

Higher-Order Chaotic Iteration Sequences

Mads Rosendahl

DIKU, University of Copenhagen

Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark

E-mail `rose@diku.dk`

1993

Abstract

Chaotic iteration sequences is a method for approximating fixpoints of monotonic functions proposed by Patrick and Radhia Cousot. It may be used in specialisation algorithms for Prolog programs and in abstract interpretation when only parts of a fixpoint result is needed to perform program optimisations. In the first part of this paper we reexamine the definition of chaotic iteration sequences and show how a number of other methods to compute fixpoints may be formulated in this framework. In the second part we extend the technique to higher-order functions.

The method of expressing solutions to problems in a recursive style has a long history in Computer Science. Often a solution algorithm may use itself on a subproblem and an implementation may be based on recursion. At times it is more natural to express the solution as a fixpoint equation in which well-definedness is only guaranteed using a fixpoint theorem. This is common in abstract interpretation, data flow analysis, “first” and “follow” computation, etc.

In strictness analysis [11] we may construct functions over the domain $\mathcal{2} = \{0, 1\}$ with $0 \leq 1$. If we want to evaluate the function

$$f^\sharp(x, y, z) = (y \wedge z) \vee f^\sharp(z, x, f^\sharp(y, 1, 1))$$

for certain arguments, we will in general need to evaluate the function for all possible arguments and recompute until stability. It is, however, often only necessary to recompute the function for some of the possible arguments.

By restricting the iteration to a smaller set of arguments, we may greatly simplify the computational task of evaluating the function. In this paper we consider such recursively defined functions for which recursion is not sufficient in the implementation.

1 Language

As a start we will consider a simple language consisting of one recursively defined function over a complete partially ordered set D . Programs p in this language will have the form

$$p \quad : \quad \mathbf{letfix} \ f(x_1, \dots, x_k) = e_f \ \mathbf{in} \ e$$

where the expressions e_f and e are built from parameters, constants, basic operations, and function calls.

$$\begin{aligned} e : & x_i \\ & | c_i \\ & | op_i(e_1, \dots, e_k) \\ & | f(e_1, \dots, e_k) \end{aligned}$$

We may specify the semantics of this language using the functions **M** and **E**.

$$\begin{aligned} \mathbf{M}[\mathbf{letfix} \ f(x_1, \dots, x_k) = e_f \ \mathbf{in} \ e] & : D \\ \mathbf{E}[e] & : (D^k \rightarrow D) \rightarrow D^k \rightarrow D \end{aligned}$$

with

$$\begin{aligned} \mathbf{M}[\mathbf{letfix} \ f(x_1, \dots, x_k) = e_f \ \mathbf{in} \ e] & = \mathbf{E}[e](\text{fix } \mathbf{E}[e_f]) \perp_{D^k} \\ \mathbf{E}[x_i] \phi \rho & = sel_i(\rho) \\ \mathbf{E}[c_i] \phi \rho & = c_i \\ \mathbf{E}[op_i(e_1, \dots, e_k)] \phi \rho & = \underline{op}_i(\mathbf{E}[e_1] \phi \rho, \dots, \mathbf{E}[e_k] \phi \rho) \\ \mathbf{E}[f(e_1, \dots, e_k)] \phi \rho & = \phi(\mathbf{E}[e_1] \phi \rho, \dots, \mathbf{E}[e_k] \phi \rho) \end{aligned}$$

for constants $c_i \in D$ and standard operations $\underline{op}_i \in D^k \rightarrow D$. The function sel_i selects the i^{th} element in the tuple given as argument and “fix” finds the least fixpoint of its argument. To guarantee the well-definedness of this definition we must require that the function $\mathbf{E}[e_f]$ is continuous. This can be

done by only using continuous standard operations op_i . It is, however, not necessary for all subexpressions of e_f to be continuous. Only the argument to “fix” needs to be continuous. In some applications (eg. data flow analysis) it is often natural locally to use non-monotonic operations while the fixpoint is found for a continuous function.

2 Fixpoint iteration

Consider the expression

$$\mathbf{letfix} \ f(x_1, \dots, x_k) = e_f \ \mathbf{in} \ f(v_1, \dots, v_k)$$

with constants $v_i \in D$ and expression e_f . The value of this expression is defined as

$$\mathbf{fix}(\mathbf{E}[e_f]) \langle v_1, \dots, v_k \rangle = (\bigsqcup_i (\mathbf{E}[e_f])^i \lambda \rho. \perp_D) \langle v_1, \dots, v_k \rangle$$

Our aim is to find an approximation to $\mathbf{fix}(\mathbf{E}[e_f])$ which has the correct value for $\langle v_1, \dots, v_k \rangle$.

2.1 Argument needs

If we want to evaluate the expression $\mathbf{letfix} \ f(x_1, \dots, x_k) = e_f \ \mathbf{in} \ f(v_1, \dots, v_k)$ we do not necessarily need to compute the fixpoint of f for all arguments in D^k . Only arguments which may be reached from the call $f(v_1, \dots, v_k)$ can influence the result. We may formalise the notion of reachability as follows.

