AN INDUCTIVE APPROACH TO FINDING FIXPOINTS IN ABSTRACT INTERPRETATION

Nigel Jagger*
Department of Computer Science
University of York
York, YO1 5DD
United Kingdom

Abstract: A new method for finding *fixpoints* in *abstract interpretation*, based on the technique of *computational induction*, is presented. This technique has the advantage that, unlike other existing techniques, it allows elements of the fixpoint to be computed separately without restricting the size of abstract domains which can be handled. Since abstract interpretation typically only requires one or two elements of the fixpoint in order to test some property of a program, it is expected that in the general case this method will perform better that any of the other methods.

The well known technique of *strictness analysis* is used to introduce abstract interpretation. After a short review of existing techniques for finding fixpoints, the method based on computational induction is presented. It is first shown how to apply this to strictness analysis, then the more general case is considered. How to deal with nested function applications and mutual recursion is demonstrated next. Analyses involving higher-order functions are discussed briefly and then the overall correctness of the approach is examined. Finally, to illustrate the method's practical appeal, an example of an analysis featuring non-flat domains is presented.

1 Introduction

The technique of *abstract interpretation* is used to determine properties of programs without recourse to formal proof (see [2] for a comprehensive introduction). It involves assigning a non-standard and approximate semantics to a program in terms of various *monotonic* and *continuous* operators defined over a suitably simplified *abstract domain*. The domain is designed to model the properties being examined and the operators express the corresponding behaviour of the program primitives. The process of abstract interpretation may be viewed as consisting of two separate phases. The first phase involves replacing the constants and primitives in the program by their abstract counterparts and, typically, this is fairly straightforward. The second phase, where the properties of the program are actually tested, requires discovering the *least fixpoint* of the abstract program and is considerably more complicated, as shall be seen shortly. To illustrate the overall process, the familiar abstract interpretation, known as strictness analysis, will now be considered.

A function is said to be *strict* in an argument if, whenever the argument is undefined, the result of applying the function to that argument is undefined. More formally, if ω denotes some undefined computation, then an n-ary function f is strict in its ith argument if forall $e_1, \ldots, e_{i-1}, e_{i+1}, \ldots, en$

$$f\ e_1 \cdots e_{i-1}\ \omega\ e_{i+1} \cdots e_n \equiv \omega \tag{1}$$

Mycroft[11] was the first to present an abstract interpretation for strictness analysis and it has been extensively used in the implementation of lazy functional languages (see, for example, [12]).

Typically, strictness analysis uses a two-point domain $\bot \sqsubseteq \top$, where $\bot$ denotes undefined objects, and $\top$ denotes objects which may be defined. Primitives in the program are expressed in terms of the domain operators *meet* and *join*, denoted by $\sqcap$ and $\sqcup$, respectively. Assuming a simple first-order non-strict functional language, with the conditional being defined in the usual way, the translation phase of the analysis may be implemented by the following set of rewrite rules.

$$c \longrightarrow \top, \quad \text{if } c \text{ is a constant eg. } 0, True$$
$$x\ op\ y \longrightarrow x \sqcap y, \quad \text{if } op \text{ is a strict binary operator eg. } *, =$$
$$if\ x\ then\ y\ else\ z \longrightarrow x \sqcap (y \sqcup z)$$

Although the above rules only approximate the true semantics of the program primitives, they at least give a *safe* interpretation. This assures that an analysis can never falsely detect strictness. The operators $\sqcap$ and $\sqcup$ possess useful properties, such as commutativity and associativity, which may be exploited during an analysis. Indeed, there is a one to one correspondence between $\sqcap$ and $\sqcup$ and the connectives $\wedge$ and $\vee$ of predicate logic, where $\bot$ and $\top$ are identified with *false* and *true*, respectively. This means that all of the laws which apply to $\sqcap$ and $\sqcup$ may also be used to simplify abstract expressions.

To discover if the function f, above, is strict in its ith argument, the rewrite rules are applied to f in order to obtain its abstract form $f^{\#}$. The corresponding functional $F^{\#}$ is then obtained from $f^{\#}$ by the inclusion of an extra parameter, in the usual way. Having determined the least fixpoint $fix\ F^{\#}$ (see below), the function's strictness is tested by seeing if

$$fix\ F^{\#}\ \top \cdots \top \bot \top \cdots \top \equiv \bot \tag{2}$$

where the lone $\bot$ occurs in the ith argument position. If this holds then strictness follows by a simple monotonicity argument.

Formally, the *least fixpoint* of a monotonic and continuous functional is defined in terms of the limit of an ascending chain of approximations, as follows.

$$fix\ F \equiv \bigsqcup_{n=1}^{\infty} (F^n\ \Omega) \tag{3}$$

(Here, Ω denotes the completely undefined function and $\bigsqcup$ denotes the *least upper bound*.) Since the domains used in abstract interpretation are normally assumed to be finite, the chain of approximations $F^n\ \Omega$ will converge to render the fixpoint in finite n. This provides an obvious method for finding the fixpoint, simply find the first $n \geq 0$ such that

$$F^n\ \Omega \equiv F^{n+1}\ \Omega$$

*Supported under SERC Grant GR/E16571.

To see how this analysis works in practise, an example of strictness detection will now be worked through.

