FUNCTIONAL PEARL
Lazy Least Fixed Points in ML

FRANÇOIS POTTIER*
INRIA Paris-Rocquencourt
B.P. 105
78153 Le Chesnay Cedex, France

1 Introduction

In this paper, we present an algorithm for computing the least solution of a system of monotone equations. This algorithm can be viewed as an effective form of the following well-known fixed point theorem:

Theorem *Let $\mathcal{V}$ be a finite set of* variables*. Let $(\mathcal{P}, \leq, \bot)$ be a partially ordered set of* properties *with a least element and with finite height. Let $\mathcal{E} : \mathcal{V} \to (\mathcal{V} \to \mathcal{P}) \to \mathcal{P}$ be a system of monotone equations; that is, for every $v \in \mathcal{V}$, let $\mathcal{E}(v)$ be a monotone function of $\mathcal{V} \to \mathcal{P}$, ordered pointwise, into $\mathcal{P}$. Then, there exists a least valuation $\phi : \mathcal{V} \to \mathcal{P}$ such that, for every $v \in \mathcal{V}$, $\phi(v) = \mathcal{E}(v)(\phi)$.* ◇

We refer to the function $\mathcal{E}$ as a system of equations because to every left-hand side—a variable—it associates a right-hand side—something that can be evaluated to a property, provided every variable is assigned some property.

Why might such an algorithm be of interest? Broadly speaking, it is useful in the analysis of cyclic structures: grammars, control flow graphs, transition systems, etc. For instance, it can be used to determine which non-terminal symbols of a context-free grammar produce the empty word, and to compute "first" and "follow" sets. More generally, least fixed point computations are required in data flow analysis (Kildall, 1973; Kam & Ullman, 1976), abstract interpretation (Cousot & Cousot, 1977), and model checking of temporal logic formulae (Liu & Smolka, 1998).

The simplest algorithm, known as Kleene iteration, computes a sequence of valuations, defined by $\phi_0 = \lambda v.\bot$ and $\phi_{n+1} = \lambda v.\mathcal{E}(v)(\phi_n)$. That is, at each iteration, the right-hand side of every equation is re-evaluated.

Of course, Kleene iteration is inefficient. A more natural idea is to re-evaluate a single right-hand side at a time, and, at each step, to appropriately decide which right-hand side to re-evaluate. In particular, one should re-evaluate the right-hand side associated with a variable v_2 only if the property associated with some variable v_1, where v_2 *depends* on v_1, has changed since v_2 was last examined. This scheme, known as chaotic iteration, appears in many papers; one of the earliest sources is perhaps Kildall (1973).

* (*e-mail:* Francois.Pottier@inria.fr)

But what does it mean for v_2 to depend on v_1? When the right-hand sides are just syntax, one may consider that v_2 depends on v_1 if the right-hand side $\mathcal{E}(v_2)$ contains an occurrence of v_1. When the right-hand sides are functions in $(\mathcal{V} \to \mathcal{P}) \to \mathcal{P}$, as is the case here, this amounts to considering that v_2 depends on v_1 if, for some valuation ϕ, the evaluation of $\mathcal{E}(v_2)(\phi)$ requires the evaluation of $\phi(v_1)$. This relation is known as *static dependency*, because its graph can be built ahead of time, before the iterative computation begins.

A more interesting, and perhaps surprising, approach is to consider that v_2 depends on v_1 if the last evaluation of $\mathcal{E}(v_2)(\phi)$ required the evaluation of $\phi(v_1)$, where ϕ was the then-current valuation. This relation is known as *dynamic dependency*, because its graph evolves as the iterative computation progresses. When the right-hand sides are functions, dynamic dependencies can be discovered by observing the behavior of these functions, that is, by observing how they interact with their argument ϕ.

In this paper, we present an algorithm—an on-demand version of chaotic iteration with dynamic dependencies—with a simple interface, good asymptotic complexity, and a 180 line implementation (Pottier, 2009a).

