an optimisation of the basic algorithm as it in general would require fewer iterations, but depending on the method used to represent pending environments the simpler algorithm may be better. The report failed to prove the basic algorithm correct for self-applicatory functions, even over the original binary domain; on the other hand it also failed to find any counter examples. Since then I have found a simple proof of the correctness of binary pending analysis, making use of the correctness of the "improved" algorithm. A similar proof works for all finite domains, but needs yet another incremental fixed point theorem! These proof are given below.

The proofs are split into two parts:

Optimality - the resulting function is no greater than the minimal fixed point

Safety - it is no lower

Safety is clearly most important for abstract interpretation, but is the more difficult. Optimality is proved in both the binary and finite domain case by direct comparison with the minimal fixed point. It is "obviously" true at one level, as the pending analysis makes values smaller, but it also makes the intermediate functionals non-monotonic whence nothing is obvious anymore.

Both optimality and safety proofs depend on the pseudo-monotonicity of the defining functional, that is, if $f \leqslant g$ and either f or g is monotonic then $G\ f \leqslant G\ g$. There are examples of monotonic functionals which are not pseudo-monotonic, for example:

$$G \quad : \quad (2 \to 2) \to (2 \to 2)$$

$$\begin{array}{lcll} G\ f & = & \top & f \text{ monotonic} \\ & = & \bot & \textbf{otherwise} \end{array}$$

G is a monotonic functional, with fixed point $f = \top$, unfortunately pending analysis yields the answer $f =$ **not**, which is neither monotonic, fixed nor safe. However, any functional which is defined using terms and monotonic primatives is pseudo-monotonic. (That is all functions produced by programs.)

Because pseudo-monotonicity requires one of the functions to be monotonic, the proofs for pending analysis have to work by comparing its non-monotonic intermediate functions with other monotonic functions that have known properties. In the case of optimality this can be the fixed point itself, however more work is required for safety.

In addition to producing the new proofs for simple pending analysis, the fixed point properties of configurations and the non-deterministic algorithm are reconsidered. Corrections are given for slips in the proofs in YCS107 and an updated algorithm is given which reflects the techniques used in the proofs in this paper, relaxing monotonicity conditions.

Note: the user is assumed to have a copy of YCS107 as several definitions of operators etc. are not repeated here.

1. Optimality of binary and general pending analysis

Recall that standard binary pending analysis is the function $X_{\{\}}$, where

$$\begin{array}{lll} X_A|_a & = & \bot \qquad\qquad a \in A \\ & = & (\ G\ X_{A+\{a\}}\ )|_a \qquad \textbf{otherwise} \end{array}$$

Where G is the defining functional and A is the set of application values that are "pending".

We want to prove that $X_{\{\}} \leqslant fix\ G$, to do this we prove in general:

Theorem - optimality of binary pending analysis

$$X_A \leqslant fix\ G$$

Proof

Induction on size A.
Base case A complete
$X_A = \bot \leqslant fix\ G$
QED base case

Inductive case.
Sub-cases
case $a \in A$
$X_A|_a = \bot \leqslant fix\ G|_a$
QED $a \in A$
case $a \notin A$
$X_A|_a = G\ X_{A+\{a\}}|_a$
but
$X_{A+\{a\}} \leqslant fix\ G$ - induction
$\Rightarrow\ G\ X_{A+\{a\}} \leqslant G\ fix\ G = fix\ G$ - pseudo-monotonicity
QED $a \notin A$
QED inductive cases

QED Theorem

Notice the pseudo-monotonicity step, that would not allow us for instance to prove $X_{A+\{a\}} \leqslant X_A$.

The proof for general finite height domains is similar, except in that the definition involves a recursion:

$$\begin{array}{lll} X_C^L|_a & = & C(a) \qquad a \in \textbf{dom}\ C \\ X_C^L|_a & = & fix\ h \qquad \textbf{otherwise} \end{array}$$
where
$$h(d) = (\ H\ X^L_{C+(a \to d)}\ )|_a$$

Here H is used for the defining functional when the domain is non-binary. C is a configuration, that is a partial function that represents the set of assumptions made about the target function during the course of the algorithm.