Example 1

Consider the recursive function $fact$, below, which computes the factorial of its first argument, accumulating the result as a product of its second argument.

$$fact\ n\ m\ =\ if\ n = 0\ then\ m\ else\ fact\ (n-1)\ (n * m)$$

Now, suppose it is required to discover if the function is strict in its second argument. Applying the above rewrite rules to $fact$ renders its abstract form $fact^{\#}$.

$$fact^{\#}\ n\ m\ =\ (n \sqcap \top) \sqcap (m \sqcup fact^{\#}\ (n \sqcap \top)\ (n \sqcap m))$$

This may be simplified using the law that $\top$ is an identity of $\sqcap$.

$$fact^{\#}\ n\ m\ =\ n \sqcap (m \sqcup fact^{\#}\ n\ (n \sqcap m))$$

The functional for the current example is formed as follows.

$$Fact^{\#}\ f\ n\ m\ =\ n \sqcap (m \sqcup f\ n\ (n \sqcap m))$$

Since $\sqcap$ and $\sqcup$ are monotonic and continuous, it follows that $Fact^{\#}$ will also be monotonic and continuous, therefore it is appropriate to use successive approximation to determine the fixpoint.

$$\begin{aligned} Fact^{\#^0}\ \Omega\ n\ m &\equiv \bot \\ Fact^{\#^1}\ \Omega\ n\ m &\equiv n \sqcap m \\ Fact^{\#^2}\ \Omega\ n\ m &\equiv n \sqcap m \\ &\vdots \end{aligned}$$

It is here that the point of convergence has been reached, hence the least fixpoint of $Fact^{\#}$ is

$$fix\ Fact^{\#}\ n\ m\ \equiv\ n \sqcap m$$

Returning now to the problem of discovering if the function is strict in its second argument; $fix\ Fact^{\#}\ \top\ \bot$ returns $\bot$, therefore $fact$ is strict. Notice that $fact^{\#}\ \top\ \bot$ cannot be computed directly as it would never terminate; it is to ensure termination, that the fixpoint is constructed.

In the above example, laws about $\sqcap$ and $\sqcup$ were used to render each approximation in the chain into a canonical form, so that they could then be compared for convergence. This turns out to be a viable way of finding fixpoints in simple strictness analysis, indeed, Martin[8] has shown that it is almost possible to fully implement a procedure for detecting convergence "syntactically". His approach is to use conjunctive or disjunctive normal forms as the canonical representation for the approximations. However, problems arise when one attempts to apply this method to higher-order functions and non-flat domains. In fact it turns out to be virtually impossible to extend this technique beyond simple strictness analysis.

The more usual approach to finding fixpoints involves performing some kind of pointwise comparison, which has the advantage that it imposes no restrictions on the sorts of domains which can be handled. The naive method is to simply generate complete tables of argument/result pairs for the intermediate approximations, this, however, proves to be prohibitively expensive in practise. More realistic strategies concentrate on providing compact representations for the approximations, probably the best known example being the *frontiers method*, which was first presented by Peyton Jones and Clack[13] and later improved upon by Martin and Hankin[10]. It uses a very efficient representation for the intermediate values, namely the boundary between $\bot$s and $\top$s in the table, and also exploits monotonicity properties in order to minimise the cost of comparing successive approximations. Although this represents a considerable improvement on the naive approach, it still involves monitoring a sizeable portion of the table, and, as was stated earlier, abstract interpretation typically only requires one or two elements of the fixpoint to test some property of a program.

A much more direct method for finding fixpoints is *pending analysis*[16]. This allows elements of the fixpoint to be computed separately, thus avoiding the problems of scale associated with the frontiers method. Young and Hudak observed that, in a two-point domain, it is safe to return $\bot$ from an abstract function call with the same arguments as one of its ancestors in the call graph, which overcomes the problems of non-termination, as identified in the preceding example. Several authors have attempted to extend pending analysis to larger domains, (eg. [5,9,16]), but the consensus seems to be that this is not generally possible. What is required then is a method which combines the best of both worlds ie. a fast direct analysis which works with non-flat domains, and to this end, this paper now explores computational induction as a mechanism for handling fixpoints.

2 Computational Induction

Computational induction is a procedure for proving properties of the least fixpoints of recursive programs (see [7] for a fuller treatment). It is given by the following theorem.

Theorem 1: *For a monotonic and continuous functional F and an admissible predicate P, $P\ (fix\ F)$ holds if*

- *$P\ \Omega$ holds, and*
- *forall n, $P\ (F^n\ \Omega) \vdash P\ (F^{n+1}\ \Omega)$.*

2.1 Admissibility

Computational induction is only valid for certain classes of predicates, the so called *admissible* predicates. Roughly speaking, a predicate is admissible if it is continuous at limit points. More precisely, P is admissible if

$$\forall n (P\ (F^n\ \Omega)) \Longrightarrow P\ (\bigsqcup_{n=1}^{\infty} (F^n\ \Omega)) \qquad (4)$$

There are various classes of predicates which are known to uphold this property (see [7] for further details). One such class includes P of the form

$$P\ f:\ \bigwedge_{i=1}^{n} (\alpha_i\ f \sqsubseteq \beta_i\ f) \qquad (5)$$

for finite n, where the α_i, β_i are monotonic and continuous functionals. It turns out that the kind of predicates needed for finding fixpoints in abstract interpretation are of the form

$$P\ f:\ f\ d_1 \cdots d_n \equiv r$$

where $d_1, \ldots, dn$ and r are elements from finite domains and ranges. Re-expressing this as a conjunction of inequalities give

$$P\ f:\ \alpha_1\ f \sqsubseteq \beta_1\ f\ \wedge\ \alpha_2\ f \sqsubseteq \beta_2\ f$$

where α_1, α_2, β_1 and β_2 are defined as follows.

$$\begin{aligned}\alpha_1\ f &= f\ d_1 \cdots d_n\\ \alpha_2\ f &= r\\ \beta_1\ f &= r\\ \beta_2\ f &= f\ d_1 \cdots d_n\end{aligned}$$

Since, α_1, α_2, β_1 and β_2 are clearly monotonic and continuous, it follows that this class of P must be admissible.