Fixed point computation algorithms seem to be often re-invented and re-implemented in ad hoc ways. The author believes that a modular, re-useable version of such an algorithm can be useful in many situations. It should help programmers focus on constructing a concise problem statement, in the form of a system of equations, and avoid the pitfall of mixing problem statement and iteration logic.

Our algorithm is implemented in imperative OCAML code, but presents a purely functional, higher-order interface. How and why encapsulation is properly achieved is somewhat subtle, and would deserve formal verification. The author offers this as a challenge to researchers interested in modular formal proofs of ML programs.

2 Interface

The fixed point computation algorithm is packaged as an OCAML module, *Fix*. This module defines a functor, *Fix.Make*, which expects two parameters: an implementation of maps over variables and a partially ordered set of properties. The functor produces a function *lfp* that accepts a system of monotone equations and produces its least solution.

2.1 Maps

A signature for imperative maps appears in Figure 1. A module that implements this signature must supply a fixed type of keys, *key*; a type of maps, *'data t*, which is parametric in the type of the data that is stored in the maps; and operations to create a fresh map in an empty state (*create*), revert an existing map to an empty state (*clear*), add an entry to a map (*add*), look up a key in a map (*find*), and iterate over all entries in a map (*iter*). The function *find* is expected to raise the exception *Not_found*, defined in OCAML's standard library, if the key is absent.

Imperative maps are mutable data structures: *create* produces a fresh map; *clear*

```
module type IMPERATIVE_MAPS = sig
  type key
  type 'data t
  val create: unit → 'data t
  val clear: 'data t → unit
  val add: key → 'data → 'data t → unit
  val find: key → 'data t → 'data
  val iter: (key → 'data → unit) → 'data t → unit
end
```

Fig. 1. A signature for imperative maps

```
module type PROPERTY = sig
  type property
  val bottom: property
  val equal: property → property → bool
end
```

Fig. 2. A signature for properties

```
module Make
  (M : IMPERATIVE_MAPS)
  (P : PROPERTY)
  : sig
    type variable = M.key
    type property = P.property
    type valuation = variable → property
    type rhs = valuation → property
    type equations = variable → rhs
    val lfp: equations → valuation
  end
```

Fig. 3. The algorithm's interface

and *add* modify an existing map. Imperative maps are a weaker requirement than persistent maps: an imperative map is easily obtained by wrapping a persistent map in a reference cell. Thus, the client is free to choose between an efficient imperative implementation, such as arrays and hash tables, and a persistent implementation, such as the balanced binary trees offered by OCAML's standard library module *Map*.

2.2 Properties

A signature for properties appears in Figure 2. Properties must be equipped with a partial order, that is, a reflexive, antisymmetric, transitive relation, written $\leq$. There must be a least property, *bottom*. Perhaps surprisingly, an implementation of the ordering is not required: we just need equality, *equal*. *bottom* is used as the initial element of an ascending Kleene chain, while *equal* is used to detect stabilization.

2.3 The functor Make

The functor *Make* (Figure 3) is parameterized with an implementation M of maps over variables and an implementation P of properties. Its result signature begins

with a number of auxiliary type definitions. In particular, a valuation is a mapping of variables to properties; a right-hand side is a (monotone) mapping of valuations to properties; a system of equations is a mapping of variables to right-hand sides. Then comes the main function, *lfp*, which maps a system of equations to its least fixed point, a valuation. The system of equations, which is named *eqs* in the following, has type *equations*, that is, *variable* → (*variable* → *property*) → *property*. This corresponds to our earlier notation $\mathcal{V} \to (\mathcal{V} \to \mathcal{P}) \to \mathcal{P}$.