We define the function $\mathbf{A}[e]$ to return the set of argument tuples in calls to the function f that will occur when evaluating the expression e

$$\mathbf{A}[e] : (D^k \rightarrow D) \rightarrow D^k \rightarrow \mathcal{P}(D^k)$$

$$\mathbf{A}[x_i] \phi \rho = \emptyset$$

$$\mathbf{A}[c_i] \phi \rho = \emptyset$$

$$\mathbf{A}[op_i(e_1, \dots, e_k)] \phi \rho = \mathbf{A}[e_1] \phi \rho \cup \dots \cup \mathbf{A}[e_k] \phi \rho$$

$$\mathbf{A}[f(e_1, \dots, e_k)] \phi \rho = \{ \langle \mathbf{E}[e_1] \phi \rho, \dots, \mathbf{E}[e_k] \phi \rho \rangle \} \cup \mathbf{A}[e_1] \phi \rho \cup \dots \cup \mathbf{A}[e_k] \phi \rho$$

It is worth noting that the function $\mathbf{A}[e]$ is not necessarily continuous in any of its arguments. This is because we have used a power set construction $\mathcal{P}(D^k)$ rather than a power domain and thereby forgetting the structure of

D . Continuity is, however, only important when we compute a fixpoint and in this case we may achieve continuity in a different way. Instead we define the function \mathcal{A} as follows

$$\mathcal{A}[[e]]\phi S_0 = \text{fix}(\lambda S.S_0 \cup \bigcup_{\rho \in S} \mathbf{A}[[e]]\phi\rho)$$

In the expression

$$\mathbf{letfix} f(x_1, \dots, x_k) = e_f \mathbf{in} f(v_1, \dots, v_k)$$

the immediate argument need is $\rho^0 = \langle v_1, \dots, v_k \rangle$. The indirect needs are $\mathcal{A}[[e_f]]\phi\{\rho^0\}$. This definition depends on the function environment ϕ . The next step will be to show how argument needs may be used to simplify the fixpoint iteration.

2.2 Iteration sequence

For the expression

$$\mathbf{letfix} f(x_1, \dots, x_k) = e_f \mathbf{in} f(v_1, \dots, v_k)$$

define the function

$$F_V(\phi) = \lambda\rho. \mathbf{if} \rho \in V \mathbf{then} \mathbf{E}[[e_f]]\phi\rho \mathbf{else} \phi(\rho)$$

which computes a better approximation to ϕ for arguments $\rho \in V \subseteq D^k$. An *iteration sequence* is a sequence of function denotations ϕ^0, ϕ^1, \dots and sets $V^i \subseteq D$ such that

$$\begin{aligned} \phi^0 &= \lambda\rho. \perp_D \\ \phi^{i+1} &= F_{V^i}(\phi^i) \end{aligned}$$

An iteration sequence is then completely determined by the sets V^i . The natural aim of fixpoint iteration is to keep these sets as small as possible while securing that the sequence of function denotations stabilise quickly with the correct value for $f(v_1, \dots, v_k)$.

2.3 Simple fixpoint iteration

A simple approach to fixpoint iteration is to define the sets V^i in an iteration sequence as follows:

$$\begin{aligned} V^0 &= \{\langle v_1, \dots, v_k \rangle\} \\ V^{i+1} &= V^i \cup \bigcup_{\rho \in V^i} \mathbf{A}[[e_f]]\phi^{i+1}\rho \end{aligned}$$

The claim now is that if the sequence of sets (V^i) with function denotations (ϕ^i) stabilise after s iterations then

$$\phi^s \langle v_1, \dots, v_k \rangle = \text{fix } \mathbf{E} \llbracket e_f \rrbracket \langle v_1, \dots, v_k \rangle$$

The correctness of this claim is a special case of the correctness of chaotic fixpoint iteration discussed later.

2.3.1 Termination.

We may notice that (ϕ^i) and (V^i) form an increasing sequences of values.

$$\begin{aligned} \phi^i &\sqsubseteq \phi^{i+1} \sqsubseteq \text{fix } \mathbf{E} \llbracket e_f \rrbracket \\ V^i &\subseteq V^{i+1} \subseteq D^k \end{aligned}$$

This means that we know they will stabilise if D is finite. Termination may, however, also be achieved when D is infinite or has infinite height if restrictions are placed on the expression E . The use of widening [2] in abstract interpretation is an example of this. See also [13] and [6] for a discussion of methods to guarantee termination.

2.4 Spurious calls

With a stabilised iteration sequence we may note that the set

$$V^s - \mathcal{A} \llbracket e_f \rrbracket \phi^s V^0$$

is not necessarily empty. Arguments in this set are needed in the approximations to the fixpoint but are no longer needed when the sequence is stable. Gallagher and Bruynooghe [5] call such calls “spurious” and propose a method to remove some but not all them. The result of the simple iteration sequence with spurious calls is called the *partial function graph* in [5] and [16].