2.2 Strictness Analysis

To illustrate how the above may be used in abstract interpretation, the following reconsiders the example of strictness analysis presented earlier. Recall that it was required to discover if $fact$ was strict in its second argument. In abstract terms, this entailed finding the value of $fix\ Fact^{\#}\ \top\ \bot$. Stating this as a predicate, what is needed is to prove $P\ (fix\ Fact^{\#})$, where P is defined by

$$P\ f:\ f\ \top\ \bot \equiv \bot$$

If $P\ (fix\ Fact^{\#})$ holds then the function is strict, otherwise it must be assumed that it is not.[1] Since P is admissible, this can be proved using computational induction by first showing that $P\ \Omega$ holds, which is trivial, and then showing that

$$P\ (Fact^{\#^n}\ \Omega) \vdash P\ (Fact^{\#^{n+1}}\ \Omega)$$

Unfolding $Fact^{\#}$, gives

$$Fact^{\#^n}\ \Omega\ \top\ \bot \equiv \bot \vdash \top \sqcap (\bot \sqcup Fact^{\#^n}\ \Omega\ \top\ (\top \sqcap \bot)) \equiv \bot$$

which simplifies to

$$Fact^{\#^n}\ \Omega\ \top\ \bot \equiv \bot \vdash Fact^{\#^n}\ \Omega\ \top\ \bot \equiv \bot$$

Clearly the conclusion follows from the hypothesis, hence, it has been proved inductively that $fact$ is strict in its second argument. There are close similarities between this approach and the technique of pending analysis discussed earlier. Applying an hypothesis in an inductive proof corresponds to identifying and breaking a cycle in an abstract computation. Viewed in this way, the inductive approach may be seen as a more formal justifitation for why pending analysis works. In order to extend the inductive method to larger domains, a fairly major reworking is needed, and it is here that most of the similarities with pending analysis are lost.

3 More General Abstract Interpretations

The above procedure is not immediately applicable to more general abstract interpretations. The problem is that for any predicate which asserts anything other than "$\equiv \bot$", the proof fails to negotiate the base case $P\ \Omega$. The solution then is to use computational induction to identify the point at which the required element of the fixpoint has converged. Suppose, for some abstract functional $F^{\#}$, it is required to find the value of $fix\ F^{\#}\ d_1 \cdots d_n$. All that has to be done is to find the least $i \geq 0$, such that forall j

$$F^{\#^i}\ \Omega\ d_1 \cdots d_n \equiv F^{\#^{i+j}}\ \Omega\ d_1 \cdots d_n \qquad (6)$$

Notice that it is not sufficient simply to determine the least $i \geq 0$ such that

$$F^{\#^i}\ \Omega\ d_1 \cdots d_n \equiv F^{\#^{i+1}}\ \Omega\ d_1\ \cdots d_n \qquad (7)$$

[1] Failure to detect strictness does not imply that the function is non-strict, only that its strictness cannot be decided using abstract interpretation.

because of the existence of *plateaux* (cf. [4]).

There are various ways of finding the correct value of i. One possibility is to just enumerate i until (6) holds, however, although this is guaranteed to succeed eventually, it is potentially very inefficient. A better method is to concentrate on the plateaux of $F^{\#^i}\ \Omega\ d_1 \cdots d_n$. That is, only values of i which satisfy (7) need be tried. In this way, the proof simply acts as a check to see if a plateau corresponds to the fixpoint. Although this is an improvement on enumeration, better still may be achieved by allowing the proof to dictate values of i to try.

A proof of (6) is tried with i initially set to 0. Typically, this will present requirements for some of the other elements of the fixpoint. Each additional requirement is then incorporated into the proof and initialised, as above. This often leads to some form of contradiction in the proof, which may be taken as a hint as to more profitable values of i to try out. The proof is continually refined in this manner until no further contradictions arise and all of the requirements have been accounted for. The precise details of this are probably best conveyed by working through an example.

Example 2

In this example, the four-point chain $0 \sqsubseteq 1 \sqsubseteq 2 \sqsubseteq 3$ will be taken as the abstract domain. Let $p^{\#}$ and $q^{\#}$ be monotonic operators defined over the domain as follows.