It is guaranteed that the application *eqs* *v* is performed at most once per variable v. This enables the client to perform expensive pre-computation, or to allocate an auxiliary data structure, at this time. Without this guarantee, there could arise a need for the client to maintain an explicit map of variables to auxiliary data. Here, this mechanism is offered transparently. This staging opportunity is the reason why we define *equations* as *variable* → (*variable* → *property*) → *property*, as opposed to the arguably more standard (*variable* → *property*) → (*variable* → *property*).

The functor *Make* produces the least solution of the system of equations. This takes the form of a function, henceforth referred to as *get*, of type *variable* → *property*. The algorithm is lazy: no actual computation takes place when *Make* is applied. The fixed point computation takes place, on demand, when *get* is applied. Memoization is performed, so that two invocations of *get* at a single variable v, or at two variables v_1 and v_2 that are related in the dependency graph, do not cause repeated work.

2.4 Specification

Since we offer this code as a proof challenge, it is worth devoting some space to its specification. We do not wish to impose the use of a specific methodology, so the specification that we sketch remains informal. We do have in mind a plausible approach, namely the use of a program logic in the style of Hoare, under a partial correctness interpretation. Tools that follow this approach include Pangolin (Régis-Gianas & Pottier, 2008) and Ynot (Chlipala *et al.*, 2009).

We assume that we know what it means for an OCAML value of type *variable* to *implement* a mathematical object $v \in \mathcal{V}$, and for an OCAML value of type *property* to implement a mathematical object $p \in \mathcal{P}$. In a formal treatment, these predicates would be abstract: that is, they would be parameters of the functor *Make*.

Let P be a mathematical predicate of one argument. An OCAML expression e is said to *satisfy* the post-condition P if the following conditions are met:

1. the execution of e does not have any side effect, such as mutating a reference, or throwing a catchable exception; (however, the execution of e may diverge, and may cause a fatal failure by executing the instruction **assert** *false*;)
2. if the execution of e successfully completes, then the OCAML value that is produced implements some mathematical object x such that $P\ x$ holds.

An expression e is said to *compute* x if it satisfies the post-condition $\lambda y.(y = x)$. This is the partial correctness interpretation of a post-condition.

We now use these basic notions in a straightforward way to define higher-order versions of the "implements" predicate. These definitions form the specification.

1. An OCAML function *request* of type *valuation* implements $\phi \in \mathcal{V} \to \mathcal{P}$ if, for every $w \in \mathcal{V}$ and for every OCAML value v that implements w, the OCAML expression *request* v computes $\phi(w)$.
2. An OCAML function *eqs* of type *equations* implements $\mathcal{E} \in \mathcal{V} \to (\mathcal{V} \to \mathcal{P}) \to \mathcal{P}$ if, for every $w \in \mathcal{V}$ and $\phi \in \mathcal{V} \to \mathcal{P}$, for all OCAML values v and *request* that respectively implement w and ϕ, the OCAML expression *eqs* v *request* computes $\mathcal{E}(w)(\phi)$.
3. An OCAML function *lfp* of type *equations* → *valuation* implements a least fixed point algorithm if, for every system of monotone equations $\mathcal{E}$ and for every OCAML function *eqs* that implements $\mathcal{E}$, the OCAML expression *lfp eqs* computes an OCAML function, say *get*, such that *get* implements the least solution of $\mathcal{E}$.

This is a purely functional specification, in the sense that the function that the client must provide (*eqs*) is requested to have no side effect, and the functions that we provide (*request*, *lfp*, and *get*) claim to have no side effect.

The challenge is to prove that the version of *lfp* presented in this paper does implement a least fixed point algorithm in the above sense, or (to put it slightly differently) that it is sound for the client to believe so. We discuss the problem further in §3.7 and §4.1.

2.5 How the algorithm is used

In order to use the algorithm, it suffices to instantiate the formal parameters M, P, and *eqs*. We illustrate this in Figure 4 by showing actual code for computing the nullable symbols of a context-free grammar. The code is parameterized with an implementation of maps over nonterminal symbols and with a description of the grammar itself. The property space is *Boolean*, a 5-line module (not shown) whose signature is *PROPERTY* **with type** *property* = *bool*. The code seems as concise as one could hope.