Note that h is in general not monotonic, so the standard meaning of fixed point does not apply. This point will be dealt with later, however in the mean time dangerous steps are marked !FIX!.

Again we want to prove that $X_C^L \leqslant fix\ H$. We need to include the obvious proviso that $C \ll fix\ H$

Theorem - optimality of non-binary pending analysis

$$C \ll \mathit{fix}\ H \ \Rightarrow\ X^L_C \ \leqslant\ \mathit{fix}\ H$$

Proof

Induction on size C.
Base case C complete
$X^L_C = C \leqslant \mathit{fix}\ H$
QED base case

Inductive case.
Sub-cases
case $a \in \mathbf{dom}\ C$
$X^L_C|_a = C(a) \leqslant \mathit{fix}\ H|_a$
QED $a \in A$
case $a \notin \mathbf{dom}\ C$
$X^L_C|_a = \mathit{fix}\ h$
where
$h(d) = (H\ X^L_{C+(a\to d)})|_a$
but
$d \leqslant \mathit{fix}\ H|_a$
$\Rightarrow\ C+(a\to d) \ll \mathit{fix}\ H$
$\Rightarrow\ X^L_{C+(a\to d)} \leqslant \mathit{fix}\ H$ - induction
$\Rightarrow\ H\ X^L_{C+(a\to d)} \leqslant H\ \mathit{fix}\ H$ - pseudo-monotonicity
$\Rightarrow\ h(d) \leqslant \mathit{fix}\ H|_a$
therefore $\mathit{fix}\ h \leqslant \mathit{fix}\ H|_a$ - !FIX!
QED $a \notin A$
QED inductive cases

QED Theorem

These two proofs are straight-forward and need none of the results from YCS107 except for the pseudo-monotonicity of functions defined by terms.

2. Safety of binary pending analysis

We want to prove that $X_{\{\}} \geqslant \mathit{fix}\ G$, to do this we prove that in general:

$$A \subseteq A' \ \Rightarrow\ X_A \geqslant Z_{A'}$$

Z_A is the improved algorithm defined by:

$$\begin{array}{lll} Z_A|_a = \bot & & a \in A \\ = (G\ Z_{A\oplus\{a\}})|_a & & \textbf{otherwise} \end{array}$$

where $A \oplus \{a\}$ is A extended by a and then downwards closed. That is:

$$A \oplus \{a\} = A \cup \{\, a' \mid a' \leqslant a \,\}$$

In YCS107 we proved that $Z_A = \mathit{fix}\ G_A$, where G_A is defined as follows:

$$\begin{array}{lll} G_A\ I\ |_a = \bot & & a \in A \\ = G\ I|_a & & \textbf{otherwise} \end{array}$$

So $Z_{\{\}} = \mathit{fix}\ G_{\{\}} = \mathit{fix}\ G$. Thus the general result will be sufficient for safety of pending analysis.

Theorem - safety of binary pending analysis

$$A \subseteq A' \;\Rightarrow\; X_A \geqslant Z_{A'}$$

Proof

Induction on size A.
Base case A complete $\Rightarrow$ A' complete
$X_A = \bot = Z_{A'}$
QED base case

Inductive case.
Sub-cases
case $a \in A'$
$X_A|_a \geqslant \bot = Z_{A'}|_a$
QED $a \in A'$
case $a \notin A' \Rightarrow a \notin A$
$X_A|_a = G\ X_{A+\{a\}}|_a$
$Z_{A'}|_a = G\ Z_{A' \oplus \{a\}}|_a$
but
$A + \{a\} \subseteq A' \oplus \{a\}$
$\Rightarrow X_{A+\{a\}} \geqslant Z_{A' \oplus \{a\}}$ - induction
$\Rightarrow G\ X_{A+\{a\}} \geqslant G\ Z_{A' \oplus \{a\}}$ - pseudo-monotonicity
QED $a \notin A'$
QED inductive cases

QED Theorem

3. Safety of non-binary pending analysis

We might hope to prove a similar result for non-binary functions:

$$C \geqslant C' \;\Rightarrow\; X^L_C \geqslant X_{C'}$$

Unfortunately this fails. If we follow the same proof procedure as for optimality we need to prove $fix\ h \geqslant fix\ h'$ where:

$$h(\ d\ ) = (\ H\ X^L_{C+(\ a \to d\ )}\ )|_a$$
$$h'(\ d\ ) = (\ H_C\ X_{C' \oplus (\ a \to d\ )}\ )|_a$$

This is in general not true because H_C can "hoist" up the value at a if **dom** C contains elements comparable with a.