$p^{\#}$	0	1	2	3
0	0	0	1	2
1	1	1	2	3
2	2	3	3	3
3	3	3	3	3

$q^{\#}$	0	1	2	3
	1	2	2	2

Now consider the abstract functional $F^{\#}$, below, defined in terms of $p^{\#}$ and $q^{\#}$.

$$F^{\#}\ f\ x\ y = p^{\#}\ x\ (f\ y\ (q^{\#}\ x))$$

Suppose, it is required to find the value of $fix\ F^{\#}\ 2\ 1$. Since, $F^{\#}$ is monotonic and continuous and this corresponds to an admissible predicate, it is appropriate to use computational induction. The first step assumes $i = 0$ and therefore involves trying to prove that the above is 0. In the following, explicit base cases are omitted from the proofs, as they follow immediately. For simplicity of notation, compositions of the form $F^{\#^n}\ \Omega$ will just be abbreviated $f^{\#}$.

$$(1)\quad f^{\#}\ 2\ 1 \equiv 0 \ \vdash\ p^{\#}\ 2\ (f^{\#}\ 1\ 2) \equiv 0$$

To make any further progress, the value of $f^{\#}\ 1\ 2$ is required. A proof clause asserting this to be 0 is now added to the existing proof.

$$\begin{aligned}(1)\quad & f^{\#}\ 2\ 1 \equiv 0, \ \vdash\ p^{\#}\ 2\ (f^{\#}\ 1\ 2) \equiv 0,\\ (2)\quad & f^{\#}\ 1\ 2 \equiv 0 \qquad\quad p^{\#}\ 1\ (f^{\#}\ 2\ 2) \equiv 0\end{aligned}$$

This still constitutes a valid inductive proof because it is known that conjunction is admissible. Applying hypothesis (2) in conclusion (1) produces the result 1, which in turn generates a contradiction. This provides a hint as to what value of $f^{\#}\ 2\ 1$, (ie. i), to try next.

$$\begin{aligned}(1)\quad & f^{\#}\ 2\ 1 \equiv 1, \ \vdash\ p^{\#}\ 2\ (f^{\#}\ 1\ 2) \equiv 1,\\ (2)\quad & f^{\#}\ 1\ 2 \equiv 0 \qquad\quad p^{\#}\ 1\ (f^{\#}\ 2\ 2) \equiv 0\end{aligned}$$

The validity of applying an arbitrary hypothesis in a conclusion is derived from the following law of predicate logic.

$$p_1, \ldots, p_n \vdash q_1, \ldots, q_m \iff (p_1, \ldots, p_n \vdash q_1), \ldots, (p_n, \ldots, p_n \vdash q_m)$$

This says that a formula is valid if each conclusion is either valid or follows from at least one of the hypotheses. Notice that at each stage in the proof, it is possible to identify a value of i for each clause, as defined above, and therefore re-construct the proof completely formally. The next thing to be done is to add a proof clause for the element $f^{\#}\ 2\ 2$.

$$\begin{array}{lllll} (1) & f^{\#}\ 2\ 1 \equiv 1, & & p^{\#}\ 2\ (f^{\#}\ 1\ 2) \equiv 1, \\ (2) & f^{\#}\ 1\ 2 \equiv 0, & \vdash & p^{\#}\ 1\ (f^{\#}\ 2\ 2) \equiv 0, \\ (3) & f^{\#}\ 2\ 2 \equiv 0 & & p^{\#}\ 2\ (f^{\#}\ 2\ 2) \equiv 0 \end{array}$$

Applying hypothesis (3) in conclusion (3) produces the result 1, which again leads to a contradiction. Taking this as a hint, clause (3) is re-adjusted as follows.

$$\begin{array}{lllll} (1) & f^{\#}\ 2\ 1 \equiv 1, & & p^{\#}\ 2\ (f^{\#}\ 1\ 2) \equiv 1, \\ (2) & f^{\#}\ 1\ 2 \equiv 0, & \vdash & p^{\#}\ 1\ (f^{\#}\ 2\ 2) \equiv 0, \\ (3) & f^{\#}\ 2\ 2 \equiv 1 & & p^{\#}\ 2\ (f^{\#}\ 2\ 2) \equiv 1 \end{array}$$

This also leads to a contradiction, so further re-adjustment is required. After two further iterations, clause (3) becomes valid.

$$\begin{array}{lllll} (1) & f^{\#}\ 2\ 1 \equiv 1, & & p^{\#}\ 2\ (f^{\#}\ 1\ 2) \equiv 1, \\ (2) & f^{\#}\ 1\ 2 \equiv 0, & \vdash & p^{\#}\ 1\ (f^{\#}\ 2\ 2) \equiv 0, \\ (3) & f^{\#}\ 2\ 2 \equiv 3 & & p^{\#}\ 2\ (f^{\#}\ 2\ 2) \equiv 3 \end{array}$$

Now, applying hypothesis (3) in conclusion (2) gives the result 3, which again leads to a contradiction. Taking this as a hint and re-adjusting (2), this clause now becomes valid.

$$\begin{array}{lllll} (1) & f^{\#}\ 2\ 1 \equiv 1, & & p^{\#}\ 2\ (f^{\#}\ 1\ 2) \equiv 1, \\ (2) & f^{\#}\ 1\ 2 \equiv 3, & \vdash & p^{\#}\ 1\ (f^{\#}\ 2\ 2) \equiv 3, \\ (3) & f^{\#}\ 2\ 2 \equiv 3 & & p^{\#}\ 2\ (f^{\#}\ 2\ 2) \equiv 3 \end{array}$$

All that remains now is to apply hypothesis (2) in conclusion (1), which, after re-adjustment, renders clause (1) valid.

