ISSN 0105-8517

Local Computation of Simultaneous Fixed-Points

Henrik Reif Andersen

DAIMI PB – 420
October 1992

COMPUTER SCIENCE DEPARTMENT
AARHUS UNIVERSITY
Ny Munkegade, Building 540
DK-8000 Aarhus C, Denmark

TRYKKESTED:
MATEMATISK INSTITUT
AARHUS UNIVERSITET

PB - 420 H.R. Andersen: Local Computation of Simultaneous Fixed-Points

Local Computation of Simultaneous Fixed-Points*

Henrik Reif Andersen

Department of Computer Science, Aarhus University
Ny Munkegade, DK-8000 Århus C, Denmark
E-mail: henrikan@daimi.aau.dk

Abstract

We present a very simple, yet general algorithm for computing simultaneous, minimum fixed-points of monotonic functions, or turning the viewpoint slightly, an algorithm for computing minimum solutions to a system of monotonic equations. The algorithm is local (demand-driven, lazy, ...), i.e. it will try to determine the value of a single component in the simultaneous fixed-point by investigating only certain necessary parts of the description of the monotonic function, or in terms of the equational presentation, it will determine the value of a single variable by investigating only a part of the equational system.

In the worst-case this involves inspecting the complete system, and the algorithm will be a logarithmic factor worse than a global algorithm (computing the values of all variables simultaneously). But despite its simplicity the local algorithm has some advantages which promise much better performance on typical cases. The algorithm should be seen as a schemata that for any particular application need to be refined to achieve better efficiency, but the general mechanism remains the same. As such it seems to achieve performance comparable to, and for some examples improving upon, carefully designed *ad hoc* algorithms, still maintaining the benefits of being local.

We will illustrate this point by tailoring the general algorithm to concrete examples in such (apparently) diverse areas as type inference, model checking, and strictness analysis. Especially in connection with the last example, strictness analysis, and more generally abstract interpretation, it is illustrated how the local algorithm provides a very minimal approach when determining the fixed-points, reminiscent of, but improving upon, what is known as Pending Analysis [19]. In the case of model checking a specialised version of the algorithm has already improved on earlier known local algorithms [2,1].

1 Introduction

Fixed-points arise everywhere in computer science, when giving semantics of programming languages, in program analysis, in program optimization, in program verification, and

*This work is supported by the Danish Natural Science Research Council

many other situations. We will present a general algorithm for computing such fixed-points in complete partial orders (cpo's), and hence lattices, of finite height. (Any poset of finite height is trivially a cpo, but we stick to the term cpo because we will only apply the finiteness property when it is strictly necessary.) The algorithm will be well-suited to situations where the fixed-points belong to large products of cpo's, and examples of type inference problems, strictness analysis, and model checking problems will be shown to fit into the general framework and yield efficient algorithms.

The simultaneous fixed-points will be described as solutions to sets of monotonic equations and the algorithm works on such descriptions in a local fashion, i.e. the algorithm will compute the fixed-point 'demand-driven' or *locally*, assuming that only some of the components of the fixed-points are really of interest, start from one such component and investigate what is necessary to determine the value. In this respect it differs from most fixed-point finding algorithms which tends to *globally* compute the complete fixed-point. A notable example of such an algorithm is due to Kildall [8], which describes the algorithm as solving a problem of dataflow analysis. We show how this, indeed very simple, global algorithm in a version suitable for the present framework, is related to the local, and show that the possible benefits of the local algorithm compared to the global has a cost of a logarithmic factor in the worst-case, but the typical case would out-perform the global algorithm.

The component of interest could be for instance a variable denoting 'error' in the case of type inference, and hence the algorithm would only search locally for an error in the program under consideration, without necessarily assigning types to all program fragments. In the case of strictness analysis, this component will typically be a function applied to one particular argument, which will then be computed without necessarily computing the behaviour of the function on *all* arguments as would the global algorithm.

To be more precise, we will consider systems of equations on the following form:

$$\begin{aligned} x_1 &= f_1(x_{i_{11}}, \ldots, x_{i_{1a_1}}) \\ &\vdots \\ x_n &= f_n(x_{i_{n1}}, \ldots, x_{i_{na_n}}) \end{aligned} \tag{1}$$

where for $1 \leq j \leq n, 1 \leq k \leq a_j$ we have $1 \leq i_{jk} \leq n$. Associated with each variable x is a set of values D_x which we require to be a complete partial order (cpo) with a bottom element denoted by $\bot_{D_x}$ and of finite height. For convenience and when there is no risk of confusion we will write D_j for D_{x_j} and f_{x_j} for f_j. We use $s(x)$ to denote the tuple of *sons* of x, i.e.

$$s(x_j) = (x_{i_{j1}}, \ldots, x_{i_{ja_j}})$$

and we use $s^{-1}(x)$ to denote the set of *parents* of x, i.e. the set

$$s^{-1}(x) = \{x_j | \exists k. s(x_j)_k = x\}.$$

Moreover, for $1 \leq j \leq n$ the function f_j must be monotonic with type

$$f_j : D_{i_{j1}} \times \ldots \times D_{i_{ja_j}} \to D_j$$

We refer to a monotonic equation system as the tuple (V, s, D, f), where $V = (x_1, \ldots, x_n)$ is the tuple of variables, s the 'sons of' function, $D = (D_1, \ldots, D_n)$ the tuple of value domains, and $f = (f_1, \ldots, f_n)$ the tuple of functions.

As it is well-known a monotonic equation system (1) has a minimum solution, the minimum fixed-point of f, given by

$$\mu f = \bigsqcup_{i \in \omega} f^i(\vec{\bot}) \tag{2}$$

with the definition

$$\begin{aligned} f^0(x) &= x \\ f^{i+1}(x) &= f(f^i(x)) \end{aligned}$$

yielding an increasing chain $\vec{\bot} \sqsubseteq f(\vec{\bot}) \sqsubseteq f^2(\vec{\bot}) \sqsubseteq \ldots$, where $\sqsubseteq$ is the ordering on the cpo $D_1 \times \ldots \times D_n$, and $\bigsqcup$ is the least upper bound of increasing chains.