We can proceed however by creating a new functional H^F_C which is monotonic and has an incremental fixed point theorem, but which doesn't hoist up the target value. We will find we can prove that $X^L_C \geqslant fix\ H^F_{C'}$

The new functional is defined in a similar manner to H^L_C and H_C except with a different bounding functional F_C:

$$H^F_C = F_C\ H$$
$$F_C\ I = C^\top \wedge I$$

That is F_C simply bounds above by C whilest retaining monotonicity. Notice too that $H^F_{\{\}} = H$, so that a safety comparison with H^F_C will prove safety of $X^L_{\{\}}$. Note too that like H_C (and unlike H^L_C) H^F_C requires that the configuration C be monotonic.

The incremental fixed point theorem for H^L_C merely added to the configuration ($C + (\ a \to d\ )$), whereas H_C needed a special augmentation operator ($C \oplus (\ a \to d\ )$). The similar theorem for H^F_C needs an adding operator ($C \wedge (\ a \to d\ )$) that bounds above but which preserves monotonicity:

$$\mathbf{dom}\ (\ C \wedge (\ a \rightarrow d\ )\ ) \quad = \quad \mathbf{dom}\ C \ +\ \{\ a\ \}$$

$$C \wedge (\ a \rightarrow d\ )|_a \quad = \quad d \quad \wedge \quad \bigwedge_{a' \geqslant a} C\ (\ a'\ )$$

$$C \wedge (\ a \rightarrow d\ )|_{a'} \qquad = \quad d \quad \wedge \quad C\ (\ a'\ )\ a' \ \leqslant \ a$$
$$= \quad C\ (\ a'\ ) \qquad\qquad \text{otherwise}$$

Essentially $C \wedge (\ a \rightarrow d\ )$ is the biggest monotonic configuration less than both C and $\{\ a \rightarrow d\ \}$.

The incremental fixed point theorem is familiar enough by now. Its proof is nearly identical in form to that for H_C except that all the lemmas needed about B_C are trivialy true for F_C. We will label the steps as for the proof for H_C to emphasise similarity.

Theorem - incremental fixed point theorem for H^F_C

let $g(\ d\ ) \ = \ (\ H^F_C\ (\ fix\ H^F_{C \wedge (\ a \rightarrow d\ )}\ )\ )_a$
and $d_{fix} \ = \ fix\ g$
and $C_{fix} \ = \ C \wedge (\ a \rightarrow d_{fix}\ )$
then
(i) $(\ fix\ H^F_C\ )_a \ = \ d_{fix}$
(ii) $fix\ H^F_C \ = \ fix\ H^F_{C_{fix}}$

Proof

In the proof for H_C we needed to prove $d_{lhs} \leqslant d_{fix}$ in order to prove that $fix\ H_{C_{fix}} \leqslant fix\ H_C$, however this is obvious for H^F_C as $H^F_{C_{fix}}$ is simply bounded above by a smaller configuration and $H^F_{C_{fix}} \leqslant H^F_C$, so we can jump straight away to:
(3) $fix\ H^F_{C_{fix}} \leqslant fix\ H^F_C$

We now need to prove the opposite inequality:
(4) **let** $I \ = \ fix\ H^F_{C_{fix}}$
prove that $H^F_C\ I \ = \ I$
(4.1) $(\ H^F_C\ I\ )_a \ = \ d_{fix}$
– immediate from def'n of g and $g(\ d_{fix}\ ) \ = \ d_{fix}$
(4.2) $\Rightarrow \quad H^F_{C_{fix}}\ I \ = \ H^F_C\ I$
– as (4.1) $\Rightarrow\ H^F_C\ I \ \leqslant \ (\ C \wedge (\ a \rightarrow d'\ )\ )^{\top}$
(4.3) That is $I \ = \ (\ H^F_C\ I\ )$
QED (4)
(5) thus $fix\ H^F_{C_{fix}} \ = \ I \ \geqslant \ fix\ H^F_C$ – minimality of $fix\ H^F_C$
QED (ii)
(6) $fix\ H^F_C \ = \ H^F_C\ (\ fix\ H^F_C\ ) \ = \ H^F_C\ (\ fix\ H^F_{C_{fix}}\ )$
$\Rightarrow \quad (\ fix\ H^F_C\ )_a \ = \ g(\ d_{fix}\ ) \ = \ d_{fix}$
QED (i)