2.4.1 Example.

Consider the function F defined as a fixpoint in the domain of functions over $\mathcal{P}(\{\text{"a"}, \text{"b"}, \text{"c"}\})$ ordered with subset inclusion.

$$\text{letfix } F(D) = F(F(F(D \cup \{\text{"a"}\}) \cup \{\text{"b"}\}) \cup \{\text{"c"}\}) \cup D \text{ in } F(\{\text{"a"}\})$$

To compute $F(\{\text{"a"}\})$ we only need to evaluate $F(\{\text{"a"}\})$ and $F(\{\text{"a"}, \text{"b"}, \text{"c"}\})$. In the iteration we will, however, also need to evaluate the function for some subsets of $\{\text{"a"}, \text{"b"}, \text{"c"}\}$. The simple iteration method will find all non-empty subsets of $\{\text{"a"}, \text{"b"}, \text{"c"}\}$ as needed.

2.4.2 Example.

Although spurious calls do not appear for strict functions on flat domains (as in minimal function graph semantics [9]) they may appear for the two-point strictness domain. Consider the function

$$f^\sharp(x, y, z) = f^\sharp(0, z, f^\sharp(x, z, y)) \vee (y \wedge z)$$

and the call $f^\sharp(0, 1, 1)$. In the fixpoint only this call is needed although in the iteration also $f^\sharp(0, 1, 0)$ will be evaluated.

2.4.3 Alternative.

The iteration method proposed by Gallagher and Bruynooghe [5] uses the following sequence:

$$\begin{aligned} \phi^{i+1} &= F_{\mathbf{LC}(V^i)}(\phi^i) \\ V^{i+1} &= V^0 \cup \bigcup_{\rho \in V^i} \mathbf{A}[[e_f]]\phi^{i+1}\rho \end{aligned}$$

where $\mathbf{LC}(V^i)$ is the lower closure of V^i . The use of the lower closure corresponds to using a Hoare power domain rather than a power set in the definition of \mathbf{A} . This strategy may be applied when D is finite but it may be impractical if the domain has finite height but infinite depth. The limit of the sequence $(\mathbf{LC}(V^i))$ will in general be larger than $\mathcal{A}[[e_f]](\text{fix } \mathbf{E}[[e_f]])V^0$.

3 Chaotic fixpoint iteration

Our aim is to find an approximation to $\text{fix}(\mathbf{E}[[e_f]])$ which has the correct value for $\rho^0 = \langle v_1, \dots, v_k \rangle$. We may observe that an approximation only needs to have the correct values for arguments in $\mathcal{A}[[e_f]]\phi\{\rho^0\}$ to be a fixpoint. If we take the result of the simple iteration method with stable values ϕ^s and V^s then $W = \mathcal{A}[[e_f]]\phi^s\{\rho^0\}$ will make

$$\lambda\rho. \text{ if } \rho \in W \text{ then } \phi^s(\rho) \text{ else } \perp_D$$

the least solution to the equation $\phi = F_W(\phi)$. This observation is central in the definition of chaotic iteration sequences.

We will here use a modified definition of chaotic iteration sequences compared to the original definition by Cousot [3]. The main difference is that forward dependencies in the sequence are removed so as to make it directly applicable in an algorithm.

3.1 Chaotic iteration sequences

A chaotic iteration sequence¹ is a sequence of sets (V^i) and function denotations (ϕ^i) which satisfy:

$$\begin{aligned}\phi^{i+1} &= F_{V^i}(\phi^i) \\ \exists m. \forall i. \exists \ell \in [1, m]. \rho^0 &\in V^{i+\ell} \wedge \mathcal{A}[[e_f]]\phi^{i+\ell}\{\rho^0\} \subseteq \bigcup_{j=1}^m V^{i+\ell+j}\end{aligned}$$

This is a condition on an iteration sequence and unlike the simple iteration strategy, there may be many ways to construct such a sequence. With ρ^0 as the initial call, the intuition is that ρ^0 must appear repeatedly in the sequence of V s and that any direct or indirect needs from ρ^0 also must appear regularly in the sequence.

Again the claim is that if the sequence of (V^i) and (ϕ^i) stabilises then we have a correct value for ρ^0 . For details consult [3]; we will here examine some special cases of this fixpoint iteration method.

3.1.1 Example.

A simple algorithm arise if we require m to be 1. We may then use the following sequence.

$$\begin{aligned}\phi^{i+1} &= F_{V^i}(\phi^i) \\ V^{i+1} &= \mathcal{A}[[e_f]]\phi^{i+1}V^0\end{aligned}$$

Although $\mathcal{A}[[e_f]]$ is defined as a fixpoint, it may, being distributive, be computed using a simple depth-first strategy without iteration. If the sequence of (V^i) and (ϕ^i) stabilises after s iterations, we have

$$\begin{aligned}\phi^s &= F_{V^s}(\phi^s) \\ V^s &= \mathcal{A}[[e_f]]\phi^sV^0 = \mathcal{A}[[e_f]]\phi^sV^s\end{aligned}$$

3.2 Correctness

We will here consider the correctness of the special case of the iteration sequence

$$\begin{aligned}\phi^s &= F_{V^s}(\phi^s) \\ V^s &= \mathcal{A}[[e_f]]\phi^sV^0 = \mathcal{A}[[e_f]]\phi^sV^s\end{aligned}$$

The sequences is started with a set V^0 of initial calls.