2 Algorithm

Tentatively the algorithm will proceed as follows. We associate with each variable x a *marking* $m(x)$, which denotes the current value of x. Initially the marking of all variables will be 'unknown'. We assign the variable of interest, x^0 say, marking $\bot$ and puts x^0 in a set of *active* variables, for which the right-hand sides will be inspected to verify that the current marking is identical to whatever the right-hand side evaluates to. In evaluating right-hand sides we will always try to inspect as few of the sons as needed, utilizing that the function might be determined by the current marking of only some of the sons. When evaluating a right-hand side it might of course turn out that we do indeed need the value of some sons, which will be assumed to have the value $\bot$ and put on the list of active nodes to be examined. In doing so, we keep track of dependencies between variables, and whenever it turns out that a variable changes its marking (actually, it can only increase) all variables that might depend on this particular variable is put in the active set to be reexamined. At some point the set of active nodes will become empty, and we have actually found (part of) the fixed-point.

This approach has two benefits:

1. Only variables *reachable* from the root variable x^0 through the 'sons-of' relation will ever be investigated, a kind of *syntactic dependency analysis.*

2. Moreover, only variables that turns out to be *actually needed* in determining a right-hand side will ever be investigated, a kind of *semantic dependency analysis.*

Of course, in the worst case the set of variables visited might be precisely the set of variables 'syntactically' reachable from the root variable, but potentially much fewer might be needed. Another important property of the sketched algorithm is that all this happens on-the-fly: The complete system does not have to be computed *a priori*, but the right-hand sides can be supplied on demand.

2.1 'Unknown' values

In order to formally present the algorithm we will introduce notation for 'unknown' values. Technically, we will define a special kind of lifting $(\_)_?$ of cpo's, and characterize a class of functions which behaves properly with respect to unknown arguments.

Let the ?-*lifting* $D_?$ of a poset D be defined by

$$D_? = \{?\} \cup \{\lfloor d \rfloor | d \in D\}$$

with the ordering $\leq_D$ defined by

$$\forall x \in D_?.\ ? \leq_{D_?} x, \text{ and}$$
$$\forall a, b \in D.\ \lfloor a \rfloor \leq_{D_?} \lfloor b \rfloor \ \Leftrightarrow\ a \leq_D b,$$

hence ? is a bottom element of $D_?$.

Definition 1 For $1 \leq i \leq k$ let D_i and D be posets and suppose that $f : (D_1)_? \times \ldots \times (D_k)_? \to D_?$ is a function. Then f is

- ?-*strict* if
$$f(?, \ldots, ?) = ?,$$
- ?-*reflecting* if
$$f(x_1, \ldots, x_k) = ? \ \Rightarrow\ \exists i.\ x_i = ?,$$
- ?-*monotonic* if
$$\vec{x} \leq \vec{y} \ \Rightarrow\ f_v(\vec{x}) \leq f_v(\vec{y}) \text{ or } f_v(\vec{y}) = ?,$$

and finally,

- ?-*faithful* if

$$f(x_1, \ldots, x_{i-1}, ?, x_{i+1}, \ldots, x_k) = y \ \Rightarrow\ \forall x.\ f(x_1, \ldots, x_{i-1}, \lfloor x \rfloor, x_{i+1}, \ldots, x_k) = y.$$

A function fulfilling all four is said to be ?-*nice*.

These four notions are intended to capture some intuitive understanding of unknown values: f must depend on its arguments, f can only yield an 'unknown' result if one of its arguments are unknown, f is monotonic in the usual sense, except that sometimes, increasing the arguments can cause f to yield an unknown value, and finally, if f yields a known value with some argument unknown, it must be independent of that particular argument (with the other arguments fixed).

We will say that a ?-nice function $f' : (D_1)_? \times \ldots \times (D_k)_? \to D_?$ is a ?-*nice extension* of $f : D_1 \times \ldots \times D_k \to D$ if

$$\forall x_i \in D_i.\ f'(\lfloor x_1 \rfloor, \ldots, \lfloor x_k \rfloor) = \lfloor f(x_1, \ldots, x_k) \rfloor \tag{3}$$

i.e. on the lifted elements f' agrees with f. Notice, that for a ?-nice extension f' of f we have

$$\mu x.f'(x \sqcup \lfloor \bot \rfloor) = \lfloor \mu x.f(x) \rfloor \tag{4}$$

which follows directly by induction on the approximants, utilizing (3). (We have taken the liberty of writing $\lfloor \bot \rfloor$ for $(\lfloor \bot_{D_1} \rfloor, \ldots, \lfloor \bot_{D_n} \rfloor) \in (D_1)_? \times \ldots \times (D_n)_?$.)

In a trivial manner, all monotonic functions $D_1 \times \ldots \times D_k \to D$ can be extended to ?-nice functions by mapping any vector of arguments which includes a ? to ?. However, this will result in an algorithm which does not fully exploit the possibility of minimizing the search because it makes the semantic dependency analysis void; all functions will require the presence of all arguments before giving a value.

As an example of how to choose better ?-nice functions, we consider the case of two-point domains.

Example 1 Consider the situation where all D_i's are identical to the two-point cpo $\mathbf{O} = \{0, 1\}$ with $0 < 1$. We can define ?-nice extensions of the usual boolean connectives $\wedge$ and $\vee$ as shown in the two tables.

