

Demand-Driven Higher-Order Fixpoint Iteration

Mads Rosendahl*

University of Roskilde, Dept. of Computer Science
Building 42.1, P.O. Box 260, DK-4000 Roskilde, Denmark
madsr@ruc.dk

Abstract. Our aim is to show that techniques from higher-order strictness analysis may be used as a general algorithmic principle in a functional programming language. Certain problems may be expressed as the search for the least solution that satisfy certain given properties. This is often done using some kind of fixpoint iteration.

We will present a fixpoint operation that can be used for second-order functions and extend this to higher-order functions. The technique is based on using partial function graphs to represent higher-order objects. The main problem in finding fixpoints for higher-order functions is to establish a notion of *neededness* so as to restrict the iteration to those parts of the function that may influence the result. This is here done through a uniform extension of the domain of values with need information. The result is an iteration strategy which will terminate if base domains are finite.

1 Introduction

Functional definitions can be circular in a number of different ways. Functions may be defined recursively in terms of themselves. We may have programs that use circular data structures, and functions may have circular argument dependencies. Circular data structures or circular dependencies can be used as a powerful algorithmic principle in a lazy language. The phrase 'circular programs' was coined by R. Bird [2] who showed how several passes over a data structure can be expressed in a single function, provided one may pass parts of the result of the function as an argument to the function.

In the field of attribute grammars circularity testing plays an important role. The traditional approach is to disallow circular attribute grammars but under certain restrictions it is well-defined and useful to allow circularity. A number of evaluators for circular attribute grammars have been reported in the literature [10][32].

In dataflow analysis and abstract interpretation the usual approach is to specify the analysis as a circular definition and subsequently solve it using some kind fixpoint iteration. For a number of years the implementation of higher order strictness analysis on non-flat domains [3] [15] [27] has attracted special attention. Not so much because of the importance of the analysis in its own right, but more as a bench mark for how well different techniques perform. In a

* This research was partly sponsored by the IT-University in Copenhagen

previous work [19] we constructed a strictness analysis for Haskell. It had some very attractive properties both regarding speed and precision. In this paper we will extend that work and show that the underlying fixpoint evaluation method is not restricted to strictness analysis but may be used for other purposes.

Second order fixpoint. Consider the following multiplication function written in ML

```
fun mul (x,y)=if x= 0 then 0 else y+mul(x-1,y);
```

The strictness function for `mul` is

$$mul^\#(x, y) = x \wedge (1 \vee (y \wedge mul^\#(x \wedge 1, y)));$$

It would be convenient if this function could be realized as

```
fun smul(x,y) = glb(x,lub(1,glb(y,smul(glb(x,1),y))));
```

It is, of course, not that simple. The program would fail to terminate since it is implemented recursively. As we shall see, we may define a fixpoint operator so that the strictness function can be defined as the fixpoint for a second order function.

```
fun Smul smul(x,y) =glb(x,lub(1,glb(y,smul(glb(x,1),y))));
val smul = Fix Smul;
```

This approach can be extended to the higher-order case so that fixpoint definition may be used in a more freely as a general algorithmic principle.

2 Domains and fixpoints

The theory behind the circular definitions is based on the fixpoint theorems of monotonic maps on complete lattices or continuous maps on chain-complete partially ordered sets. We are interested in computable fixpoint and will therefore impose some extra requirements regarding finite, unique representation and existence of certain extra operations. In general our approach is to separate the sets of values from the domain structure.

Domain. A domain D is a set with the following properties and operations.

- D has a partial order \sqsubseteq . A partial order is a reflexive, antisymmetric and transitive relation.
- D a least element \perp . Hence $\forall x \in D. \perp \sqsubseteq x$.
- D is chain-complete. For any subset (x_i) where $x_i \in D$ and where $x_i \sqsubseteq x_{i+1}$ (ie. a chain) there exists a least upper bound $\bigsqcup_i x_i: \forall d. (\forall i. x_i \sqsubseteq d) \Rightarrow \bigsqcup_i x_i \sqsubseteq d$.
- Any two elements of D with an upper bound will also have a least upper bound. The least upper bound of two elements x and y in D is written $x \sqcup y$.

Notice that we do not require general pair-wise least upper bounds to exist. We can therefore have flat domains and other domains without a top element. The last condition is an alternative to the stronger requirement of D being a complete lattice. We will frequently use domains where not all pairs of elements have a least upper bound. This is the case for flat domains larger than the two-point domain.

Any element of D which is the least upper bound of a chain without being an element of the chain is called a *limit point*.

We will require that all values which are not limit points can be represented in our programming language in a finite and unique way and that there is a total ordering of values for implementation purposes. This ordering does not need to be related in any way to the partial order on the domain. We will return to the implementation issue in a later section.

Function domain. Given a set S and a domain D . The set $S \rightarrow D$ of all functions from S to D is a domain. The domain has the following structure:

The bottom element is $\lambda x. \perp_D$, where \perp_D is the bottom element in D .