¹The definition in [3] with our notation is $\exists m \forall \rho \in V^0 \forall i \geq 0 \exists \ell \in [1, m] \rho \in V^{i+\ell} \wedge \mathcal{A}[[e_f]]\phi^{i+\ell}\{\rho\} \subseteq \bigcup_{p=0}^{\ell+1} V^{i+p}$

3.2.1 Termination.

The sequence (V^i) will not in general form an increasing sequence. Nevertheless, if D is finite we know that the sequence will stabilise. The sequence (ϕ^i) form an increasing sequence and although $\mathcal{A}[[e_f]]$ is not monotonic in its second last argument it is monotonic in its last argument. This means that if (ϕ^i) stabilises then (V^i) will form an increasing sequence. Now, assume that $\phi^i = \phi^{i+1}$, we then have that

$$V^{i+1} = \mathcal{A}[[e_f]]\phi^i V^0 = \mathcal{A}[[e_f]]\phi^{i+1} V^0 = V^{i+2}$$

If $V^i = V^{i+1}$ then $V^i = \mathcal{A}[[e_f]]\phi^i V^0$ and for all $\rho \in V^i$ we have $\mathbf{E}[[e_f]]\phi^i \rho = \mathbf{E}[[e_f]]\phi^{i+1} \rho$ so that $\phi^{i+1} = \phi^{i+2}$. This shows that two consecutive identical V and ϕ values or three consecutive identical ϕ values guarantee stability. That two consecutive identical ϕ values does not guarantee stability is often referred to as the *plateau problem* of fixpoint iteration.

3.2.2 Theorem 1.

For the stabilised values V^s and ϕ^s of (V^i) and (ϕ^i) we have

$$\rho \in V^0 \Rightarrow \phi_i^s \rho = (\text{fix } \mathbf{E}[[e_f]])\rho$$

3.2.3 Proof.

See [3] or [4].

3.2.4 Theorem 2.

For function environment ϕ and set of calls V we have

$$\mathcal{A}[[e_f]]r_{\mathcal{A}[[e_f]]\phi V}\phi V = \mathcal{A}[[e_f]]\phi V$$

where $r_V \phi$ is the restriction of the function environment ϕ to the values in V .

$$r_V \phi = \langle \lambda \rho. \text{if } \langle 1, \rho \rangle \in V \text{ then } \phi(\rho) \text{ else } \perp_B, \dots \rangle$$

3.2.5 Proof.

The function \mathbf{A} computes the immediate needs in an expression. This may also be stated as

$$\mathbf{E}[[e]]r_{\mathbf{A}[[e]]}\phi\rho = \mathbf{E}[[e]]\phi\rho$$

This may be proved by structural induction over expressions. The theorem then follows by induction over the recursion depth.

3.2.6 Theorem 3.

For the stabilised values V^s and ϕ^s of (V^i) and (ϕ^i) we have

$$r_{V^s}\phi^s = r_{V^s}\text{fix } \mathbf{E}[[e_f]]$$

3.2.7 Proof.

For the stable value ϕ^s we have $\phi^s = F_{V^s}(\phi^s)$. This implies that $r_{V^s}(\mathbf{E}[[e_f]]\phi^s) = r_{V^s}\phi^s$ and that $r_{V^s}(\sqcup_i \mathbf{E}[[e_f]]^i \phi^s) = r_{V^s}\phi^s$. Now $\sqcup_i \mathbf{E}[[e_f]]^i \phi^s \sqsupseteq \text{fix } \mathbf{E}[[e_f]]$ so that $r_{V^s}\phi^s \sqsupseteq r_{V^s}(\text{fix } \mathbf{E}[[e_f]])$. On the other hand, it follows easily that $\phi^s \sqsubseteq \text{fix } \mathbf{E}[[e_f]]$.

3.2.8 Theorem 4.

For the stabilised values V^s and ϕ^s of (V^i) and (ϕ^i) we have

$$\mathcal{A}[[e_f]]\phi^s V^0 = \mathcal{A}[[e_f]](\text{fix } \mathbf{E}[[e_f]])V^0$$

3.2.9 Proof.

For the stable value V^s we have that $V^s = \mathcal{A}[[e_f]]\phi^s V^0$. Using theorem 2 with $\phi = \mathbf{E}[[e_f]]$ and $V = V^s$ gives

$$\mathcal{A}[[e_f]]r_{V^s}(\text{fix } \mathbf{E}[[e_f]])V^0 = \mathcal{A}[[e_f]](\text{fix } \mathbf{E}[[e_f]])V^0$$

Theorem 3 implies that

$$\mathcal{A}[[e_f]](r_{V^s}\phi^s)V^0 = \mathcal{A}[[e_f]](\text{fix } \mathbf{E}[[e_f]])V^0$$

Using theorem 2 again with $\phi = \phi^s$ gives us

$$\mathcal{A}[[e_f]]r_{V^s}\phi^s V^0 = \mathcal{A}[[e_f]]\phi^s V^0$$

3.2.10 Comment.

It should be noted that for arbitrary chaotic iteration sequences [3] we only have that $r_{V^s}\phi^s \sqsubseteq r_{V^s}(\text{fix } \mathbf{E}[[e_f]])$ since such sequences may compute an upper bound to the needs. We are, however, guaranteed that $r_{V^0}\phi^s = r_{V^0}(\text{fix } \mathbf{E}[[e_f]])$.