$\wedge$	?	0	1
?	?	0	?
0	0	0	0
1	?	0	1

$\vee$	?	0	1
?	?	?	1
0	?	0	1
1	1	1	1

These extension are non-trivial, e.g. $0 \wedge ? = 0$. Notice also the ?-monotonic behaviour showing that $\wedge$ is not monotonic in the usual sense: For $(0, ?) < (1, ?)$ we get

$$\wedge(0, ?) = 0 \wedge ? = 0 > ? = 1 \wedge ? = \wedge(1, ?).$$

Hence although at one stage the value of the conjunction can be determined by looking at only the first argument, if this argument increases its value to 1, the second argument must be inspected in order to determine the value of the conjunction.

The ?-nice extensions here capture the well-known facts that sometimes the values of a conjunction/disjunction can be determined by considering only one of the arguments. □

2.2 The local algorithm

To prove the algorithm correct we will use a lemma, which captures a key property of fixed-points. Recall, that an *embedding-projection pair* (j, p) between to cpo's D and E is a pair of monotonic functions $j : D \to E$ and $p : E \to D$, which satisfy

$$(i)\ p \circ j = id_D \qquad (ii)\ j \circ p \leq id_E,$$

where $\leq$ is the pointwise extension of the ordering of E to functions $E \to E$.

Lemma 2 (Projection lemma) *Suppose D and E are cpo's with bottoms. Let (j, p) be an embedding-projection pair between D and E with p being ω-continuous. For any ω-continuous function $f : E \to E$ and element $y \in D$, which satisfy*

$$(i)\ \ \forall x \in p^{-1}(y).\ p(f(x)) = y$$
$$(ii)\ \ y \leq_D p(\mu x.f(x))$$

we have

$$y = p(\mu x.f(x)).$$

Proof: See appendix A □

To understand the role of the projection lemma assume we have given a monotonic equation system (V, s, D, f), and take $E = \Pi_{x\in V}(D_x)_?$, the V-indexed product of the $(D_x)_?$'s. Let B be a subset of the variables, and take $D = \Pi_{x\in B}(D_x)_? \times \Pi_{x\notin B}\{?\}$. Taking $j : D \to E$ to be the obvious inclusion, and defining $p : E \to D$ by

$$p(m)(v) = \begin{cases} m(v) & \text{if } v \in B \\ ? & \text{otherwise} \end{cases}$$

it is easy to see that (j, p) is indeed an embedding-projection pair. Now, suppose we have given a particular $m \in D$ (playing the role of y in the lemma) with the property that all elements in E which projects to m, maps through f to elements which projects to m; in other words no matter what elements are supplied for the variables outside B, the value of f at this modified m stays the same on the variables in B. Then, if we also know, that m is indeed less than the projection of the minimum fixed-point of f, we conclude that m is *precisely* the projection of the fixed-point of f. In more pragmatic terms, this means that we have found a *part* of the fixed-point, namely the part corresponding to the variables in B.

The local algorithm is stated in figure 1. We have used a syntax from which the semantics should be obvious. The active set of variables mentioned previously is denoted by A, and $b(x)$ is a vector of values in $\mathbf{O} = \{0, 1\}$ with the j'th coordinate equal to 1 if the j'th son has previously been inspected. For $b \in \mathbf{O}$ we have used the conditional $b \to x$ defined by

$$b \to x = \begin{cases} \bot & \text{if } b = 0 \\ x & \text{if } b = 1 \end{cases}$$

Theorem 3 (Correctness) *The algorithm of figure 1 correctly computes part of the fixed-point μf, i.e. it terminates with a set B and a value assignment m, s.t. $x \in B$, and $m|_B = \mu f|_B$.*

Proof: (Sketch) The correctness proof is straightforward exploiting the projection lemma and using Hoare Logic with the following invariant I for the while-loop:

$$\begin{array}{ll} i) & \forall v \in V.\ m(v) \le f'_v(m(s(v))) \ \&\ m(v) \le (\mu x.f'(x \sqcup \lfloor\bot\rfloor))(v) \\ ii) & d(v) = \{u \in s^{-1}(v) | \exists i.b_u(i) = 1 \ \&\ s(u)_i = v\} \\ iii) & b_v(j) = 1 \ \Rightarrow\ m(s(v)_j) \neq ? \\ iv) & \{v \in V | ? \neq m(v) < f'_v(m(s(v)))\} \subseteq A \subseteq V \\ v) & x \in \{v | m(v) \neq ?\} \end{array}$$

(We have used $m(s(v))$ to mean $(m(s(v)_1), \ldots, m(s(v)_{a_v}))$.) We leave out the proof that this is a valid invariant. Now, when $A = \emptyset$, I implies by (i) and (iv) that

$$\forall v.m(v) \neq ? \ \Rightarrow\ m(v) = f'_v(m(s(v))) \ \&\ m(v) \le (\mu\vec{x}.f'(\vec{x} \sqcup \lfloor\bot\rfloor))(v).$$

Input: Monotonic equational system (V, s, D, f), a ?-nice extension f' of f, and a variable $x^0 \in V$.
Output: A marking $m : \Pi_{v\in V}(D_v)_?$ and a set $B \subseteq V$ with $x^0 \in B$ such that m equals μf on B.

for all $v \in V$ **do** $m(v) := ?$ $d(v) := ?$
$A := \{x^0\}$ $m(x^0) := \lfloor\bot\rfloor$ $d(x^0) := \emptyset$ $b(x^0) := \vec{0}$
while $A \neq \emptyset$ **do**
  pick an $x \in A$ $A := A \setminus \{x\}$
  $r := f_x(b(x)_1 \to m(s(x)_1), \ldots, b(x)_{a_x} \to m(s(x)_{a_x}))$
  if $r = ?$ **then**
    pick a j *s.t.* $b(x)_j = 0$
    $b(x)_j := 1$
    if $m(s(x)_j) = ?$ **then**
      $m(s(x)_j) := \lfloor\bot\rfloor$ $d(s(x)_j) := \{x\}$ $b(s(x)_j) := \vec{0}$
      $A := \{s(x)_j, x\} \cup A$
    else
      $d(s(x)_j) := \{x\} \cup d(s(x)_j)$
      $A := \{x\} \cup A$
    fi
  else if $r > m(x)$ **then**
    $m(x) := r$ $A := d(x) \cup A$
  fi
od