QED theorem

The final step is to prove the correctness of X^L_C by comparison with $fix\ H^F_C$

Theorem - correctness of non-binary pending analysis

$$C \ \geqslant \ C' \ \Rightarrow \quad X^L_C \quad \geqslant \quad fix\ H^F_{C'}$$

Proof

Induction on size C.
Base case C complete $\Rightarrow$ C' complete
$X^L_C \ = \ C \ \geqslant \ C' \ \geqslant \ fix\ H^F_{C'}$
QED base case

Inductive case.
  Sub-cases
    case $a \in \mathbf{dom}\ C$
      $X^L_C|_a \;=\; C(a) \;\geqslant\; C'(a) \;\geqslant\; fix\ H^F_{C'}|_a$
    QED $a \in A$
    case $a \notin \mathbf{dom}\ C$
      $X^L_C|_a \;=\; fix\ h$
      where
        $h(d) \;=\; (H\ X^L_{C+(a\to d)})|_a$
      but
        $\forall\ d \quad X^L_{C+(a\to d)} \;\geqslant\; fix\ H^F_{C'\wedge(a\to d)}$ - induction
        $\Rightarrow \quad H\ X^L_{C+(a\to d)} \;\geqslant\; H\ fix\ H^F_{C'\wedge(a\to d)}$ - pseudo-monotonicity
                $\geqslant \quad H^F_{C'}\ fix\ H^F_{C'\wedge(a\to d)}$
      so
        $h \geqslant g$ - where g is the function in the fixed point theorem
        $\Rightarrow \quad fix\ h \;\geqslant\; fix\ g \;=\; (fix\ H^F_{C'})_a$ - !FIX!
    QED $a \notin A$
QED inductive cases

QED Theorem

Strangely enough there appears to be no pending like algorithm that is equal to $fix\ H^F_C$ as there is for $fix\ H_C$. This is because, even when we know a value is in the configuration C, we only know that $fix\ H^F_C$ is lower than this (rather than equal as with H^L_C and H_C). Even if such an algorithm did exist, it would not be very interesting, as it would be "worse" than simple pending analysis in that it would converge slower and be more complex. X_C is of course a useful algorithm as although it is more complex, it has better convergence.

4. Errata - Fixed point properties of monotonic configurations

Note the proof given in YCS107 was intended for monotonic functionals and monotonic configurations. In fact as stated, it starts iterating at a non-monotonic configuration so works instead for fully-monotonic functionals over possibly non-monotonic configurations! The iteration in the proof should start at $C^\perp$, and is then correct.

5. Fixed point properties of non-monotonic configurations

We can produce a fixed point theorem for non-monotonic configurations, namely:

Theorem

$$C \ll HC \quad \Rightarrow \quad C \ll fix\ H$$

Proof

let $f_0 = \bot$
$f_{i+1} = H\ f_i$
be the standard sequence to give $fix\ H$.
We want to prove inductively that $C \gg f_i$

base case
$f_0 = \bot$ and $C \gg \bot$
QED base case
inductive case
by inductive hypothesis $C \gg f_i$
$\Rightarrow$ $HC \gg H\ f_i$ - pseudo-monotonicity
$\Rightarrow$ $C \gg f_{i+1}$
QED inductive case

So if we can construct (by some means!) a configuration with $C \gg HC$ it is a safe (albeit non-monotonic) approxiomation to $fix\ H$. If in addition one can show by some alternative argument that $C \ll fix\ H$, then the configuration will be exactly $fix\ H|_{\mathbf{dom}\ C}$.