3 Implementation

The algorithm's implementation relies on a number of mutable auxiliary data structures: a dependency graph; a workset; a couple of tables that respectively hold permanent and transient data. We briefly describe these data structures, then move on to the algorithm's core functions.

Everything that follows is placed within the definition of *Make* and within the definition of *lfp*. This means that we are free to refer to the formal parameters M, P, and *eqs*. This also implies that a distinct instance of the algorithm's mutable state is created afresh at every call to *lfp*.

```
module type GRAMMAR = sig
  type terminal
  type nonterminal
  type production =
    | Epsilon
    | T of terminal
    | N of nonterminal
    | Seq of production × production
    | Alt of production × production
  val productions: nonterminal → production
end

module Analyze
  (M : IMPERATIVE_MAPS)
  (G : GRAMMAR with type nonterminal = M.key)
= struct
  open G
  module F = Fix.Make(M)(Boolean)
  let nullable : nonterminal → bool =
    F.lfp (fun nt request →
      let rec nullable = function
        | Epsilon → true
        | T _ → false
        | N nt → request nt
        | Seq (prod1, prod2) → nullable prod1 ∧ nullable prod2
        | Alt (prod1, prod2) → nullable prod1 ∨ nullable prod2
      in nullable (productions nt)
    )
end
```

Fig. 4. Finding which symbols in a context-free grammar are nullable

3.1 The dynamic dependency graph

We maintain a directed graph whose edges represent dependency information: an edge $v_2 \rightarrow v_1$ means that v_2 depends on v_1. We also say that v_2 *observes* v_1, or that v_1 is a *subject* of v_2. A signal travels along this edge, from subject v_1 to observer v_2, when the value associated with v_1 changes. A variable may observe itself.

The graph is dynamic. When the algorithm is *inactive* (that is, outside an invocation of *get*), the graph is empty. During a *run* of the algorithm (that is, during an invocation of *get*), the set of its nodes grows with time, as new variables are discovered (the set of all variables is not known in advance), while the set of its edges changes with time: an edge $v_2 \rightarrow v_1$ might appear at some point in time, and disappear at some later point.

For increased modularity, the signature (Figure 5) and implementation of the graph data structure are stand-alone. They could, if desired, be placed in a separate module, with no reference to *Fix*. The type of the graph nodes, *'data node*, is parametric in a type variable *'data*. Each graph node carries a piece of data, which is provided when the node is created (*create*), and can be looked up at any time (*data*).

Three functions are provided to inspect and modify edges. *predecessors n* pro-

```
module Graph : sig
  type 'data node
  val create: 'data → 'data node
  val data: 'data node → 'data
  val predecessors: 'data node → 'data node list
  val set_successors: 'data node → 'data node list → unit
  val clear_successors: 'data node → unit
end
```

Fig. 5. A signature for the dependency graph

```
type node =
    data Graph.node

and data =
  { rhs: rhs; mutable property: property }

let property node =
  (Graph.data node).property
```

Fig. 6. The data carried by each graph node

duces a list of the predecessors of the node n. *set_successors* n *ns* assumes that the node n initially has no successors, and creates an edge from n to each of the nodes in the list *ns*. Only one edge per destination node is created, even if *ns* has duplicate elements. *clear_successors* n removes all of the edges that leave node n.

Implementing this signature is a simple exercise in imperative programming. The code is not shown: it can be found online (Pottier, 2009a).

3.2 Node data

Each graph node stands for a variable v, and carries information about this variable. This is shown in Figure 6. The field *rhs* stores the result of the application *eqs* v, so that we can hold our promise of performing this application at most once. The mutable field *property* holds the current property, a lower approximation of the fixed point, at v. During a run, its value increases with respect to $\leq$. The function *property* provides access to this field.