Figure 1: The local algorithm.

hence by equation (4) we get

$$\forall v.m(v) \neq ? \ \Rightarrow\ m(v) = f'_v(m(s(v))) \ \&\ m(v) \le (\mu f)(v)$$

and as f'_v is a ?-nice extension of f_v we finally get

$$\forall v.m(v) \neq ? \ \Rightarrow\ m(v) = f_v(m(s(v))) \ \&\ m(v) \le (\mu f)(v)$$

Taking $B = \{v | m(v) \neq ?\}$, the result follows from the discussion following the projection lemma. □

The correctness proof is independent of whatever particular implementation is used for the datastructures of the algorithm. These choices will of course have a great impact on the complexity of the algorithm. To illustrate this, let us consider what a general implementation could look like. The set of active nodes A could be implemented as a stack with constant insertion and extraction times, and the algorithm will behave very much as a depth-first traversal (Tarjan [16]), m, d, and b are all partial maps, with ?

meaning 'outside domain of definition', that could be implemented by some balanced search tree with search time bounded by $\log n$, n being the number of variables. Each entry in m is a simple value; in d it's a dynamically growing list, and each entry in b could be implemented as a simple counter bc with the understanding that the $b(x)_j$ of the algorithm equals 1 if the counter $bc(x)$ is bigger than j, i.e. $b(x)_j = 1$ *iff* $bc(x) > j$.

To sketch the complexity analysis, let $ht(D)$ be the *height of* D i.e. the length of the longest strictly increasing chain in D minus one, and let $c(f)$ be the maximal cost of evaluating f on any of its arguments. Now, observe that every variable will appear in A at most $\sum_{u \in s(v)}(ht(D_u)+1)$ times, as a variable is only reentered into A if one of its sons gets an increased value, which for each son only can happen $ht((D_u)_?) = ht(D_u)+1$ times. Hence, the worst-case complexity is, by this informal amortized cost argument:

$$O(\sum_{v \in V}(c(f_v) \sum_{u \in S(v)}(ht(D_u)+1))\log n)$$

Using more uniform bounds on the functions and domains we arrive at:

Theorem 4 (Complexity) *If the cost of computing each of the functions of f is bounded by c, the arity bounded by a, and the height of the cpo's bounded by h, then the worst-case complexity is*

$$O(ncah \log n).$$

In practice this bound is *very* pessimistic, it will only be reached in very pathological circumstances: The fixed-point will be identical to the 'highest' element in the cpo and as the algorithm proceeds all the variables change their values in the smallest possible steps. Moreover, all the variables must be reachable from the root variable in the syntactical and the semantical sense.

It is, however, very difficult to express anything about the behaviour of the algorithm on typical cases and even to define what a typical case is. However, the examples to be shown later will give some indication of when it is successful.

Before proceeding with the discussion on the local algorithm, let us briefly compare the local algorithm with the *global* algorithm of Kildall [8], shown in figure 2, suitable reformulated to fit our framework. In principle it is constructed from the local algorithm by initializing the marking of all variables to $\lfloor\bot\rfloor$, inserting all variables in the active set of variables, and removing the semantic dependency by analysis always taking $d(x) = s^{-1}(x)$. Of course the branch corresponding to f_x yielding an unknown result is removed. By an argument analogous to the local case, the worst-case complexity is $O(ncah)$, hence the worst-case behaviour of the local algorithm is a logarithmic factor worse, due to the searches in connection with the partial maps. This means that from a strict complexity argument our algorithm is worse, but as it has already been pointed out the local algorithm offers some benefits which the global lacks.

We now turn attention to three examples showing how the local algorithm can be applied. Not all details are included, the general lines are sketched, and emphasis is put on the points of general interest.

Input: Monotonic equational system (V, s, D, f)
Output: A marking $m = \mu f$

for all $v \in V$ **do** $m(v) := \lfloor\bot\rfloor$
$A := V$
while $A \neq \emptyset$ **do**
  pick a $y \in A$ $A := A \setminus \{y\}$
  $r := f_x(m(s(x)))$
  if $r > m(x)$ **then**
    $m(x) := r$ $A := s^{-1}(x) \cup A$
  fi
od

Figure 2: Kildall's global algorithm.

3 Example: Strictness analysis

Our first example will be on *strictness analysis* as introduced by Mycroft [10] in a version due to Wadler [17]. However, most of the remarks and constructions apply equally well to abstract interpretation in general.

We assume that we have given an *abstract program* as set of mutually recursive function declarations:

$$\begin{aligned} f_1(x_{11}, x_{12}, \ldots, x_{1a_1}) &= e_1 \\ &\vdots \\ f_n(x_{n1}, x_{n2}, \ldots, x_{na_n}) &= e_n \end{aligned}$$

where the free variables of the body e_j is included in $\{x_{j1}, \ldots, x_{ja_j}, f_1, \ldots, f_n\}$. (We will not bother to define any particular syntax for expressions.) Each function has a type

$$f_j : D_{j1} \times \ldots \times D_{ja_j} \to D_j$$

where the D's are cpo's of finite height with bottom (for strictness analysis this will typically be finite lattices) and we assume that all the bodies are indeed well-defined with respect to this typing.

To rephrase this in terms of a monotonic equational system (1) we introduce a variable $v_{f_j;\vec{x}}$ for each pair of function and argument $\vec{x}$ in the ?-lifted product

$$(D_{j1})_? \times \ldots \times (D_{ja_j})_?.$$

The equation for $v_{f_j;\vec{x}}$ will be a bit special; we will think of the right-hand side $g_{f_j;\vec{x}}$ as a function on *all* variables of the system, i.e. one for each pair of function and possible argument. Although finite, this can be quite a lot of variables!