$$\begin{array}{lllll} (1) & f^{\#}\ 2\ 1 \equiv 3, & & p^{\#}\ 2\ (f^{\#}\ 1\ 2) \equiv 3, \\ (2) & f^{\#}\ 1\ 2 \equiv 3, & \vdash & p^{\#}\ 1\ (f^{\#}\ 2\ 2) \equiv 3, \\ (3) & f^{\#}\ 2\ 2 \equiv 3 & & p^{\#}\ 2\ (f^{\#}\ 2\ 2) \equiv 3 \end{array}$$

Hence, the value of $fix\ F^{\#}\ 2\ 1$ is 3. Notice, that in the course of determining $fix\ F^{\#}\ 2\ 1$, two further elements, $fix\ F^{\#}\ 1\ 2$ and $fix\ F^{\#}\ 2\ 2$, were discovered. This merely indicates that $fix\ F^{\#}\ 2\ 1$ depends on $fix\ F^{\#}\ 1\ 2$ and $fix\ F^{\#}\ 2\ 2$. Additional elements such as these may be worth retaining for possible re-use in other proofs.

Fortunately, although it is quite usual for an element to depend on several others, it is rare for it to depend on many. Indeed, if this were the case, the inductive method would be no better than using frontiers. Incidentally, the above suggests that when several elements of a fixpoint are required, say as part of a program compilation, it may be more efficient to determine them collectively rather than individually. This would offer the maximum opportunity to re-use common elements discovered during proofs.

3.1 Nested Applications and Mutual Recursion

In this section, an example is presented which shows how the above method extends naturally to definitions involving nested function applications and mutual recursion. It uses the following generalization of computational induction for dealing with multiple definitions.

Theorem 2: *For monotonic and continuous functionals* $F_1, \ldots, F_m$ *and an admissible predicate* P, $P\ (fix\ F_1) \cdots (fix\ F_m)$ *holds if*

- $P\ \Omega \cdots \Omega$ *holds, and*

forall n, $P\ (F_1^n) \cdots (F_m^n) \vdash P\ (F_1^{n+1}) \cdots (F_m^{n+1})$.

Example 3

Consider the following mutually recursive abstract functions, defined in terms of the operators $p^{\#}$ and $q^{\#}$, above.

$$\begin{array}{l} f^{\#}\ x \equiv p^{\#}\ x\ (f^{\#}\ (g^{\#}\ (q^{\#}\ x))) \\ g^{\#}\ x \equiv f^{\#}\ (q^{\#}\ x) \end{array}$$

Suppose it is required to determine the value of $f^{\#}\ 0$. Since this is admissible, it is valid to use computational induction. The first step asserts that $f^{\#}\ 0$ is 0.

$$(1) \quad f^{\#}\ 0 \equiv 0 \ \vdash \ p^{\#}\ 0\ (f^{\#}\ (g^{\#}\ 1)) \equiv 0$$

Theorem 2 enables properties of the fixpoint of $g^{\#}$ to be proved simultaneously with properties of the fixpoint of $f^{\#}$. This means that a clause for $g^{\#}\ 1$ may now be added to the proof.

$$\begin{array}{lllll} (1) & f^{\#}\ 0 \equiv 0, & & p^{\#}\ 0\ (f^{\#}\ (g^{\#}\ 1)) \equiv 0, \\ (2) & g^{\#}\ 1 \equiv 0 & \vdash & f^{\#}\ 2 \equiv 0 \end{array}$$

The next step is to determine $f^{\#}\ 2$.

$$\begin{array}{lllll} (1) & f^{\#}\ 0 \equiv 0, & & p^{\#}\ 0\ (f^{\#}\ (g^{\#}\ 1)) \equiv 0, \\ (2) & g^{\#}\ 1 \equiv 0, & \vdash & f^{\#}\ 2 \equiv 0, \\ (3) & f^{\#}\ 2 \equiv 0 & & p^{\#}\ 2\ (f^{\#}\ (g^{\#}\ 2)) \equiv 0 \end{array}$$

Now $g^{\#}\ 2$ is required.

$$\begin{array}{lllll} (1) & f^{\#}\ 0 \equiv 0, & & p^{\#}\ 0\ (f^{\#}\ (g^{\#}\ 1)) \equiv 0, \\ (2) & g^{\#}\ 1 \equiv 0, & \vdash & f^{\#}\ 2 \equiv 0, \\ (3) & f^{\#}\ 2 \equiv 0, & & p^{\#}\ 2\ (f^{\#}\ (g^{\#}\ 2)) \equiv 0, \\ (4) & g^{\#}\ 2 \equiv 0 & & f^{\#}\ 2 \equiv 0 \end{array}$$

Applying hypotheses (4) and (1), in that order, in conclusion (3) produces the result 1, which leads to a contradiction. Taking this as a hint, clause (3) is re-adjusted as follows.

$$\begin{array}{lllll} (1) & f^{\#}\ 0 \equiv 0, & & p^{\#}\ 0\ (f^{\#}\ (g^{\#}\ 1)) \equiv 0, \\ (2) & g^{\#}\ 1 \equiv 0, & \vdash & f^{\#}\ 2 \equiv 0, \\ (3) & f^{\#}\ 2 \equiv 1, & & p^{\#}\ 2\ (f^{\#}\ (g^{\#}\ 2)) \equiv 1, \\ (4) & g^{\#}\ 2 \equiv 0 & & f^{\#}\ 2 \equiv 0 \end{array}$$