6. Errata - Non-deterministic algorithm using monotonic configurations

The definition of $C \uparrow C'$ should read:

$$\mathbf{dom}\ (C \uparrow C') = \mathbf{dom}\ C \cup \mathbf{dom}\ C'$$

$$(C \uparrow C')|_a = C^\top|_a \vee C'^\top|_a$$

That is the least monotonic configuration greater than both C and C'. Simple least upper bound is used below for the non-monotonic version.

7. Non-deterministic algorithm allowing non-monotonic configurations

With the above version of the fixed point theorem for non-monotonic configurations we can mirror the algorithm used in YCS107 to allow non-monotonic intermediate configurations. As with the case for using standard pending analysis as opposed to the version using monotonicity it depends on the usage and implementation which is better, but at least this allows the use of a simpler algorithm if desired.

The algorithm is nearly the same as the monotonic version, except that it uses simple configuration upper-bound ($C \vee C'$) rather than the upward sum ($C \uparrow C'$) defined above. $C \vee C'$ is in fact the operator defined wrongly in YCS107 just to confuse things.

$$\mathbf{dom}\ (C \vee C') = \mathbf{dom}\ C \cup \mathbf{dom}\ C'$$

$$(C \vee C')|_a = C|_a \vee C'|_a$$

In addition this algorithm makes no monotonicity provision at step 3.

Define a sequence

let C_i be a sequence of configurations defined thus:
C_0 any configuration such that $C_0 \ll fix\ H$
for each i either
(1) choose $a \notin \mathbf{dom}\ C,\ d \leqslant (\ fix\ H\ )|_a$
$C_{i+1} = C_i \vee (\ a \rightarrow d\ )$
$\Rightarrow \quad C_{i+1} \ll fix\ H$
(2) choose $C \subseteq C_i$
then choose $C' \subseteq HC$
$C_{i+1} = C_i \vee C'$
$\Rightarrow \quad C_{i+1} \ll fix\ H$ -pseudo-monotonicity
(3) choose any (possibly non-monotonic) $C_{i+1} \ll C_i$
$\Rightarrow \quad C_{i+1} \ll fix\ H$

Obtain result

for any i choose $C \subseteq C_i$ such that $C \gg HC$
$\Rightarrow \quad C \subseteq fix\ H$

Again if step 3 is never used (and why should it be?) the algorithm is bound to terminate as the sequence C_i is increasing. With suitable instantiation this furnishes an alternative proof of the various algorithms above. In addition it allows for optimised versions of the algorithm.

8. Meaning of fixed point for non-monotonic functions

For non-monotononic functions, we cannot use the standard fixed point algorithm without comment:

$$d_0 = \bot$$
$$d_{i+1} = f(\ d_i\ )$$
$$fix\ f = \bigvee d_i$$

We can construct the set d_i as before, but it is not in general increasing, and the least upper bound need not be fixed. This is particularly nasty when viewed as an effective algorithm in a finite domain as the sequence d_i need not stabalise, leading to possible non-termination. There are several definitions we can use to extend the operator *fix* to non-monotonic functions however:

(i) the definition used above

(ii) use d_i as above but use $\bigvee_n \bigwedge_{i>n} d_i$

(iii) use d_i as above but use $\bigwedge_n \bigvee_{i>n} d_i$

(iv) define a new increasing sequence

$$d_0 = \bot$$
$$d_{i+1} = f(\ d_i\ ) \vee d_i$$
$$fix\ f = \bigvee d_i$$

All these definitions are equivalent to standard *fix* on monotonic functions. (i), (ii) and (iii) all share the termination problem, so an alternative proof of termination is necessary. (iv) is an increasing sequence, so is bound to terminate. Note that using (iv) we have $f(\ fix\ f\ ) \leqslant fix\ f$. The other definitions have no such nice properties. It is not necessarily true that (iv) ascends faster than the rest, although it looks as though it should. However in non-perverse examples it probably will "on average"!