Now, to evaluate $g_{f_j;\vec{x}}$ we simply proceed, by for instance executing an interpreter for lambda-expressions (if that is the language we have used for the expressions) and when at some point we need the value of a function at a specific argument which is 'unknown', we suspend the evaluation, return the value ?, and proceed with the algorithm, which picks a son (this should of course be the one that made us halt in the first place), assumes it has the value $\lfloor\bot\rfloor$ and proceeds.

To tabulate a function for a range of values U we simply execute the algorithm for each element of U, reusing, of course, earlier stored results.

For this to be valid, we must ensure that the right-hand sides are all monotonic. This could be done by restricting the syntax, but care has to be taken if expressions like $f(f(\ldots),\ldots)$ are to be allowed (cf. the problems with Pending Analysis reported in [5]). We consider this problem to be outside the scope of the current discussion. In the case of higher-order functions, e.g.

$$f : (D \to E) \to E$$

another difficulty arises implicitly: When f is applied to an argument h, we must search for the node $v_{f;h}$, which involves comparing functions. Assuming that D and E are simple domains this is not too bad, the function space will be reasonably small and the task not impossible. Finally, very few functional programs (except perhaps when using continuations) seem to apply functionals on many different functions, so in practice few comparisons will be needed. Moreover, any of the techniques for compactly representing functions could be used to speed up this part.

Considering the second-order case (like with f above) this approach of computing strictness has two major benefits compared to iterative algorithms:

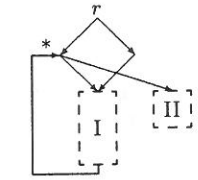
1. Only first order functions will ever have to be compared, no second order functions must be compared to determine stability of the iteration, and in general comparison of n'th order functions for $n+1$'st order analysis.

2. Potentially only a very small proportion of the huge number of possible function-argument pairs will be needed.

In this second respect our local algorithm is very similar to Pending Analysis [19], but it can be *exponentially faster*, due to the explicit treatment of dependencies. To see why, we first briefly describe the Pending Analysis:

> As just described the evaluation start with a function applied to one particular argument. If in evaluating such a function application, any previously visited function-argument pair is reencountered, the value is simply assumed to be bottom. In the case of a lattice of height one this suffice to make sure that the minimum fixed-point will be correctly computed and in the general case the application is re-evaluated until it is stable, every time in a recursive occurrence of a call, using the previously computed value.

To see how this differs from our algorithm, consider the following graph of function calls assumed to occur in the evaluation of a function application. Each arrow indicate a function call.



The pending analysis will traverse this as a tree from the root r, i.e. the root of (I) will be visited twice and so will all nodes in (I). If the structure of (I) is again as above it is not difficult to see that the Pending Analysis will perform exponentially many calls, and this is *not* merely a problem that can be solved using 'dynamic programming' or 'memorization'. To see why, assume that the Pending Analysis simply stores the computed values (as suggested in [19]) and whenever an application is revisited this stored value is used. Now, suppose that we first visit the left branch of the diamond, and through the upgoing edge from (I) visit the application (*) a second time, which is then assumed to have the value $\bot$. Then all applications in (I) will be evaluated under the assumption that this is the value of (*), but having visited (I), suppose we finally visit (II) and discover that the value of (*) should have been something bigger than $\bot$. Now, if, as suggested for Pending Analysis, the call in the right branch simply reuses all the values computed for (I) we will end up with a result which is potentially too small! (This is a disaster for strictness analysis — the value will be 'unsafe'.) Surely, the fix is to recompute the values in (I) reflecting the change of (*), which is precisely what our algorithm is doing, moreover it does it in a very minimal fashion by chasing explicit dependencies.

Let us return to a concrete example, the function `cat` (for 'concatenate') defined in the following program:

```
foldr(f,[],a)     = a
foldr(f,h :: t,a) = f(h,foldr(f,t,a))
append(l,m)       = foldr(cons,l,m)
cat(l)            = foldr(append,l,nil)
```

with the types

```
cons   : α × α list → α list
foldr  : (α × γ → γ) × α list × γ → γ
append : α list × α list → α list
cat    : α list list → α list
```

Let $\mathbf{2} = \mathbf{O} = \{0,1\}$, $\mathbf{4} = \{0,1,2,3\}$ with $0 < 1 < 2 < 3$ likewise for $\mathbf{6}$. Then Wadler's analysis [17] suggests the following abstract types, when **cat** is to be instanciated to

`c list list` for some ground type c.[1]

$$\begin{array}{ll} \texttt{cons} & : \mathbf{2} \times \mathbf{4} \to \mathbf{4} \\ \texttt{foldr} & : (\mathbf{4} \times \mathbf{4} \to \mathbf{4}) \times \mathbf{6} \times \mathbf{4} \to \mathbf{4} \\ \texttt{append} & : \mathbf{4} \times \mathbf{4} \to \mathbf{4} \\ \texttt{cat} & : \mathbf{6} \to \mathbf{4} \end{array}$$

The size of $(\mathbf{4}\times\mathbf{4} \to \mathbf{4})\times\mathbf{6}\times\mathbf{4}$ is around 10^{11}, so hopefully we do not need to evaluate `foldr` on all its arguments! Actually, in computing `foldr` at one argument the local algorithm visits at most 24 variables. This case is particularly simple, as the same function is used for the argument of `foldr` in any recursive call, but the general point remains the same. If we look at the functional defining `foldr` this is actually defined on a lattice of $4^{10^{11}}$ elements with height $3*10^{11}$, so any attempt of iterating from the bottom inside this huge lattice can be fatal.