The ordering is defined as $f \sqsubseteq g \Leftrightarrow \forall x. f(x) \sqsubseteq g(x)$.

Assume $f \sqsubseteq h$ and $g \sqsubseteq h$ for functions f, g , and h , then $f \sqcup g = \lambda x. f(x) \sqcup g(x)$.

Given a chain (f_i) , then $\bigsqcup_i f_i = \lambda x. \bigsqcup_i f_i(x)$.

In the remainder of this paper $S \rightarrow D$ will denote this function domain without any implicit requirement of the functions being monotonic or continuous.

Monotonicity. A function f from a domain D_1 to D_2 is monotonic if it preserves the order: $\forall x, y \in D_1. x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$

The function f is *continuous* if it is monotonic and preserves the least upper bound of chains. $\forall (x_i) \in D. f(\bigsqcup_i x_i) = \bigsqcup_i (f(x_i))$

Fixpoint iteration. The central result in domain theory is that continuous functions on a domain have a least fixpoint. A similar result exists for monotonic functions on a complete lattice [33]. The fixpoint can be found as the limit of a chain starting with the bottom element with repeated applications of the function.

$$\perp \sqsubseteq f(\perp) \sqsubseteq f(f(\perp)) \sqsubseteq f(f(f(\perp))) \sqsubseteq \dots f(\bigsqcup_i f^i(\perp)) = \bigsqcup_i (f^i(\perp))$$

We will refer to the fixpoint of the function as $\text{lfp}f$ - the least fixpoint of f .

Partial fixpoint iteration. Let $G : (D_1 \rightarrow D_2) \rightarrow (D_1 \rightarrow D_2)$ be continuous and let S_i be a sequence of subsets of D_1 . We will now define a sequence of approximations ϕ_i as follows

$$\begin{aligned} \phi_0 &= \lambda x. \perp \\ \phi_{i+1} &= \lambda x \text{ if } x \in S_i \text{ then } G\phi_i x \text{ else } \phi_i x \end{aligned}$$

The aim is to find a limit ϕ_n which for some subset of D_1 computes the same function as $\text{lfp}G$.

Unfortunately the functions ϕ_i are not guaranteed to be monotonic, nor will the sequence in general be increasing.

As an example consider the domain $\{0, 1, 2\}$ with ordering $0 \leq 1 \leq 2$, and the function:

$$\begin{aligned} G\phi x &= \phi(\phi(x)) \oplus 1 \\ x \oplus 1 &= (x + 1) \sqcap 2 \end{aligned}$$

and sets $S_i = \{1\}$. The functions ϕ_i are then:

$$\begin{aligned} \phi_0 &= [0 \mapsto 0, 1 \mapsto 0, 2 \mapsto 0] \\ \phi_1 &= [0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 0] \\ \phi_2 &= [0 \mapsto 0, 1 \mapsto 2, 2 \mapsto 0] \\ \phi_3 &= [0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 0] \\ &\dots \end{aligned}$$

The problem is that function application in general is not monotonic. As a consequence a selective reevaluation of the function on needed arguments does not necessarily form an increasing chain and could result in non-termination of a fixpoint algorithm. Further conditions are needed to guarantee such a sequence to approximate the fixpoint.

Pseudo-monotonicity. A function $G : (D_1 \rightarrow D_2) \rightarrow (D_1 \rightarrow D_2)$ is pseudo-monotonic iff G preserves monotonicity, is continuous for monotonic arguments and

$$\begin{aligned} \forall f, f' : D_1 \rightarrow D_2, f \sqsubseteq f'. f \text{ monotonic} &\Rightarrow G(f) \sqsubseteq G(f') \wedge \\ f' \text{ monotonic} &\Rightarrow G(f) \sqsubseteq G(f') \end{aligned}$$

This property is called pseudo-monotonicity in [9] and called weak monotonicity in [11].

Lemma. Assume that $G_1 : (D_1 \rightarrow D_2) \rightarrow (D_1 \rightarrow D_2)$ is pseudo-monotonic and let us consider the function $G f x = f(G_1 f x)$. The function G is pseudo-monotonic.

Function needs. Given $G : (D_1 \rightarrow D_2) \rightarrow (D_1 \rightarrow D_2)$, a function $f : D_1 \rightarrow D_2$ and an element $x \in D_1$. The *needs* of the evaluation $G f x$ is the set of values in D_1 for which the function f will be called during the evaluation. This set will be denoted $\mathbf{N} G f x$.

Needs is in this way an internal property but may be externalised [28] as follows: Given Functions G, f and a value x , let $S = \mathbf{N} G f x$, then

$$\forall g. (\forall y \in S. f y = g y) \Rightarrow G f x = G g x$$

In other words: if a value is not needed then the function will return the same value independently of how argument was defined for that value.

This externalised view gives a lower bound of needed arguments. It is safe to let the needs be defined as a super set of arguments that are actually required to compute the value.