3.3 The workset

The workset is a data structure where elements are inserted and retrieved in an arbitrary order. The elements of the workset are graph nodes. A node never occurs twice in the workset. Our implementation is based on OCAML's standard library module *Queue*, which provides FIFO ordering. One could just as well use *Stack* instead, which provides LIFO ordering, or a priority queue, with heuristic priorities (Demers *et al.*, 1987).

The workset offers just two functions (Figure 7). *insert* n inserts the node n into the workset. *repeat* f repeatedly applies the function f to a node extracted out of

```
module Workset : sig
  val insert: node → unit
  val repeat: (node → unit) → unit
end
```

Fig. 7. A signature for the workset

```
let signal subject =
  List.iter (fun observer →
    Graph.clear_successors observer;
    Workset.insert observer
  ) (Graph.predecessors subject)
```

Fig. 8. Emitting a signal

```
let permanent : property M.t =
  M.create()

let transient : node M.t =
  M.create()

let freeze () =
  M.iter (fun v node →
    M.add v (property node) permanent
  ) transient;
  M.clear transient
```

Fig. 9. Tables

the workset, until the workset becomes empty. The function f is allowed to use *insert*.

When the algorithm is inactive, the workset is empty.

3.4 Subjects, observers, signals

We can now explain the subject-observer mechanism in greater detail. A node is either *awake* or *asleep*. It is awake when it is scheduled for examination, that is, when it is in the workset. In that case, it is never an observer: that is, it has no successors in the graph. It is asleep when it is not in the workset. In that case, it observes a number of nodes, its subjects, and is awoken when one of them emits a signal.

A signal emitted by some subject is broadcast to all of its observers (Figure 8). By the above invariant, these observers must be asleep, that is, out of the workset. Each of them is awoken, in turn, which means that it stops observing (it loses all of its own successors) and is inserted into the workset.

3.5 Tables

The algorithm maintains a couple of tables (Figure 9). Both are maps whose keys are variables. They have disjoint domains.

The *permanent table* maps variables to properties. It is a fragment of the least solution of the system of equations. It persists across runs. It is initially empty, and grows forever. It is used to implement memoization.

The *transient table* maps variables to graph nodes. It is used only within a run: when the algorithm is inactive, it is empty. During a run, it fills up with new nodes, which represent variables that may not yet have reached a fixed point. At the end of a run, every variable is known to have stabilized, so the properties contained in the transient table are copied into the permanent table, and the transient table is cleared. This task is performed by *freeze*.

3.6 The core algorithm

The core of the algorithm is in the functions *solve* and *node_for* (Figure 10). A call to *solve node* evaluates the right-hand side at *node*. If this leads to a change, then the current property is updated, and *node* emits a signal towards its observers. The auxiliary function *node_for v* returns a node associated with the variable v, assuming that v does not appear in the permanent table.

When *solve node* is invoked, *node* is not in the workset, and has no subject. In order to re-evaluate the right-hand side at *node*, we invoke the client function *data.rhs*, where *data* is the data record associated with *node*. We must offer the client read access to the current valuation: this is done by passing *request* as a parameter to *data.rhs*. As far as the client is concerned, *request* is a pure function: to a variable v, it associates a property. It does this, internally, by looking up v in the permanent table, which succeeds if the least solution at v has been computed during an earlier run; and, if that fails, by invoking *node_for* to obtain the node *subject* that stands for v and by returning the current property at that node.

In reality, *request* does more than that: it also *spies* on the client. A call to *request* that succeeds by looking up the permanent table is side-effect free, as one might expect. However, if a call to *request* goes through *node_for* and returns the current property at a certain node *subject*, then the fact that *node* depends on *subject* is logged. This is done by inserting *subject* in the list *subjects*. After *data.rhs* completes, the nodes accumulated in the list *subjects* actually become subjects of *node*, through a call to *set_successors*. The dynamic dependencies obtained in this way are correct by construction: if *data.rhs* satisfies its specification, then its result must be determined by the values that it requested.