3.2.11 Minimal needs.

Theorem 2 only tells us that the needs computed by $\mathcal{A}[[e_f]]$ are sufficient to compute the fixpoint. A definition of the set of necessary needs for a first-order eager functional language can be found in [12]. That definition is not directly applicable to this situation, and, furthermore, seems not to be useful for fixpoint iteration as the result of the iteration is needed to find the needs that where not necessary.

3.3 Fixpoint algorithm

We may construct a chaotic iteration sequence efficiently using the following algorithm.

```
given  $V^0$ 
 $\phi = \lambda\rho.\perp$ 
repeat
   $S = V^0$            {calls, which should be analysed}
   $V = \emptyset$       {calls, which has been analysed}
  for  $\rho \in S$ 
     $V = V + \{\rho\}$ 
     $S = S + \mathbf{A}[[e_f]]\phi\rho - V$ 
     $\phi = \phi[\rho \mapsto \mathbf{E}[[e_f]]\phi\rho]$ 
  end
until  $V$  and  $\phi$  stable
```

An interesting property of this iteration strategy is that it will compute the correct fixpoint from possibly wrong (or imprecise) intermediate results. We know, however, that we approximate the solution from below so if we find a fixpoint, it will be the least.

In the algorithm we may use that the evaluations $\mathbf{A}[[e_f]]\phi\rho$ and $\mathbf{E}[[e_f]]\phi\rho$ can be done at the same time. The algorithm may be further optimised by using a depth-first strategy when collecting needed calls.

3.3.1 Applications.

The obvious application of the algorithm is to compute parts of a fixpoint, as in

$$(\text{fix } \mathbf{E}[[e_f]])\langle v_1, \dots, v_k \rangle$$

It may, however, also be used to compute the indirectly needed calls

$$\mathcal{A}[[e_f]](\text{fix } \mathbf{E}[[e_f]])\{\langle v_1, \dots, v_k \rangle\}$$

This is eg. used in multiple specialisation of prolog programs [16, 5]. Further possible applications are outlined in [4]. The algorithm above may be used to compute the set of needed calls. The set V will, when the sequence stabilise be identical to this set.

4 Higher-order functions

An underlying assumption behind the previous algorithm is that we have an equality operator for the domain D . We must be able to check whether arguments to calls have already been evaluated and we must compare results with previously evaluated values when checking for stability. Although this assumption is reasonable for domains as $\mathbb{2}$ and $\{\text{"a"}, \text{"b"}, \text{"c"}\}$ it does not hold for higher-order functions. We will here show how fixpoint iteration using chaotic iteration sequences may be extended to higher-order functions.

4.0.2 Language.

Let us consider a small language based on recursion equation systems. The language allows higher-order functions. We disallow anonymous functions (lambda abstractions); they may be removed from a program using the technique of λ -lifting [8]. A program p consists of a number of function definitions:

```

letfix  $f_1$   $x_1 \cdots x_k = e_1$ 
and    $\vdots$             $\vdots$ 
and    $f_n$   $x_1 \cdots x_k = e_n$  in  $e_{n+1}$ 

```

An expression is build from parameters and function names by application and basic operations.

x_i	Parameters
f_i	Function names
$op_i(e_1, \dots, e_k)$	Basic operations
$e_1(e_2)$	Application

For notational simplicity we assume that all functions have the same number of arguments, mainly to avoid subscripted subscripts. An actual implementation should not make this assumption, nor should it assume that the functions are named f_1, \dots, f_n , parameters x_1, \dots, x_k , and basic operations op_1, op_2, \dots

Programs must be strongly monomorphically typed so that each function may be assigned a finite type of the form:

$$\tau : D \mid \tau \rightarrow \tau$$

where D denotes a given domain. We assume that functions respect the types and that the functions are continuous. As for the first-order case, we do not require D to be finite or all subexpressions to be continuous.