Two properties are used of *fix* in the above theorems, both are true for all definitions:

(1) $(\ d \leqslant d' \ \Rightarrow\ f(\ d\ ) \leqslant d'\ ) \ \Rightarrow\ fix\ f \leqslant d'$

(2) *fix* is pseudo monotonic, that is:

$f \leqslant g$ and one of f or g is monotonic
$\Rightarrow$ $fix\ f \leqslant fix\ g$

The former is a simple induction on d_i. The pseudo-monotonicity is only slightly more subtle,

Proof - pseudo-monotonicity of *fix* (all defn's)

let d_i be the sequence for f and c_i be the sequence for g
(using whichever definition we are interested in)
we want to show by induction that $d_i \leqslant c_i$
base case
$d_0 = \bot = c_0$
QED base case
inductive case
by inductive hypothesis $d_i \leqslant c_i$
if f is monotonic then
$f(d_i) \leqslant f(c_i) \leqslant g(c_i)$
otherwise g is monotonic and
$f(d_i) \leqslant g(d_i) \leqslant g(c_i)$
either way $f(d_i) \leqslant g(c_i)$ whence by whichever
definition we use we get $d_{i+1} \leqslant c_{i+1}$
QED inductive case
in each case the construction of $fix\ f$ from d_i uses standard
operations, so $fix\ f \leqslant fix\ g$
QED theorem

9. Conclusions

If I had this proof before writing YCS107, it would have been shorter and simpler. However, it would have been a pity not to have the incremental fixed point theorem for B_C, and in general to have missed the breadth and depth of analysis. Also I would have been unlikely to have produced the non-deterministic algorithm which promises far more power and flexibility than the other algorithms considered.

It is very nice to finally have no loose ends dangling.

Appendix - Notation index

The following is an index to most of the notation used in this report and YCS107.

Symbol	Report	Section	Description
$Appl$	YCS107	§2.1	domain of possible arguments
D	""	""	domain of possible results
$Interp = Appl \rightarrow D$	""	""	type of target function
G	""	""	defining functional for binary target function
A	""	""	set of values from $Appl$ used to record pending arguments for binary pending analysis
G_A	""	""	functional used for binary pending analysis
$f\|_S$	""	""	the function f restricted to the set S, $f\|_a = f(a)$
pseudo-monotonic	""	§2.2	functionals obeying monotonicity property when only one function argument is monotonic
X_A	""	§2.3	function generated by pending analysis
fully-monotonic	""	§2.4	functionals monotonic regardless of the monotonicity of their function argument
Z_A	""	§2.5	function generated by modified algorithm which preserves monotonicity for all binary functionals
$A \oplus \{a\}$	""	""	A extended by a and then downwards closed
C	""	§4	a configuration, that is a partial function to record assumptions made about the target function during pending analysis of non-binary functions
$C + (\ a \rightarrow d\ )$	""	""	simple configuration extension
$C \leqslant C'$ and $\geqslant, \subseteq, \ll, \gg$	""	""	various partial orderings over configurations
H	""	""	defining functional used as a function over configurations
X_C^L	""	§5	simple pending analysis function for non-self-applicatory functions
H_C^L	""	§5	version of the defining functional used in the simple pending analysis algorithm
L_C	""	§5	limiting functional used to make the resulting function agree with C over its domain
$C^\top, C^\perp$	""	§6.2	bounding configurations about C
$C \oplus (\ a \rightarrow d\ )$	""	§6.2	augmentation of C similar to simple extension $Cadd$, but preserving monotonicity.
B_C	""	§6.3	bounding functional, similar in purpose to L_C, but ensuring that its result is a monotonic function
H_C	""	§6.4	the constrained iterator, a version of the defining functional which ensures that the resulting function is monotonic
X_C	""	§6.6	the function generated by the algorithm using H_C
Bnd	""	§6.6	the one point equivalent of the functional B_C
$C \uparrow (\ a \rightarrow d\ )$	""	§10	symetric upward sum of configurations
$C \uparrow (\ a \rightarrow d\ )$	here	§7	corrected version of above
$\succ$	YCS107	Ax 4	"nearly" partial order, over which *all* normal functionals are monotonic
H_C^F, F_C	here	§4	constrained iterator and associated iterator used to prove correctness of simple pending analysis for non-binary functions
$C \wedge (\ a \rightarrow d\ )$	here	§4	monotonic glb operator for adding to configurations
$C \vee C'$	here	§8	simple symetric, not necessarily monotonic least upper bound operator for configurations
fix	here	§9	various generalisations of the fixed point operator for use with non-monotonic functions