The flag *alive* is explained later on (§3.7).

After *data.rhs* returns, *solve* compares the current property with the new property produced by *data.rhs*. If they differ (which means, in fact, that the latter is strictly greater than the former), then the current property is updated, and a signal is sent, so all observers of *node* are scheduled for re-examination.

The auxiliary function *node_for* returns a node for a variable v that does not appear in the permanent table. If a node is already associated with v, it is found in the transient table. Otherwise, v is a newly discovered variable. A new node, holding the property *bottom*, is allocated, and a new entry is made in the transient table.

```
let rec solve (node : node) : unit =
  let data = Graph.data node in
  let alive = ref true
  and subjects = ref [] in
  let request (v : variable) : property =
    assert !alive;
    try
      M.find v permanent
    with Not_found →
      let subject = node_for v in
      subjects := subject :: !subjects;
      property subjectin
  let new_property = data.rhs request in
  alive := false;
  Graph.set_successors node !subjects;
  if not (P.equal data.property new_property) then begin
    data.property ← new_property;
    signal node
  end

and node_for (v : variable) : node =
  try
    M.find v transient
  with Not_found →
    let node = Graph.create { rhs = eqs v; property = P.bottom } in
    M.add v node transient;
    solve node;                          (*or: Workset.insert node *)
    node

let inactive =
  ref true

let get (v : variable) : property =
  try
    M.find v permanent
  with Not_found →
    assert !inactive;
    inactive := false;
    let node = node_for v in
    Workset.repeat solve;
    freeze();
    inactive := true;
    property node
```

Fig. 10. The core algorithm

There is a choice between scheduling a newly created node for later examination, by inserting it into the workset, or examining it immediately, via a recursive call to *solve*. The former option is simpler. The latter may seem conceptually bolder (*solve* calls *data.rhs* calls *request* calls *node_for* calls *solve*...), but is correct as well. Because a newly discovered node has no observers, this call to *solve* does not wake

up any existing nodes, a welcome feature. This option permits a form of eager top-down discovery of new nodes. If the dependency graph happens to be acyclic, the algorithm discovers new nodes top-down, performs computation on the way back up, and runs without ever sending a signal or inserting a node into the workset! In practice, this option can improve efficiency by a constant factor, but introduces a risk of blowing up the implicit runtime stack. Either way, this choice has no impact on the algorithm's worst-case asymptotic complexity.

In order to complete the description of the algorithm, there only remains to explain its unique entry point, *get*. A call to *get* v is answered immediately if v occurs in the permanent table. Otherwise, *node_for* v is invoked, and produces a fresh node, since, at the beginning of a run, the transient table is empty. *node_for* can have the effect of inserting new nodes into the workset, so we repeatedly invoke *solve* until the workset becomes empty. At this point, every node in the transient table must have stabilized, so the transient table is copied into the permanent table and cleared. At this point, the least solution at v is known, and can be returned.

3.7 Encapsulating the state of the algorithm

We now justify the Boolean flags *alive* and *inactive*. These flags do not participate in the computation: their only purpose is to rule out certain behaviors of the client.

The flag *alive* (§3.6) is used to prevent the client from invoking a *request* function after this function has become stale, that is, after *data.rhs* has completed. We set things up so that the assertion **assert** !*alive* fails, at runtime, in such an event. Although such an invocation could seem harmless, it would allow the client to observe that two calls to *request* v can produce distinct results (a phenomenon that occurs if the current valuation is updated between the two calls).

The flag *inactive* prevents a re-entrant call by *data.rhs* to *get*. Such a call would break our internal invariant (for instance, we have assumed that, when *get* is invoked, the workset is empty) and cause the algorithm to behave erratically.