4 Example: Model checking

To explain this example we need to briefly introduce the problem of model checking in the modal μ-calculus (the full details on the construction and proofs of the claims can be found in [2,1]). The problem we want to solve is do decide whether a particular state s of a *labelled transition system*, a triple $T = (S, L, \to)$, where S is a set of *states*, L a set of *labels*, and $\to \subseteq S \times L \times S$ is a transition relation, satisfies an assertion A in the modal μ-calculus, i.e. to decide

$$s \models A$$

with respect to a proper definition of $\models$. The syntax of assertions is defined by

$$A ::= F \mid T \mid A_0 \vee A_1 \mid A_0 \wedge A_1 \mid \langle\alpha\rangle A \mid [\alpha]A \mid X \mid \mu X.A \mid \nu X.A$$

Without going into all the details we just state the result that when considering assertions with one fixed-point the problem of satisfaction can be transformed to a set of monotonic equations with a variable ranging over $\mathbf{O} = \{0,1\}$ for each pair of state of T and subassertion of A, yielding a system with $|A||S|$ equations, with the total size of the right-hand sides, all consisting of finite conjunctions and disjunctions, equal to $|A||T|$. Finite conjunctions and disjunctions are easily implemented efficiently by keeping track of the number of sons with marking 1, hence any change can be propagated in constant time.

As all the cpo's have height one the algorithm ends up running in worst-case time $O(|A||T|\log(|A||S|))$, which improves on the local algorithms of Larsen [9], Stirling and Walker [15], Winskel [18], and Cleaveland [3], and also the global algorithm of Emerson and Lei [6].

[1] Actually `foldr` is needed in two versions corresponding to the two different applications of `foldr`, but the more general of the two has enough information to deduce the strictness information for the other.

(Historically, this special application of the algorithm was discovered first, the present generalization is a clarified, rational reconstruction of the fundamental mechanisms of the algorithm of [2,1].)

5 Example: Constraint systems

Recently, it has become popular to solve various type checking, type inference, and other program analysis related problems, by constructing a set of constraints to be solved. We will show how one of these problems can be solved by the local method with optimal complexity (up to the logarithmic factor).

The particular constraint system we consider is due to Palsberg and Schwartzbach [14], used in performing what they call *safety analysis* – a version of *closure analysis* – but very similar sets of constraints have been used for type inference [13]. For each subterm of the program we will have a variable, which is going to hold information about that particular part of the program (whether this is type information or anything else is irrelevant). In this particular situation, we will think of the information of interest as subsets of what we will call tokens. The set of subsets of tokens is a finite lattice $\mathcal{P}(S)$ ordered by inclusion and S being the set of tokens. Now, the constraint system can be formulated as consisting of a set of conditional and unconditional inequalities

$$c \subseteq x, \quad r \Rightarrow y \subseteq x, \quad x \subseteq c$$

where x, y are token set variables, c a constant token set, and r is a boolean constant defined as $r = e$ where e is an expression built from disjunctions and conjunctions of other boolean constants and the inequalities $c \subseteq x$. Denote by $\mathcal{C}$ the complete set of inequalities and boolean constant definitions. We build a set of monotonic equations with

1. a variable v_x of type $\mathcal{P}(S)$ for each of the token set variables x,
2. a variable b_r of type $\mathbf{O}$ for each of the boolean constants, and
3. a variable $b_{c\subseteq x}/b_{x\subseteq c}$ of type $\mathbf{O}$ for each constraint $c \subseteq x / x \subseteq c$.

We will make use of some auxiliary functions: Let $\to$ be the conditional of type $\mathbf{O} \times \mathcal{P}(S) \to \mathcal{P}(S)$ with an obvious definition (using 1 to represent true), $(c \subseteq) : \mathcal{P}(S) \to \mathbf{O}$ has just as obvious a definition, and finally $\mathbf{viol}(\subseteq c) : \mathcal{P}(S) \to \mathbf{O}$ is defined on $u \in \mathcal{P}(S)$ as the negation of $u \subseteq c$ (to make it monotonic in u).

The equations associated with the variables are now as follows:

$$\begin{array}{lll} v_x & = & (\bigcup_{r\Rightarrow y\subseteq x \,\in\, c} b_r \to y) \cup (\bigcup_{c\subseteq x \in c} c) \\ b_r & = & b_e \qquad\qquad\qquad\qquad\qquad\qquad r = e \in \mathcal{C} \\ & & \text{where } b_e \text{ is (recursively) defined by} \\ & & \qquad b_e = \begin{cases} \bigwedge_{i\in I} b_{e_i} & \text{if } e = \bigwedge_{i\in I} e_i \\ \bigvee_{i\in I} b_{e_i} & \text{if } e = \bigvee_{i\in I} e_i \\ b_{r'} & \text{if } e = r' \end{cases} \\ b_{c\subseteq x} & = & (c \subseteq) r_x \\ b_{x\subseteq c} & = & \mathbf{viol}(\subseteq c) r_x \end{array}$$

The right-hand sides are all monotonic, and we can easily give ?-nice extensions of the functions involved. For $\bigwedge$ and $\bigvee$ we use the straightforward extension of the binary case from example 1 with the efficient implementation discussed in the model checking example. For $\bigcup : \mathcal{P}(S)^k \to \mathcal{P}(S)$ we take

$$\bigcup(u_1, \ldots, u_k) = \begin{cases} S & \text{if } \exists i.u_i = S \\ ? & \text{if } \forall i.u_i \neq S \ \&\ \exists i.u_i = ? \\ u_1 \cup \ldots \cup u_k & \text{otherwise} \end{cases}$$

The rest are as follows:

if	?	$x \neq ?$
?	?	?
0	$\bot_D$	$\bot_D$
1	?	x

	$(c \subseteq)$
?	?
x	$\begin{cases} 0 & \text{if } c \not\subseteq x \\ 1 & \text{if } c \subseteq x \end{cases}$

	viol$(\subseteq c)$
?	?
x	$\begin{cases} 0 & \text{if } x \subseteq x \\ 1 & \text{if } x \not\subseteq c \end{cases}$

Now, it is not hard to see, that the set of inequalities $\mathcal{C}$ has a solution, if and only if, the monotonic equation system has a minimum solution in which all the variables $v_{x \subseteq c}$, corresponding to what Palsberg and Schwartzbach calls 'safety constraints', are 0, thus not being violated. The algorithm they suggest has running time $O(n^3)$, n being the size of the program and the number of tokens. They argue that the number of constraints will be $O(n^2)$.