4.0.3 Semantics.

The semantic description of the language is based on closures. Instead of representing a partially applied function as a function we describe it as a closure of a function identification and a list of the provided arguments.

$$\begin{aligned} U &= D + (\{1, \dots, n\} \times U^*) \quad \rho \in U^k \\ \Phi &= U^k \rightarrow U \quad \phi \in \Phi^n \end{aligned}$$

Semantic functions.

$$\begin{aligned} \mathbf{E}_h \llbracket e \rrbracket &: \Phi^n \rightarrow U^k \rightarrow U \quad \text{Expression meanings} \\ \mathbf{M}_h \llbracket p \rrbracket &: U \quad \text{Program meanings} \end{aligned}$$

with definitions:

$$\begin{aligned}
\mathbf{E}_h[x_i]\phi\rho &= sel_i(\rho) \\
\mathbf{E}_h[f_i]\phi\rho &= [i, \epsilon] \\
\mathbf{E}_h[op_j(e_1, \dots, e_k)]\phi\rho &= op_j(\mathbf{E}_h[e_1]\phi\rho, \dots, \mathbf{E}_h[e_k]\phi\rho) \\
\mathbf{E}_h[e_1(e_2)]\phi\rho &= \mathbf{let} [i, v_1, \dots, v_\ell] = \mathbf{E}_h[e_1]\phi\rho \mathbf{and} d = \mathbf{E}_h[e_2]\phi\rho \mathbf{in} \\
&\quad \mathbf{if} \ell < k - 1 \mathbf{then} [i, v_1, \dots, v_\ell, d] \\
&\quad \mathbf{else} \phi_i(v_1, \dots, v_\ell, d_2)
\end{aligned}$$

$$\begin{aligned}
\mathbf{M}_h[\mathbf{letfix} f_1 x_1 \dots x_k = e_1 \mathbf{and} \dots \mathbf{and} f_n x_1 \dots x_k = e_n \mathbf{in} e_{n+1}] &= \\
&= \mathbf{E}_h[e_{n+1}](\mathbf{fix} \lambda\phi. (\mathbf{E}_h[e_1]\phi, \dots, \mathbf{E}_h[e_n]\phi)) \perp_{U^k}
\end{aligned}$$

4.1 Argument needs

The directly needed calls during the evaluation of an expression may be found as for the first-order case. We define a function

$$\mathbf{A}[e] : \Phi^n \rightarrow U^k \rightarrow \mathcal{P}(\{1, \dots, n\} \times U^k)$$

as

$$\begin{aligned}
\mathbf{A}_h[x_i]\phi\rho &= \emptyset \\
\mathbf{A}_h[f_i]\phi\rho &= \emptyset \\
\mathbf{A}_h[op_j(e_1, \dots, e_k)]\phi\rho &= \mathbf{A}_h[e_1]\phi\rho \cup \dots \cup \mathbf{A}_h[e_k]\phi\rho \\
\mathbf{A}_h[e_1(e_2)]\phi\rho &= \mathbf{let} [i, v_1, \dots, v_\ell] = \mathbf{E}_h[e_1]\phi\rho \mathbf{and} d = \mathbf{E}_h[e_2]\phi\rho \mathbf{in} \\
&\quad \mathbf{if} \ell < k - 1 \mathbf{then} \mathbf{A}_h[e_1]\phi\rho \cup \mathbf{A}_h[e_2]\phi\rho \\
&\quad \mathbf{else} \{[f_i, v_1, \dots, v_\ell, d]\} \cup \mathbf{A}_h[e_1]\phi\rho \cup \mathbf{A}_h[e_2]\phi\rho
\end{aligned}$$

As for the first-order case we may find the set of indirect needs from given calls V^0 in the following way.

$$\mathcal{A}_h[p]\phi V^0 = \mathbf{fix}(\lambda V. V^0 \cup \bigcup_{[j, \rho] \in V} \mathbf{A}_h[e_j]\phi\rho)$$

The function $\mathcal{A}_h[p]$ is continuous in its last argument but not necessarily in its second last argument. Notice that the ordering of U is not used in the fixpoint in \mathcal{A}_h (where we use a power set) whereas it is used in the fixpoint in \mathbf{M}_h .

4.2 Fixpoint algorithm

Consider the program

```
letfix  $f_1$   $x_1 \cdots x_k = e_1$   
and  $\vdots$   $\vdots$   
and  $f_n$   $x_1 \cdots x_k = e_n$  in  $f_m(v_1, \dots, v_k)$ 
```

with the initial call $f_m(v_1, \dots, v_k)$ where $v_i \in D$. The set of initial call descriptions is then

$$V^0 = \{[m, v_1, \dots, v_k]\}$$

The fixpoint algorithm for the first-order case may now directly be adapted to higher-order functions.

```
given  $V^0$   
 $\phi_i = \lambda\rho. \perp$  for  $i = 1, \dots, n$   
repeat  
   $S = V^0$   
   $V = \emptyset$   
  for  $[j, \rho] \in S$   
     $V = V + \{[j, \rho]\}$   
     $S = S + \mathbf{A}_h[[e_j]]\phi\rho - V$   
     $\phi_j = \phi_j[\rho \mapsto \mathbf{E}_h[[e_j]]\phi\rho]$   
  end  
until  $V$  and  $\phi$  stable
```

4.3 Approximating fixpoints

If fixpoint iteration is used in program analysis we typically need to know the result of the iteration for a set of initial needs. In first-order strictness analysis of a program with n functions each of k arguments we need to know the result for $n * k$ different argument sets. For such analyses it may be an advantage to examine the arguments one at a time rather than starting the algorithm with V^0 with $n * k$ arguments. If one of the needs required a very large or infinite set of indirect needs (*e.g.* if the underlying domain has

infinite height) it would be possible to stop the algorithm after a certain fixed number of iterations and return a safe top element value.