Chaotic iteration. Let $G : (D_1 \rightarrow D_2) \rightarrow (D_1 \rightarrow D_2)$ be pseudo-monotonic and let S_i be a sequence of subsets of D_1 . We will now define a sequence of approximations ϕ_i as follows

$$\begin{aligned} \phi_0 &= \lambda x. \perp \\ \phi_{i+1} &= \lambda x \text{ if } x \in S_i \text{ then } \phi_i x \sqcup G \phi_i x \text{ else } \phi_i x \end{aligned}$$

These functions will form an increasing chain (see below). We call this chain a chaotic fixpoint iteration if function needs for arguments in S_i and functions ϕ_j appear in a fair way in later sets S_k . A simple example of this is the sequence

$$S_{i+1} = \bigsqcup_{x \in S_i} \mathbf{N} G \phi_i x$$

This is the approach often referred to as the minimal function graph sequence [22] or as the Kleene sequence [25].

Stability. Rather than to consider the fairness of the reevaluation it is easier to examine the situation for a stabilised iteration sequence where for some n

$$\begin{aligned} \forall i > n. S_i = S_n, \phi_i = \phi_n \\ \forall x \in S_n. \mathbf{N} G \phi_n x \subseteq S_n \end{aligned}$$

If this is the case then we have

$$\forall x \in S_n. \phi_n x = (\mathbf{lfp} G)x$$

In the remainder of this section we will assume we have a stabilised iteration sequence for a function G and sets (S_i) , and that it stabilised after n iterations.

Lemma. The sequence (ϕ_i) is well-defined and forms an increasing chain.

Proof. Define the chain (θ_i) , where

$$\theta_0 = \lambda x. \perp \quad \text{and} \quad \theta_{i+1} = G \theta_i$$

Since G preserves monotonicity we know that θ_i is monotonic and that $\theta_i \sqsubseteq \theta_{i+1}$. By induction we prove that $\phi_i \sqsubseteq \theta_i$.

Assume for some i that $\phi_i \sqsubseteq \theta_i$, and given $x \in S_i$ then

$$\phi_{i+1} x = \phi_i x \sqcup G \phi_i x$$

and

$$\phi x \sqsubseteq \theta_i x \sqsubseteq \theta_{i+1} x \quad \text{and} \quad G \phi x \sqsubseteq G \theta_i x = \theta_{i+1} x$$

hence $\phi_i x$ and $G \phi_i x$ have $\theta_{i+1} x$ as an upper bound and the least upper bound will exist.

$$\phi_{i+1} x \sqsubseteq \theta_{i+1} x$$

Lemma. Define the sequence (ψ_i) , where

$$\psi_0 = \phi_n \quad \text{and} \quad \psi_{i+1} = G \psi_i$$

The sequence (ψ_i) is well-defined and forms an increasing chain with $\psi_i \sqsubseteq \theta_{i+n}$.

Proof. The proof is by induction as above.

Lemma. $\theta_i \sqsubseteq \psi_i$.

Proof. The proof is by induction.

Lemma. $\bigsqcup_i \theta_i = \bigsqcup_i \psi_i$.

Theorem. $\forall x \in S_n. \phi_n x = \mathbf{lfp} G x$.

Proof. By induction we prove that $\forall y \in S_n. \phi_n y = \psi_i y$ for all i . Assume that $\forall y \in S_n. \phi_n y = \psi_i y$ and given $x \in S_n$

$$\psi_{i+1} x = \psi_i x \sqcup G \psi_i x = \phi_n x$$

since

$$\begin{aligned} \psi_i x &= \phi_n x \\ G \psi_i x &= G \phi_n x = \phi_n x \end{aligned}$$

In the last step we used the externalised view of neededness. Since ϕ_n and ψ_i return the same value for needed arguments then evaluating G with any of these function will return the same result.

Comment. The theorem holds independently of the evaluation strategy. The traditional approach in fixpoint iteration is to use a breadth first strategy. But it is also possible to use depth-first.

In a fixpoint iteration some values are needed early on but not when the iteration has stabilised. Such values are called spurious arguments. The theorem shows that it is safe to omit those when no longer needed.

Termination. The theorem above states a partial correctness of certain types of fixpoint iteration. There are various ways to guarantee that the iteration terminates. The easiest is to require the domains D_1 and D_2 to be finite and that the sequence (S_i) is a chain.

Fixpoint iteration may be used as a general programming paradigm just as recursion or while-loops. When a program uses recursion we may be able to verify that it terminates but as a programming construct it will not in general terminate. We know that there are two sources of non-termination with recursion: circularity and an infinite set of dependent values. Similarly there are two sources of non-termination with fixpoint iteration: an infinite set of dependent values and infinite chain of approximations of result. The latter may occur when the domain D_2 has infinite height.

3 Realising domains

Using fixpoint iteration in practice requires efficient representation of values and function graphs. This is even more important in the higher-order case where arguments to functions may be functions or function graphs.