This leads to contradictions in both clauses (2) and (4). Readjusting these clauses accordingly, gives

$$\begin{array}{lllll} (1) & f^{\#}\ 0 \equiv 0, & & p^{\#}\ 0\ (f^{\#}\ (g^{\#}\ 1)) \equiv 0, \\ (2) & g^{\#}\ 1 \equiv 1, & \vdash & f^{\#}\ 2 \equiv 1, \\ (3) & f^{\#}\ 2 \equiv 1, & & p^{\#}\ 2\ (f^{\#}\ (g^{\#}\ 2)) \equiv 1, \\ (4) & g^{\#}\ 2 \equiv 1 & & f^{\#}\ 2 \equiv 1 \end{array}$$

Applying hypothesis (4) in conclusion (3) creates the additional requirement of finding the value of $f^{\#}\ 1$. Including this as a further clause in the proof produces the following.

$$\begin{array}{lllll} (1) & f^{\#}\ 0 \equiv 0, & & p^{\#}\ 0\ (f^{\#}\ (g^{\#}\ 1)) \equiv 0, \\ (2) & g^{\#}\ 1 \equiv 1, & & f^{\#}\ 2 \equiv 1, \\ (3) & f^{\#}\ 2 \equiv 1, & \vdash & p^{\#}\ 2\ (f^{\#}\ (g^{\#}\ 2)) \equiv 1, \\ (4) & g^{\#}\ 2 \equiv 1, & & f^{\#}\ 2 \equiv 1, \\ (5) & f^{\#}\ 1 \equiv 0 & & p^{\#}\ 1\ (f^{\#}\ (g^{\#}\ 2)) \equiv 0 \end{array}$$

At this point all of the conclusions follow from the hypotheses, therefore, the fix-value of $f^{\#}\ 0$ is 0.

3.2 Higher-order Functions

This section briefly considers how the above may be applied to higher-order functions. The theory of abstract interpretation for higher-order functions was developed by Burn *et al*[3]. Abramsky[1] subsequently adapted this to allow for polymorphically typed functions. Clearly, there is no fundamental problem for applying the inductive method to functions which take functions as arguments, indeed, function spaces behave just like any other abstract domain. There is, of course, the matter of finding a suitable representation for the elements of function spaces but this has to be resolved no matter which method is used to determine the fixpoint.

The situation for functions which return functions as results is slightly more complicated. In particular, it is not at all clear what should be done with partial function applications. One possible solution is to saturate partial applications with "top" elements from the appropriate domains. This is a safe way to avoid the situation where hypotheses and conclusions in a proof involve the same abstract function but with differing numbers of arguments. Although this is a safe solution to the problem, it is not entirely satisfactory. It is possible that useful information may be lost in the process of approximating the "unknown" arguments.

Having now considered the essential elements of a functional system, it is time to reflect on the overall correctness of the inductive method.

3.3 Correctness

A procedure for finding fixpoints is *correct* if it is both *sound* and *finitely terminating*. For the purposes of establishing correctness, it is assumed that all abstract functions have finite arguments and are defined over finite domains and ranges. Clearly, the inductive method is sound because it corresponds to a well-founded and admissible inductive proof. Moreover, it is finitely terminating since there can be at most a finite number of possible combinations of finite arguments taken from finite domains, which therefore puts a ceiling on the number of proof clauses that might need validating. Consequently, it is concluded that this approach is correct. Notice that it is the finiteness of this method which enables it to be completely automated.

4 A Practical Example

This section presents a more practical example of an analysis, due to Wadler[15], for discovering if a function is strict in the *heads* and the *tails* of a list argument. It uses the two-point domain $\bot \sqsubseteq \top$, introduced earlier, plus the four-point domain $\bot \sqsubseteq \infty \sqsubseteq \bot\epsilon \sqsubseteq \top\epsilon$, where $\bot$ denotes undefined lists, ∞ denotes infinite lists or approximations thereof, [2] $\bot\epsilon$ denotes finite lists containing some undefined elements and $\top\epsilon$ denotes defined lists. The abstract operators used in the analysis are $\sqcup$, and $cons^\#$ which is the abstract form of the list constructor ":" and is defined as follows.

$cons^\#$	$\bot$	$\top$
$\bot$	∞	∞
∞	∞	∞
$\bot\epsilon$	$\bot\epsilon$	$\bot\epsilon$
$\top\epsilon$	$\bot\epsilon$	$\top\epsilon$

[2] An approximation to an infinite list is a list ending in $\bot$

The empty list [], being totally defined, has the abstract value $\top\epsilon$. In his paper, Wadler considers an analysis for a class of functions described by the following program scheme.

$$\mathcal{F}\ []\ \overline{ys} = \mathcal{G}\ \overline{ys}$$
$$\mathcal{F}\ (x : xs)\ \overline{ys} = \mathcal{H}\ x\ xs\ \overline{ys}$$

(Here the $\overline{ys}$ are assumed to be simple variables.) This covers, for example, the reverse and concatenation functions for lists, below.

$$rev\ [] = []$$
$$rev\ (x : xs) = app\ (rev\ xs)\ ([x])$$

$$app\ []\ ys = ys$$
$$app\ (x : xs)\ ys = x : (app\ xs\ ys)$$

(Here, $[x]$ is simply a shorthand for $x : []$.) Omitting the actual details, the above scheme is abstracted as follows.