The advantage with this approach is that rather than failing the whole analysis, we only stop the analysis where the fixpoint could not be found easily. The rest of the arguments can be analysed unchanged. This makes it possible to decide on how one should balance the trade-off between precision and time complexity of the analysis.

5 Example

A classic example showing the complexity of fixpoint iteration with higher-order functions is the *concat* function.

$$\begin{aligned}
 \text{fold } f \ l \ v &= \text{case } l \ \text{of} \\
 &\quad \text{nil} \Rightarrow v \\
 &\quad y : ys \Rightarrow f \ y \ (\text{fold } f \ ys \ v) \\
 \text{append } x \ z &= \text{case } x \ \text{of} \\
 &\quad \text{nil} \Rightarrow z \\
 &\quad y : ys \Rightarrow y : \text{append } ys \ z \\
 \text{concat } l &= \text{fold } \text{append } \ l \ \text{nil}
 \end{aligned}$$

Assume *concat* has type $\text{list}(\text{list}(\mathbb{N}_\perp)) \rightarrow \text{list}(\mathbb{N}_\perp)$. In Wadler's analysis of lists [15, 1] the abstraction of values in $\text{list}(\mathbb{N}_\perp)$ are represented in a four-point domain $\{\perp, \infty, \perp \in, \top \in\}$ and the domain $\text{list}(\text{list}(\mathbb{N}_\perp))$ is abstracted as a six-point domain. The strictness versions of these function are here written using values $\{0, 1, 2, 3\}$ for the four-point domain and numbers

$\{0, 1, 2, 3, 4, 5\}$ for the six-point domain.

$$\begin{aligned}
\mathit{fold}^\sharp f l v &= \mathit{if } l = 0 \text{ then } 0 \text{ else} \\
&\quad \mathit{if } l = 1 \text{ then } f \ 3 \ (\mathit{fold}^\sharp f \ 1 \ v) \ \mathit{else} \\
&\quad \mathit{if } l = 5 \text{ then } v \ \vee \ (f \ 3 \ (\mathit{fold}^\sharp f \ 5 \ v)) \\
&\quad \mathit{else } (f \ (l - 2) \ (\mathit{fold}^\sharp f \ 5 \ v)) \ \vee \ (f \ 3 \ (\mathit{fold}^\sharp f \ l \ v)); \\
\mathit{append}^\sharp x z &= \mathit{if } x = 0 \text{ then } 0 \ \mathit{else} \\
&\quad \mathit{if } x = 1 \text{ then } \mathit{cons}^\sharp 1 \ (\mathit{append}^\sharp 1 \ z) \ \mathit{else} \\
&\quad \mathit{if } x = 3 \text{ then } z \ \vee \ (\mathit{cons}^\sharp 1 \ (\mathit{append}^\sharp x \ z)) \\
&\quad \mathit{else } (\mathit{cons}^\sharp 1 \ (\mathit{append}^\sharp x \ z)) \ \vee \ (\mathit{cons}^\sharp 0 \ (\mathit{append}^\sharp 3 \ z)); \\
\mathit{cons}^\sharp x xs &= \mathit{if } xs = 0 \text{ then } 1 \ \mathit{else } \mathit{if } xs = 1 \text{ then } 1 \ \mathit{else } (x + 2) \wedge xs; \\
\mathit{concat}^\sharp l &= \mathit{fold}^\sharp \mathit{append}^\sharp l \ 3;
\end{aligned}$$

To analyse the strictness of concat we should evaluate $\mathit{concat}^\sharp 0$. The needs are

$$\begin{aligned}
\mathit{concat}^\sharp(0) &\rightarrow 0 \\
\mathit{fold}^\sharp(\mathit{append}^\sharp(), 0, 3) &\rightarrow 0
\end{aligned}$$

and as the result is 0 the function is strict.

To evaluate the tail-strictness of concat we should evaluate $\mathit{concat}^\sharp 1$ the needs are then

$$\begin{aligned}
\mathit{concat}^\sharp(1) &\rightarrow 1 \\
\mathit{fold}^\sharp(\mathit{append}^\sharp(), 1, 3) &\rightarrow 1 \\
\mathit{append}^\sharp(3, 1) &\rightarrow 1 \\
\mathit{cons}^\sharp(1, 1) &\rightarrow 1
\end{aligned}$$

As the result is not 0 the function is not tail-strict and no further strictness analysis is needed (the function cannot be head-and-tail strict).

6 Related works

Chaotic fixpoint iteration is often compared to minimal function graphs in the literature. The direction of these concepts are, however, somewhat orthogonal. A minimal function graph semantics [9] is an instrumented

standard semantics which may be used to prove the correctness or soundness of certain program analyses. Normally, we do not expect or require the standard semantics to be implemented and as such it is not described as a fixpoint algorithm.