Given a domain D we assume that all but limit points may be represented uniquely in a programming language using a type D . We further assume that we can provide the following values and operations.

- A bottom value: `bot`
- A less-or-equal operation: `leq`
- A least upper bound operation: `lub`. This operation may fail since we only require that any two values with an upper bound has a least upper bound.
- A total order: `cmp`. This ordering is only used for implementation purposes and does not need to agree with the partial ordering of the domain. Equality in the two orderings must, however, agree since we require values in D to be represented uniquely. The ordering is implemented with an operation that return a value in the set `LESS`, `EQUAL`, `GREATER`. Alternatively one could have used values `-1`, `0`, and `1` as often done in the Java/C world.

We will use ML as implementation language but we only use constructions that are well-known in many programming languages. Values are represented in a similar to way in the Collections Framework in Java.

Integer domain: `intdom`. The set of natural numbers can be made into a domain using the usual integer ordering. We then get the domain \mathbb{N}_0^∞ of natural numbers with zero as bottom element and infinity as limit point.

```
val bot = 0;
fun leq(a,b:int) = a < b;
fun lub(a,b) = if a < b then b else a;
fun cmp(a,b:int)
  = if a = b then EQUAL else if a < b then LESS else GREATER;
```

String domain: `strdom`. We can define `strdom` as the flat domain of strings. As bottom element we will use the empty string and the ordering just specifies that the bottom element is less than or equal to all values in the domain. As total ordering we use the lexicographical ordering.

List domain. From any domain we can construct the domain of lists of values from that domain. The empty list is the bottom element and shorter lists are less than or equal to longer lists provided that they are related element-wise. If a domain D is represented using type `D` then we use type `D list` for the list domain.

Tuples. Two domains can be tupled into a domain of tuples of values tuple domain `tupdom`. The bottom element is the tuple of the bottom element from the two domains.

Powerset domain. For a domain we can construct the domain of subsets of values from that domain. The empty set is the bottom element and we use ordinary subset ordering. Sets are represented as sorted lists where elements are sorted according to the total ordering on the domain.

If a domain D is represented using type `D` then we use type `D list` for the powerset domain. Notice that the representation is the same as for list domains, but the ordering and least upper bound operations are different.

Least upper bound is set union and can be implemented as a merge of two sorted lists. Greatest lower bound is set intersection and can be performed in a similar fashion. On sets we also define a `member` and a set difference function.

Notice that the singleton set operation and the set difference function in general are not monotonic. Despite this power sets are very useful (and safe) in many applications.

We could also construct a set domain using a binary tree representation but we seem to use least upper bounds more frequently than membership tests and in that context the list representation is quite acceptable.

Domain of partial function graphs. For domains D_1 and D_2 we can construct the domain of function graphs from D_1 to D_2 . This domain is essentially just the powerset domain of tuples of values from D_1 and D_2 . Since we tend to use lookups more frequently than least upper bounds of graphs a representation using binary trees seems preferable. In our implementation language this can be achieved with a special dictionary structure: `(D1 * D2) dict`. We could, however, equally well have used a list of tuples.

We will also define some extra functions of function graphs:

```
lookup: ('a *'b) dict -> 'a -> 'b
isdef:  ('a *'b) dict -> 'a -> boolean
update: ('a *'b) dict -> ('a,'b) -> ('a *'b) dict
```

This domain does not use the (partial) ordering on D_1 and is the domain of all functions from the set D_1 to D_2 . One can also define the domain of all continuous functions from D_1 to D_2 but that is not what we will use here. The total ordering on D_1 is essential for the efficient representation of function graphs.

The `lookup` function will look-up an argument value in a graph and return the result value, if present. Otherwise it returns the bottom element of D_2 . The function `isdef` checks whether an argument is defined in the graph. The `update` operation updates a graph with a new argument-result pair.

4 Second-order fixpoint iteration

We are now ready to present a simple fixpoint operation for second order function. We call the evaluation strategy for truncated depth-first since it evaluates in a depth-first style as long as no circularities are detected. When we encounter circularities we truncate the evaluation and just use previously evaluated values or the bottom element instead.

Let D_1 and D_2 be two domains and let D_3 be the graph domain from D_1 to D_2 . Let further assume that we have defined the functions `lookup`, `isdef`, and `update` on the graph domain (see above), let `lub` be the least upper bound operation on D_2 , and `bot_g`, `eq_g` and `lub_g` is the bottom element, equality and least upper bound operations on the graph domain D_3 . We will here present the fixpoint operation for fixed domains D_1 and D_2 but in practice it may be parameterised on the domains so that the same function can be used for any domain.