$$\mathcal{F}^\#\ \top\epsilon\ \overline{ys} = \mathcal{G}^\#\ \overline{ys} \sqcup (\mathcal{H}^\#\ \top\ \top\epsilon\ \overline{ys})$$
$$\mathcal{F}^\#\ \bot\epsilon\ \overline{ys} = (\mathcal{H}^\#\ \bot\ \top\epsilon\ \overline{ys}) \sqcup (\mathcal{H}^\#\ \top\ \bot\epsilon\ \overline{ys})$$
$$\mathcal{F}^\#\ \infty\ \overline{ys} = \mathcal{H}^\#\ \top\ \infty\ \overline{ys}$$
$$\mathcal{F}^\#\ \bot\ \overline{ys} = \bot$$

From this it is possible to obtain abstract forms for the functions *app* and *rev* as follows.

$$rev^\#\ \top\epsilon = \top\epsilon$$
$$rev^\#\ \bot\epsilon = (app^\#\ \top\epsilon\ \bot\epsilon) \sqcup (app^\#\ (rev\ \bot\epsilon)\ \bot\epsilon)$$
$$rev^\#\ \infty = app^\#\ (rev^\#\ \infty)\ \infty$$
$$rev^\#\ \bot = \bot$$

$$app^\#\ \top\epsilon\ ys = ys \sqcup (cons^\#\ \top\ (app^\#\ \top\epsilon\ ys))$$
$$app^\#\ \bot\epsilon\ ys = (cons^\#\ \bot\ (app^\#\ \top\epsilon\ ys)) \sqcup (cons^\#\ \top\ (app^\#\ \bot\epsilon\ ys))$$
$$app^\#\ \infty\ ys = cons^\#\ \top\ (app^\#\ \infty\ ys)$$
$$app^\#\ \bot\ ys = \bot$$

Notice that properties of $\sqcup$ and $cons^\#$ have been used to simplify $rev^\#$ and $app^\#$. A function f is said to be *head strict* if it gives an undefined result when applied to a list containing some undefined elements. This may be verified, using Wadler's analysis, by showing that $f^\#\ \bot\epsilon \equiv \bot$. Similarly, f is said to be *tail strict* if its result is undefined whenever its argument is an infinite list or an approximation to one. This is verified by showing that $f^\#\ \infty \equiv \bot$. Both strictness tests generalise to n-ary functions in the usual way.

Suppose it is required to discover if the function rev, above, is head strict and tail strict. What has to be found are the values of $rev^\#\ \bot\epsilon$ and $rev^\#\ \infty$. Since these correspond to admissible predicates, it is valid to use computational induction. As usual, the proof begins by asserting these to be $\bot$.

(1) $rev^\#\ \bot\epsilon \equiv \bot$,
(2) $rev^\#\ \infty \equiv \bot$
$\vdash$ $(app^\#\ \top\epsilon\ \bot\epsilon) \sqcup (app^\#\ (rev\ \bot\epsilon)\ \bot\epsilon) \equiv \bot$,
$app^\#\ (rev^\#\ \infty)\ \infty \equiv \bot$

Applying hypothesis (2) in conclusion (2) leads to the requirement for the value of $app^\#\ \bot\ \infty$. However, since it is already known that $app^\#\ \bot\ ys$ is $\bot$, it follows that clause (2) is valid. Turning now to clause (1), applying hypothesis (1) produces requirements for the values of $app^\#\ \top\epsilon\ \bot\epsilon$ and $app^\#\ \bot\ \bot\epsilon$. Since, what is required is the "greater" of the two, $app^\#\ \top\epsilon\ \bot\epsilon$ will be considered first.

(1) $rev^\#\ \bot\epsilon \equiv \bot$,
(2) $rev^\#\ \infty \equiv \bot$,
(3) $app^\#\ \top\epsilon\ \bot\epsilon \equiv \bot$
$\vdash$ $(app^\#\ \top\epsilon\ \bot\epsilon) \sqcup (app^\#\ (rev\ \bot\epsilon)\ \bot\epsilon) \equiv \bot$,
$app^\#\ (rev^\#\ \infty)\ \infty \equiv \bot$,
$\bot\epsilon \sqcup (cons^\#\ \top\ (app^\#\ \top\epsilon\ \bot\epsilon)) \equiv \bot$

Applying hypothesis (3) in conclusion (3) leads to a contradiction. Re-adjusting clause (3), in the usual manner, produces the following.

$$\begin{array}{lll} (1) & rev^{\#}\ \bot\epsilon \equiv \bot, & (app^{\#}\ \top\epsilon\ \bot\epsilon) \sqcup (app^{\#}\ (rev\ \bot\epsilon)\ \bot\epsilon) \equiv \bot, \\ (2) & rev^{\#}\ \infty \equiv \bot, & \vdash\ app^{\#}\ (rev^{\#}\ \infty)\ \infty \equiv \bot, \\ (3) & app^{\#}\ \top\epsilon\ \bot\epsilon \equiv \bot\epsilon & \bot\epsilon \sqcup (cons^{\#}\ \top\ (app^{\#}\ \top\epsilon\ \bot\epsilon)) \equiv \bot\epsilon \end{array}$$

Now clause (3) is valid. Applying hypothesis (3) in conclusion (1) leads to a contradiction. Since, by monotonicity,

$$app^{\#}\ \bot\ \bot\epsilon \sqsubseteq app^{\#}\ \top\epsilon\ \bot\epsilon$$

it is correct to take the value of $app^{\#}\ \top\epsilon\ \bot\epsilon$, as the next approximation.