Pending analysis [17] is a fixpoint iterations strategy which also may be used for higher-order function. It does, however, require that the base is a boolean domain. Frontiers [7] may also be used for higher order functions but unlike the algorithm presented here, it will compute a global fixpoint. For higher-order strictness analysis, this has proved itself impractical.

The lifting of the first-order fixpoint algorithm to the higher-order case is inspired by work on constructing a minimal function graph semantics for a strict higher-order language [10].

7 Future work

In our system we have assumed that functions need all their arguments. In the algorithm, arguments are evaluated at the call and not depending on whether they are needed. Notice, however, that for non-flat domains needness does not imply strictness (consider $\lambda x.x + 1$ on the domain \mathbb{N}_0^∞). Nevertheless, it is possible to construct expressions for which our algorithm will fail to terminate whereas the fixpoint may be found in finite time. The expression

$$\mathbf{letfix} \ f(x) = 0 \ \mathbf{and} \ g(x) = g(x + 1) \ \mathbf{in} \ f(g(1))$$

will have the value 0 but the fixpoint of g cannot be found in finite time. It would be interesting to see how the algorithm could be changed so as to use a call-by-need strategy. More generally one may consider whether an optimal strategy exists which makes the fewest number of reevaluation when computing the fixpoint.

8 Conclusion

The paper describes a fixpoint iteration algorithm which may be used to compute parts of a fixpoint without reevaluating values which are not needed for the result. The method may also be used to detect indirectly needed calls in a recursion equation system. In the second part of the paper we extend the technique to higher-order functions.

References

- [1] G L Burn, C L Hankin, and S Abramsky. *The Theory of Strictness Analysis for Higher Order Functions*. In *Programs as Data Objects* (H Ganzinger and N D Jones, eds.), pp. 42–62. Volume 217 of LNCS. Springer-Verlag, Oct., 1986.
- [2] P Cousot and R Cousot. *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. In 4th POPL, Los Angeles, CA, pp. 238–252, Jan., 1977.
- [3] P Cousot and R Cousot. *Static determination of dynamic properties of recursive procedures*. In *Formal Description of Programming Concepts* (E J Neuhold, ed.). North-Holland, 1978.
- [4] P Cousot and R Cousot. *Abstract Interpretation and applications to logic programs*. *J of Logic Programming* **13**(2-3), pp. 103–180, July, 1992.
- [5] J Gallagher and M Bruyhooghe. *The Derivation of an Algorithm for Program Specialisation*. *New Gener. Comput.* **9**, pp. 305–333, 1991.
- [6] C Hankin and S Hunt. *Approximate Fixed Points in Abstract Interpretation*. In *ESOP’92*, pp. 219–232. Volume 582 of LNCS. Springer-Verlag, 1992.
- [7] S Hunt and C Hankin. *Fixed Points and Frontiers: A New Perspective*. *J of Functional Programming* **1**(1), pp. 91–120, Jan., 1991.
- [8] T Johnsson. *Lambda lifting: transforming programs to recursive equations*. In *FPCA’85*, Nancy, France. Volume 201 of LNCS. Springer-Verlag, Sept., 1985.
- [9] N D Jones and A Mycroft. *Data Flow Analysis of Applicative Programs using Minimal Function Graphs*. In 13th POPL, St. Petersburg, Florida, pp. 296–306, Jan., 1986.
- [10] N D Jones and M Rosendahl. *Higher-Order Minimal Function Graphs*. Unpublished. DIKU, Univ. of Copenhagen, Denmark, 1992.

- [11] A Mycroft. *The Theory and Practice of Transforming Call-by-Need into Call-by-Value*. In International Symposium on Programming'80, Paris, France (B Robinet, ed.), pp. 269–281. Volume 83 of LNCS. Springer-Verlag, Apr., 1980.
- [12] A Mycroft and M Rosendahl. *Minimal Function Graphs are not instrumented*. In WSA'92, Bordeaux, France, pp. 60–67. Bigre. Irista Rennes, France, Sept., 1992.
- [13] F Nielson and H R Nielson. *Finiteness Conditions for Fixed Point Iteration*. In LISP'92, San Francisco, CA, pp. 96–108. ACM Press, 1992.
- [14] R A O'Keefe. *Finite Fixed-Point Problems*. In International Conference on Logic Programming, pp. 729–743, May, 1987.
- [15] P Wadler. *Strictness analysis on non-flat domains (by abstract interpretation)*. In Abstract Interpretation of Declarative Languages (S Abramsky and C Hankin, eds.), chapter 12, pp. 266–275. Ellis-Horwood, 1987.
- [16] W Winsborough. *Multiple Specialization using Minimal-Function Graph Semantics*. J of Logic Programming **13**(2-3), pp. 259–290, July, 1992.
- [17] J Young and P Hudak. *Finding fixpoints on functional spaces*. Tech. Rep. YALEEU/DCS/RR-505. Yale Univ., Dec., 1986.