Truncated depth-first evaluation.

```
fun Fix F =
  let val phi1 = ref bot_g (* previous *)
      val phi2 = ref bot_g (* current *)
      fun f x =
        if isdef(!phi2) x then lookup (!phi2) x else
        let val r0 = lookup (!phi1) x
            val _ = phi2 := update(!phi2) (x,r0)
            val r = F f x
            val _ = phi2 := update(!phi2) (x,lub(r,r0))
        in r end;
      fun iterate x = (
        phi1 := !phi2; phi2 := bot_g; f x;
        if eq_g(!phi1,!phi2) then lookup (!phi2) x
        else iterate x);
  in iterate end;
```

Discussion. The iteration uses two function graphs: `phi1` and `phi2` where `phi1` hold values found in the last iteration and new values are stored in `phi2`. The need sets are represented implicitly in the function graphs as the set of defined arguments.

There is no guarantee that users of this function provide a correct quasi-monotonic function. If the `lub` does not exist then it must be because of errors in the function `F`. Similarly if the iteration does not stop then it is because the iterated needs are infinite. We can extend the function with some error-checking to assist the user.

The truncated depth-first evaluation strategy has been used in a number of different versions in program analysis, circular attribute grammars etc. over the years [30], [10]. It is both simple and more efficient than the breadth first evaluation strategy often referred to as minimal function graphs. The latter was, however, not a fixpoint iteration method but a framework for defining and proving certain program analyses[22].

The approach here will automatically remove spurious calls [31][13]. In the stable situation the graph `phi2` will be tabulated for all arguments which directly or indirectly are need from the initial call - except for those which are already in the graph of found fixpoints.

Variations. An obvious extension is to memoize the fixpoint function so that repeated calls will not require new iteration. The memoization can easily be built into the fixpoint function since argument-result pairs already are stored in a graph.

The iteration stops when reevaluation of the argument and all dependent arguments does not result in changed values. This also means that the iteration is at least done twice. If the dependency is not circular we will do an unnecessary reevaluation. An alternative approach could be to store used values during evaluation in a separate graph and to reevaluate until the used-value graph is a subgraph of the computed graph in the iteration. This method will be optimal for programs without circular dependencies but at the cost of some extra book-keeping. Whether it is worth it depends on the function used as argument to the fixpoint.

When the iteration is truncated because of circularities we will use the value from the previous iteration or the bottom element. In some domains it could be relevant instead to use the least upper bound of all values in the graph where the arguments are less than or equal to the one we are searching for. Such an operation is quite costly but may be worth it if the argument domain has a complex structure.

Dependency based methods. Over the last decade a number of authors have proposed techniques which store a dependency graph during the iteration. The idea is to restrict reevaluations so they only occur when the graph has been changed for some of the arguments it depends on. The top-down evaluator [4] evaluates in a depth first order but with local computation of fixpoints. The neededness analysis method [25] uses breadth-first strategy, but limits the reevaluation to arguments that depend on arguments that have been changed. The time stamp solver WRT [11] uses extra time stamps in the work list to schedule reevaluation in a fair manner.

There is a significant cost at maintaining dependency information and any efficiency gain requires that each evaluation of the function F is costly.

5 Higher-order fixpoints

In the higher-order case arguments to functions are either simple values or functions. As an example consider the tiny program:

$$\begin{aligned} app\ f\ x &= f\ x \\ g\ f &= app\ f\ 42 \\ h\ x\ y &= x + y \\ k\ x &= g(h\ x) \end{aligned}$$

We will here show to define a recursive fixpoint operator that can solve these equations. The initial complication is that values are either simple values or function so we will use an explicit boxing and unboxing of values. In an untyped language it could have been done implicitly.

As set of values we will use a type V , where simple values are either strings or numbers.

```
datatype V = S of string | I of int | Bot
          | C of (V list -> V);
```

We can now define the fixpoint operator as

```
fun Fix_rec F (vs:V list) =
  let fun ff ((C f)::xs) = f xs
      | ff ws = F ff ws
  in ff vs end ;
```

The fixpoint operation has type

```
Fix_rec: ((V list-> V)-> V list-> V)-> V list-> V
```

The operator is bit more difficult to use than in the second-order case since we must include explicit boxing and unboxing of values. The functional corresponding to our tiny example is the function `F`:

```
fun add(I i1,I i2) = I (i1+i2)| add x=Bot;
fun F call [S "app",f,x] = call [f, x]
  | F call [S "g",f]     = call [S "app",f, I 42]
  | F call [S "h",x,y]   = add(x,y)
  | F call [S "k",x]     = add(x,call[S "g",call[S "h",x]])
  | F call xs = C (fn ys => call(xs@ys));
```

This may be used as follows:

```
val f = Fix_rec F;    val r = f [S "k",I 1];
```

The aim of the remainder of the paper is to show how to extend the fixpoint iteration strategy to the higher order case. The set V contains functions and we cannot directly make it into a domain with the requirements we introduced earlier. In a domain we need ordering relations and that can not be implemented for functions.