A straightforward implementation of the union operator and the tests for inclusion gives us a local algorithm with the following running time

$$\begin{aligned} & \textstyle\sum_{v \in V}(c(f_v) \sum_{u \in S(v)}(ht(D_u) + 1)) \log n \\ = \ & \textstyle\sum_{v_x}(a_{v_x} n a_{v_x}(n+1)) \log n + \textit{`cheaper stuff'} \\ & \quad a_{v_x} \text{ being the arity of the right-hand side of } v_x \\ = \ & n^2 \log n (\textstyle\sum_{v_x} a_{v_x}^2) \\ \in \ & O(n^4 \log n) \end{aligned}$$

The way to improve on this shows as a general point how the general algorithm can be used as the backbone of more specific and efficient algorithms, while maintaining the overall structure. Here, the expensive part is the repeated computation of unions of sets, the size of which is bounded by n. But the only thing that happens in the algorithm is 'small' changes in the arguments, so by altering the way changes are propagated we will improve the bound to $O(n^3 \log n)$ (and Kildall's algorithm achieves $O(n^3)$, through the same construction).

To each variable we associate a bitvector of length n, the i'th coordinate indicating whether token number i belongs to the set or not. Then, when a variable changes marking and we add the parents to the active set, we will also propagate the *actual change* as a list of tokens, which can then be incorporated in time proportional to the size of the change. In this manner, the *amortized* cost of computing each of the union operations will be bounded by the arity multiplied with $n+1$ (the height of the lattices of the sons).

Similarly, the inclusion tests $(c \subseteq)$ and $(\subseteq c)$ can be implemented with amortized cost $O(n)$. The total running time is now bounded by

$$(\textit{total cost for } v_x\textit{'s} + \textit{total cost for } b_r\textit{'s} + \textit{total cost for } b_{c \subseteq x}/b_{x \subseteq c}\textit{'s}) \log n$$

which is

$$O((n^2 + n^3 + n^3) \log n) = O(n^3 \log n).$$

6 Related Work

The aims of minimizing the number of variables of the equation system investigated when finding a partial minimum solution is shared with the aims of Cousot and Cousot in their "chaotic fixed-point iteration" [4, sec. 4.2.1] and in the refined denotational semantics known as "minimal function graphs" (Jones and Mycroft [7]). However, whereas these papers describe general schemes for computing partial minimum solutions, they are very brief on the subject of when functions "need" the values of other functions applied to specific arguments, and how to incorporate that into an algorithm. Jones and Mycroft leave out this decisions, their description is parametrised by such proper choices, and Cousot and Cousot seems to indicate a mere *syntactic* criterion (corresponding roughly to the part of our algorithm performing syntactic dependency analysis). Contrary to this, we formalise the dependency as the notion of ?-*nice extensions* and show how this together with the explicit presence of a graph representing the semantic dependencies, allowing for efficient sharing and updating of values, making, even in the worst-case, this local method almost[2] as efficient as the global method — which in turn has bad average-case behaviour.

Similarly, the aims of fast fixed-point finding of functionals illustrated in our example on strictness analysis, is shared by the work of Nielson and Nielson [12,11]. Their approach is, however, somewhat orthogonal to the algorithms described here. They focus on classifying functions subject to the number of steps needed in computing the fixed-point iteratively and on finding classes for which equality tests of functions can be done, without having to consider by brute-force each element in the domains of the functions. (An example is: if the elements of the approximation sequence can guarenteed to be join-preserving, then the functions need only be compared on the join-irreducible elements (often called atoms) of the poset constituting the domain of the functions.) Especially, as concerns this last analysis minimizing the cost of comparing functions, the present algorithm could benefit from their results when applied to higher-order cases.

7 Conclusion

We have introduced a local fixed-point finder, and shown how it can be used for solving three problems of general interest. The pattern we have used is to reformulate the problems to problems of finding minimum solutions to sets of monotonic equations, and by

[2] "The log-factor".

tailoring the local algorithm achieve an efficient solution to the problem. We expect that this approach can be used on a variety of cases.

An interesting aspect which has not been investigated in this paper, is the potential speed-up coming from decomposing the value domains to smaller domains and thereby adding new variables. A notable example of the success of such an approach is the model checking problem, where the original problem is to compute a fixed-point of a function $f = \lambda X.A$ on a lattice $\mathcal{P}(S)$. This lattice is decomposed to the lattice $\mathbf{O}^{|S|}$ and the corresponding systems has $|S|$ variables with a total size of the right-hand sides of $|A||T|$ (T is the labelled transition system), thereby yielding a significant speed-up (from $|A||T|^2$ to $|A||T|$), which actually corresponds to computing just one approximation to the fixed-point, i.e. one application of f.

The connection to the work from data-flow analysis is intriguing and should be further investigated.

8 Acknowledgements

At various points in the work towards this paper, I have had useful discussions with Chris Hankin, Sebastian Hunt, John Launchbury, Hanne Riis Nielson, Flemming Nielson, Jens Palsberg, Mads Rosendahl, Michael Schwartzbach, Glynn Winskel and others, and last but not least Fritz Henglein who pointed me to Kildall's algorithm. Thanks are also due to Urban Engberg for correcting my english.