$$\begin{array}{lll} (1) & rev^{\#}\ \bot\epsilon \equiv \bot\epsilon, & (app^{\#}\ \top\epsilon\ \bot\epsilon) \sqcup (app^{\#}\ (rev\ \bot\epsilon)\ \bot\epsilon) \equiv \bot\epsilon, \\ (2) & rev^{\#}\ \infty \equiv \bot, & \vdash\ app^{\#}\ (rev^{\#}\ \infty)\ \infty \equiv \bot, \\ (3) & app^{\#}\ \top\epsilon\ \bot\epsilon \equiv \bot\epsilon & \bot\epsilon \sqcup (cons^{\#}\ \top\ (app^{\#}\ \top\epsilon\ \bot\epsilon)) \equiv \bot\epsilon \end{array}$$

It is now necessary to discover the value of $app^{\#}\ \bot\epsilon\ \bot\epsilon$. However, since this too is "smaller" than $app^{\#}\ \top\epsilon\ \bot\epsilon$, no further work is required. All the conclusions now follow as logical consequences of the hypotheses, hence the fix-values of $rev^{\#}\ \bot\epsilon$ and $rev^{\#}\ \bot\epsilon$ are $\bot\epsilon$ and $\bot$, respectively. The results are as one would expect, *rev* is tail strict but not head strict.

5 Conclusions and Further Work

Computational induction provides an effective mechanical method for finding fixpoints in abstract interpretation. Its correctness is assured for abstract functions of finite arguments defined over finite domains and ranges. The advantages of this method over existing techniques are two-fold. Firstly, unlike the frontiers method, it allows elements of the fixpoint to be determined separately which saves computing many elements that never get used in an analysis, and, secondly, unlike pending analysis, it is not restricted to a two-point domain. It is expected that in general the inductive approach will give better performance than either of the other methods.

A strictness detection scheme based on the proposed method is being incorporated into an interactive transformation system, called STARSHIP[6], which is currently under development. Here, strictness analysis is applied as a check prior to an instantiation step in *fold/unfold* to ensure that the full semantics of the program is preserved (see [14] for details).

There are several areas which need more work. First and foremost, it will be necessary to develop strategies for selecting clauses to validate, which minimise the number of steps needed to carry out proofs. Also, there are various technical details still to be sorted out regarding the use of higher order functions, not least of which is the matter of how to best represent partial function applications. Finally, it might be interesting to look at conservative extensions of the framework to allow infinite domains.

References

[1] S. Abramsky, "Strictness analysis and polymorphic invariance," in H. Ganzinger and N. D. Jones, editors, Proceedings of the Workshop on Programs as Data Objects, Spinger-Verlag, LNCS 217, Copenhagen, October 1985.

[2] S. Abramsky and C. Hankin, editors, Abstract Interpretation of Declarative Languages, Ellis Horwood, 1987.

[3] G. L. Burn, C. L. Hankin and S. Abramsky, "The Theory of Strictness Analysis for Higher Order Functions," in H. Ganzinger and N. D. Jones, editors, Proceedings of the Workshop on Programs as Data Objects, Spinger-Verlag, LNCS 217, Copenhagen, October 1985.

[4] C. Clack and S. L. Peyton Jones, "Strictness analysis—a practical approach," in J-P. Jouannaud, editor, Proceedings of the Conference on Functional Programming Languages and Computer Architecture, Springer-Verlag, LNCS 201, Nancy, September 1985.

[5] A. J. Dix, "Finding Fixed Points in Non-trivial Domains: Proofs of Pending Analysis and Related Algorithms," Technical Report YCS 107, University of York, Department of Computer Science, 1988.

[6] M. Firth and C.Runciman, "STARSHIP Version 0.1 Reference Manual," July, 1989.

[7] Z. Manna, Mathematical Theory of Computation, McGraw-Hill, 1974.

[8] C. C. Martin, "Finding Fixpoints Using a Syntactic Method," Imperial College, Department of Computing, 1987.

[9] C. C. Martin, "Proof of Pending Analysis," Imperial College, Department of Computing, 1989.

[10] C. C. Martin and C. Hankin, "Finding Fixed Points in Finite Lattices," in G. Kahn, editor, Proceedings of the Conference on Functional Programming Languages and Computer Architecture, Spinger-Verlag, LNCS 274, Portland, Orgegon, September 1987.

[11] A. Mycroft, "Abstract Interpretation and Optimising Transformations for Applicative Programs," PhD thesis, University of Edinburgh, Department of Computer Science, December 1981.

[12] S. L. Peyton Jones, The Implementation of Functional Programming Languages, Prentice-Hall, 1987.

[13] S. L. Peyton Jones and C. Clack, "Finding fixpoints in abstract interpretation," Technical report, University College, Department of Computer Science, July 1986.

[14] C. Runciman, M. Firth, and N. Jagger, "Preserving interactive behaviour through transformation," Paper 617 (CHM-24), IFIP Working Group, Grenoble, January 1989.

[15] P. Wadler, "Strictness analysis on non-flat domains (by abstract interpretation over finite domains)," in [2], 1987.

[16] J. H. Young, "The Theory and Practice of Semantic Program Analysis for Higher-Order Functional Programming Languages," PhD thesis, Yale University, Department of Computer Science, 1989.