6 Argument needs

In the higher-order case a function may be called in two different ways: either directly using the function symbol f or indirectly using arguments. We will treat these cases separately and for now only consider calls to functions through arguments. Calls through arguments can also be described as an analysis of which *parts* of an argument will be used. If an argument is a function we may only need to call this function with a small set of arguments, hence only using a small part of this function. The purpose of this section is to establish a representation of this set.

The central idea in our exposition is to record the possible arguments to a partially applied function at the time it is used as an argument. In the example in the previous section the function `g` is a function which, when called with an argument, will call that argument with 42. The central idea is to tabulate arguments for the needed parts when it is used as argument and not when the argument is used as a function. The problem in this is to specify which parts of arguments will be needed prior to the call. What is needed of the arguments may both depend on the function and the arguments to the function. In this way we have an extra level of circularity in the higher-order case.

Since partially applied functions can appear nested inside expressions we will use a uniform way to record needs through out the type-structure. To record which parts of the arguments to a functional value will be used during evaluation we extend the domain with an extra part that describes needs.

Let U_t be the usual set we would use to describe values of type t . We will now instrument the description with extra need information in a domain U'_t

$$\begin{aligned}
U_{t_1 * \dots * t_n \rightarrow t_m} &= U_{t_1} * \dots * U_{t_n} \rightarrow U_{t_m} \\
U_{base} &= \dots \text{ numbers and strings} \\
U'_{t_1 * \dots * t_n \rightarrow t_m} &= U'_{t_1} * \dots * U'_{t_n} \rightarrow (U'_{t_m} * N_{t_1} * \dots * N_{t_n}) \\
U'_{base} &= U_{base} * \mathbb{U} \\
N_{t_1 * \dots * t_n \rightarrow t_m} &= \wp(U'_{t_1} * \dots * U'_{t_n}) \\
N_{base} &= \mathbb{U}
\end{aligned}$$

Here U_{base} is a domain of base values (e.g. numbers and strings) and \mathbb{U} a one-point domain.

The Res domain. We will now define a domain of higher-order function graphs. In the higher-order case the argument needs of functions should be an externalised property. For now we only consider how this should be represented; in the next section we will show how they are computed.

We will use a datatype `Res` to describe these instrumented values. They are either simple values or function graphs from `Res list` to tuples of `Res` values and needs described as `Res list list list`.

```

datatype Res = S1 of string | I1 of int | Bot1
              | G of ResGraph
withtype ResGraph = (Res list, Res*Res list list list) dict;
type Needs=Res list list list;

```

The datatype `Res` can be equipped with both a partial order and a total ordering for implementation purposes, a bottom element and a least upper bounds using the domain constructions described earlier. Similarly the types `Needs` and `ResGraph` can be given a domain structure.

Higher-order values. The last part of the representation of values is to include the extra graph information into ordinary values in the higher-order function. We will use a slightly changed definition of the type `V`

```

datatype V = S of string | I of int | Bot
           | C of ResGraph*(V list -> V);

```

We can establish a close relationship between the `V` set and the `Res` domain with two functions `v2r : V -> Res` and `r2v : Res -> V`.

```

fun v2r (S s) = S1 s
  | v2r (I i) = I1 i
  | v2r Bot  = Bot1
  | v2r (C(g, _)) = G1 g;

fun r2v (S1 s) = S s
  | r2v (I1 i) = I i
  | r2v Bot1   = Bot
  | r2v (G1 g) = C(g, fn xs => r2v(#1(lookup g (map v2r xs))));

```

We may note that for r in `Res` that $r = v2r(r2v(r))$ but for v in `V` that $v \sqsubseteq r2v(v2r(v))$.

At the outer level we can externalise argument needs with a function `callneed`. It will memo functions in the argument list, perform the function call and return the result together with the memo information.

```
callneed : (V list -> V) -> V list -> V * Needs
```

From a need set we may tabulate functions in a list for the needed parts. Such a function will have type

```
tabulate : V list -> Needs -> Res list
```

Both these function are fairly straightforward and will not be described further here.

The argument to the higher-order fixpoint operation has type $(V \text{ list} \rightarrow V) \rightarrow V \text{ list} \rightarrow V$. Values of type V cannot be compared so the aim it to change it into a function of type $\text{Res list} \rightarrow \text{Res} * \text{Needs}$. Function of this type may be tabulated and represented as a function graph.

7 Higher-order iteration

We are now ready to construct the higher-order fixpoint iterator. It works in a similar fashion to the truncated depth first evaluator. We have two graphs `phi2` and `phi1` of evaluated values from the current and the previous iteration. The iteration continues until stability has been achieved.

There is an extra level of iteration involved since the set of needed parts of arguments is defined circularly. Initially we assume that no part of functional arguments are needed. We will then attempt to evaluate the function and record which parts of the arguments were needed. The arguments are then tabulated for these parts, and this continues as long as more needs are recorded.