A Proof of lemma 2

First a useful proposition:

Proposition 5 *Let D be a cpo with bottom, and f an ω-continuous function on D. For every $x \in D$ with*

$$x \leq f(x) \text{ and } x \leq \mu f,$$

the chain $\{f^i(x)\}_{i\in\omega}$ is increasing and

$$\bigsqcup_{i\in\omega} f^i(x) = \mu f.$$

Proof: Easy. □

Lemma 6 *Suppose D and E are cpo's with bottoms. Let (j, p) be an embedding-projection pair between D and E, i.e. $j : D \to E$ and $p : E \to D$ are monotonic functions with*

$$(i)\ p \circ j = id_D \qquad (ii)\ j \circ p \leq id_E,$$

where $\leq$ is the pointwise extension of the ordering of E to functions $E \to E$. Then p is a right adjoint to j, i.e. for all $x \in D, y \in E$:

$$j(x) \leq_E y \ \Leftrightarrow\ x \leq_D p(y). \qquad (5)$$

Proof: $\Rightarrow$:

$$\begin{array}{lll} j(x) \leq_E y & \Rightarrow & p(j(x)) \leq_D p(y) \quad \text{by monotonicity of } p \\ & \Rightarrow & x \leq_D p(y) \quad \text{by } (i) \end{array}$$

$\Leftarrow$:

$$\begin{array}{lll} x \leq_D p(y) & \Rightarrow & j(x) \leq_E j(p(y)) \quad \text{by monotonicity of } j \\ & \Rightarrow & j(x) \leq_E j(p(y)) \leq_E y \quad \text{by } (ii) \end{array}$$

□

Proof (Projection lemma.): We first argue that $p^{-1}(y)$ is a cpo with bottom. Let $z_0 \leq z_1 \leq z_2 \leq \ldots$ be an increasing chain in $p^{-1}(y)$. Then by ω-continuity of p,

$$p(\bigsqcup z_i) = \bigsqcup p(z_i) = y,$$

hence $\bigsqcup z_i \in p^{-1}(y)$. Moreover, for all $x \in p^{-1}(y)$, i.e. $y = p(x)$, we have $y \leq p(x)$ and therefore $j(y) \leq x$ by lemma 6 equation (5). In other words, $j(y)$ is a bottom element of $p^{-1}(y)$.

Now, from (i) it follows that f restrict to an ω-continuous function on $p^{-1}(y)$, and that $\{(f_{|p^{-1}(y)})^i(j(y))\}_{i\in\omega}$ is an increasing chain in $p^{-1}(y)$ hence $f_{|p^{-1}(y)}$ has a minimum fixed-point

$$z = \bigsqcup_{i\in\omega}(f_{|p^{-1}(y)})^i(j(y)) = \bigsqcup_{i\in\omega} f^i(j(y)).$$

From lemma 6 using assumption (ii) we get $j(y) \leq \mu x.f(x)$ and using (i) we get $j(y) \leq f(j(y))$ and hence by proposition 5,

$$z = \bigsqcup_{i\in\omega} f^i(j(y)) = \mu x.f(x)$$

from which it follows that

$$y = p(z) = p(\mu x.f(x)).$$

□

References

[1] Henrik Reif Andersen. Local computation of alternating fixed-points. Technical Report No. 260, Computer Laboratory, University of Cambridge, June 1992.

[2] Henrik Reif Andersen. Model checking and boolean graphs (extended abstract). In B. Krieg-Brückner, editor, *Proceedings of 4'th European Symposium on Programming, ESOP'92, Rennes, France*, volume 582 of *LNCS*. Springer-Verlag, 1992.

[3] Rance Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27:725–747, 1990.

[4] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of recursive procedures. In Erich J. Neuhold, editor, *Formal Description of Programming Concepts*, pages 237–277. North-Holland, 1978. Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts, St. Andrews, N.B., Canada, August 1–5, 1977.

[5] Alan 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] E. Allen Emerson and Chin-Luang Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Symposium on Logic in Computer Science, Proceedings*, pages 267–278. IEEE, 1986.

[7] Neil D. Jones and Alan Mycroft. Data flow analysis of applicative programs using minimal function graphs: Abridged version. In *Proceedings of 13th Annual ACM Symp. on Principles of Programming Languages*, 1986.

[8] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of POPL'73*, pages 194–206. ACM, 1973.

[9] Kim G. Larsen. Proof systems for Hennessy-Milner logic with recursion. In M. Dauchet and M. Nivat, editors, *Proceedings of CAAP, Nancy, Franch*, volume 299 of *Lecture Notes in Computer Science*, pages 215–230, March 1988.

[10] Alan Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, December 1981.

[11] Flemming Nielson and Hanne Riis Nielson. Finiteness conditions for fixed point iteration. Manuscript. To appear, November 1991.

[12] Hanne Riis Nielson and Flemming Nielson. Bounded fixed point iteration. In *Proceedings of the 18'th Annual Symposium on Principles of Programming Languages, POPL*, 1991. Also as DAIMI PB-359, Aarhus University, July 1991.

[13] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *Proc. OOPSLA'91, ACM SIGPLAN Sixth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 146–161, Phoenix, Arizona, October 1991.

[14] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference. Technical Report PB-389, Computer Science Department, Aarhus University, 1992. Submitted for publication.

[15] Colin Stirling and David Walker. Local model checking in the modal mu-calculus. In J. Díaz and F. Orejas, editors, *Proceedings of TAPSOFT, Barcelona, Spain*, volume 351 of *Lecture Notes in Computer Science*, pages 369–383, March 1989.

[16] R. Tarjan. Depth-first search and linear graph algorihtms. *SIAM J. Comput.*, 2(1), June 1972.

[17] Philip Wadler. Strictness analysis on non-flat domains. In Samson Abramsky and Chris Hankin, editors, *Abstract interpretation of declarative languages*, chapter 12. Ellis Horwood, 1987.

[18] Glynn Winskel. A note on model checking the modal ν-calculus. In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *Proceedings of ICALP*, volume 372 of *LNCS*, pages 761–772, 1989.

[19] Jonathan Young and Paul Hudak. Finding fixpoints on function spaces. Technical Report YALEU/DCS/RR-505, Yale University, Department of Computer Science, December 1986.