The fixpoint operation has the same type as the recursive operation defined earlier.

```
Fix: ((V list-> V)-> V list-> V)-> V list-> V
```

Internally it uses four functions

```
FF: (V list-> V) -> Res list-> V * Needs
gg: Needs -> V list -> V * Needs
ff: V list -> V
iterate: (V list-> V)-> V list-> V
```

The evaluation of functional arguments to functions is done during the tabulation of the arguments.

```
fun Fix F =
  let val phi1 = ref bot_g; val phi2 = ref bot_g
      fun FF ff rs =
        if isBot1 rs then (Bot,bot_n) else
        if isC1 rs then callneed fstcall (rs2v rs) else
        if isdef (!phi2) rs then
          let val (r1,n2)=lookup (!phi2) rs in (r2v r1,n2) end
        else
          let val _ = phi2 := update (!phi2) (rs,lookup(!phi1) rs);
              val (v1,nds) = callneed (F ff) (rs2v rs)
              val r2= lookup(!phi2)rs
              in phi2 := update (!phi2)(rs,lub_rn(r2,(v2r v1,nds)));
                (v1,nds) end;
          fun gg nd vs =
            let val (v1,n1) = FF ff (tabulate vs nd)
                val n2 = lub_n (n1,nd)
                in if eq_n(nd,n2) then (v1,n2) else gg n2 vs end
            and ff ws = let val (v,n)= gg bot_n ws in v end
```

```

fun iterate xs =
  let val _ = ( phi1:=(!phi2); phi2 := bot_g );
      val v1 = ff xs ;
  in if eq_g(!phi2,!phi1) then v1 else
      iterate xs
  end;
in iterate end ;

```

Variations. The function defined here does not memoize computed fixpoints but that extension is fairly easy to add the fixpoint operation.

The **V** and **Res** set includes strings and integers. The types could be parameterized so as to allow other more general domains in functions.

The outer iteration is based on the TDF approach. The other iteration techniques from the second-order case could also be used here since the outer iteration essentially is an iteration for a second-order function.

Example. Consider the program

$$\begin{aligned}
 g \ n \ k &= n \wedge ((k \top) \vee (g \ n \ (m \ n \ k))) \\
 m \ n \ k \ x &= k(n \wedge x)
 \end{aligned}$$

This is the strictness function for the CPS converted factorial function.

$$\begin{aligned}
 g \ n \ k &= \text{if } n = 0 \text{ then } k \ 1 \text{ else } g \ (n - 1) \ (m \ n \ k) \\
 m \ n \ k \ x &= k(n * x) \\
 id \ x &= x \\
 fac \ x &= g \ x \ id
 \end{aligned}$$

As part of a strictness analysis we may need to evaluate the call $f \top (\lambda x. \perp)$.

Expressed as a function on **V** values it may be written as

```

fun lub(I i1,I i2) = I(if i1>i2 then i1 else i2) | lub _ = I 0;
fun glb(I i1,I i2) = I(if i1>i2 then i2 else i1) | glb _ = I 0;

fun F1 call [S "g",n,k] =
  glb(n,lub(call [k,I 1], call [S "g",n,call [S "m",n,k]]))
| F1 call [S "m",n,k,x] = call [k,glb(n,x)]
| F1 call [S "bot",n] = I 0
| F1 call [S "top",n] = I 1
| F1 call [S "m1",n] = call[S "g",I 1,S "bot"]
| F1 call xs = Parteval call xs;

val f1 = Fix1 F1; val r = f1 [S "m1",I 1];

```

The result of the fixpoint iteration will be the following function graph.

```

[bot,1] => (0, [])
[g,1,bot] => (0, [])
[g,1,<>] => (0, [{}, {[1]}])
[g,1,[1] => (Bot, [])] => (0, [{}, {[1]}])
[g,1,[1] => (0, [])] => (0, [{}, {[1]}])
[m,1,bot] => (<>, [])
[m,1,bot,1] => (0, [])
[m,1,<>] => (<>, [])
[m,1,<>,1] => (Bot, [{}, {[1]}])
[m,1,[1] => (Bot, [])] => (Bot, [{}, {[1]}])
[m1,1] => (0, [])

```

The CPS-style does not give any special problems for the higher-order fixpoint iteration. In a closure based approach [31][23] we encounter the problem that the function m may be called with a second argument which is a closure of m . Such an approach will then generate an infinite set of closures and fail to terminate.

8 Conclusion

This paper is concerned with demand driven solution of fixpoint iteration. The immediate application is in the solution of demand versions of program analysis problems. The fixpoint iteration make it possible to implement more precise and complex higher-order analyses. The fixpoint operation is implemented in ML (Moscow ML [29]) and the source code will be made available electronically.

Representation techniques. Several methods are based on ways of representing the full tabulation with only a small set of points. Amongst these one should mention the frontiers algorithm [17,26] and binary decision diagrams. Such techniques may drastically reduce the need for tabulation though they still seem to require too much storage for certain examples. If it is not vital to find the least solution but only important to find a safe solution such methods may be connected with various approximation techniques so as to improve efficiency.

Conclusion We have presented a technique for solving fixpoint equations involving higher-order functions. The method may be used for higher-order program analysis and more generally as a tool in programming. For second-order functionals *needs* are sets of argument tuples. Our fixpoint iteration approach extends this notion of *needs* to the higher-order case where *needs* are higher-order functionals in arguments and needs.

References

1. K. R. Apt. From chaotic iteration to constraint propagation. *Lect. Notes Comp. Science*, 1256:36–??, 1997.
2. R. S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21(3):239–250, 1984.
3. G. L. Burn, C. L. Hankin, and S. Abramsky. The theory and practice of strictness analysis for higher order functions. In *Lecture Notes in Computer Science 217*. Springer-Verlag, 1985.
4. B. L. Charlier, O. Degimbe, L. Michel, and P. V. Hentenryck. Optimization techniques for general purpose fixpoint algorithms: practical efficiency for the abstract interpretation of prolog. *Lect. Notes Comp. Science*, 724:15–??, 1993.
5. T.-R. Chuang and B. Goldberg. A syntactic method for finding least fixed points of higher-order functions over finite domains. *Jour. Func. Prog.*, 7(4):357–394, 1997.
6. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *2nd Int. Symposium on Programming*, 1976.
7. 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*, pages 238–252, 1977.
8. P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions mathematical foundation. In *ACM Symposium in Artificial Intelligence and Programming Languages*, pages 1–12, 1977.

9. A. Dix. Finding fixed points in non-trivial domains: proofs of pending analysis and related algorithms. Tech. Rep. 107, University of York, 1988.
10. R. Farrow. Automatic generation of fixed-point-finding evaluators for circular but well-defined attribute grammars. In *Proc. SIGPLAN '86 Symposium on Compiler Construction*, volume 21, pages 85–98, 1986.
11. C. Fecht and H. Seidl. An even faster solver for general systems of equations. *Lecture Notes in Computer Science*, 1145:189–??, 1996.
12. A. Ferguson and J. Hughes. Fast abstract interpretation using sequential algorithms. In *Static Analysis Symposium 1993*, pages 45–59, 1993.
13. J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Gener. Comput.*, 9:305–333, 1991.
14. C. Hankin and D. L. Metayer. Deriving algorithms from type inference systems: application to strictness analysis. In *Proc. 21st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 202–212. ACM Press, 1994.
15. J. Hughes. Strictness detection in non-flat domains. In *Proc. a Workshop on Programs as Data Objects*, pages 112–135, 1986.
16. R. J. M. Hughes. *Lazy Memo-Functions*. 1985.
17. S. Hunt and C. Hankin. Fixed points and frontiers: A new perspective. *Journal of Functional Programming*, 1(1):91–120, 1991.
18. Jayaraman and Plaisted. Functional programming with sets. In *FPCA '87*, pages 194–211, 1987.
19. K. D. Jensen, P. Hjørnesen, and M. Rosendahl. Efficient strictness analysis of haskell. In *SAS'94 (Lect. Notes Comp. Science)*, pages 346–362. Springer-Verlag, 1994.
20. T. Johnsson. Attribute grammars and functional programming. In *FPCA '87*, pages 154–173, 1987.
21. L. G. Jones. Efficient evaluation of circular attribute grammars. *toplas*, 12(3):429–462, 1990.
22. N. D. Jones and A. Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *13th POPL*, pages 296–306, 1986.
23. N. D. Jones and M. Rosendahl. Higher-order minimal function graphs. Unpublished, DIKU, Univ. of Copenhagen, Denmark, 1992.
24. S. P. Jones and C. Clack. Finding fixpoints in abstract interpretation. In *Abstract Interpretation of Declarative Languages*, pages 246–265, 1987.
25. N. Jørgensen. Finding fixpoints in finite function spaces using neededness analysis and chaotic iteration. In *Static Analysis Symposium 1994*, pages 329–345, 1994.
26. C. Martin and C. Hankin. Finding fixed points in finite lattices. In *FPCA '87*, pages 426–445, 1987.
27. A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Int. Symposium on Programming'80*, pages 269–281, 1980.
28. A. Mycroft and M. Rosendahl. Minimal function graphs are not instrumented. In *WSA '92*, pages 60–67.
29. S. Romanenko, C. Russo, and P. Sestoft. *Moscow ML Language Overview*. Russian Academy of Science, Moscow, 2000.
30. M. Rosendahl. Abstract interpretation and attribute grammars. Ph.d. thesis, Cambridge Univ., 1992.
31. M. Rosendahl. Higher-order chaotic iteration sequences. In *PLILP'93*, pages 332–345, 1993.
32. S. Sagiv, O. Edelstein, N. Francez, and M. Rodeh. Resolving circularity in attribute grammars with application to data flow analysis. In *16th POPL*, pages 36–46, 1989.
33. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5:285–309, 1955.