Are these runtime checks useful? From a pragmatic point of view, not very much. They are a safe-guard, but one might argue that the chances for a reasonable client to trigger them are slim.

Are these runtime checks necessary? If one wishes to statically prove the algorithm correct with respect to a purely functional specification, *yes.* In our specification (§2.4), *request* claims to implement a certain valuation ϕ. Thus, it must produce identical results when applied to identical arguments. One might think of amending the specification to express the rule that "*request* must not be invoked after it has become stale", but there seems to be no way of doing so without introducing state into the specification. Similarly, there seems to be no way for a purely functional specification to express the rule that "*get* must not be invoked while a call to *get* is already in progress". Thus, the code must remain provably correct even in the face of a client that violates these unspoken rules.

In summary, we claim that these runtime checks are required in order to correctly encapsulate the internal state of the algorithm, that is, in order to prove the code partially correct with respect to a purely functional specification.

4 Comments

4.1 A formal proof challenge

This is a functional program that is not a functional program (Longley, 1999), in the sense that the code is imperative, but strives to present a purely functional interface. In particular, we have used a "spy function", *request*, in order to collect dynamic dependencies. Longley (1999) studies analogous functions.

We conjecture that the code does satisfy the purely functional specification that we have sketched (§2.4). However, to formally verify this conjecture remains a challenge. The proof must be modular, that is, it must reason about the code in isolation, without knowledge of its clients. It must somehow take advantage of the runtime checks **assert** !*alive* and **assert** !*inactive*, which we claimed are necessary. It must explain why it is sound to pretend that *request* and *get* have no side effect, and—at the same time!—check that *request* has the side effect of collecting a sound set of dynamic dependencies.

At present, the author does not know of a solution to this challenge. Powerful logics for programs that manipulate the heap, such as separation logic (Reynolds, 2002) and Hoare Type Theory (Nanevski *et al.*, 2006; Chlipala *et al.*, 2009), should provide good starting points. The author's work on hidden state (Pottier, 2008; Pottier, 2009b) might provide some further clues.

4.2 Is there a purely functional implementation?

Since the specification of our algorithm is purely functional, one might ask: is it possible to efficiently implement this specification in a purely functional manner, say, in Haskell?

In the degenerate case where the dependency graph is guaranteed to be acyclic, the answer is positive. One creates a collection of mutually recursive thunks, one per variable, and relies on lazy evaluation to automatically determine a topological ordering of the dependency graph. This is known as *data recursion*. It is used, for instance, by Ford (2002) in a packrat parser.

In the general case, we believe that the answer is negative (but would be interested to hear otherwise!). Of course, it is possible to define an ascending Kleene chain as a lazy stream, and to obtain its limit by demanding elements until two consecutive elements are found to be equal. Kashiwagi and Wise (1991) attempt to exploit this idea. However, this approach does not seem to yield optimal time complexity.

4.3 Complexity analysis

Theorem *The worst-case time complexity of the algorithm, amortized over a sequence of runs, is $O(Nhc)$, where N is the sum of the number of variables that have entered the permanent table and the number of edges carried by these variables in a static approximation of the dependency graph, h is the height of the partial order on properties, and c is the cost of evaluating a single right-hand side.* ◇

This claim is identical to earlier published results (Vergauwen *et al.*, 1994; Fecht

& Seidl, 1999). In the interests of brevity, the proof is omitted. The parameter N is bounded by the size of a syntactic representation of the equation system. The statement of the theorem assumes that *M.find* and *M.add* have constant time complexity; if that is not the case, multiply by an extra factor.

4.4 Related work

Vergauwen, Wauman, and Lewi (1994) give a simple, abstract description of a nondeterministic chaotic iteration algorithm, which they then refine by introducing a workset, by collecting dynamic dependencies, and by making the algorithm local. Our code can be viewed as an implementation of their algorithm.

The algorithm presented here is also closely related to Le Charlier and Van Hentenryck's top-down algorithm (1992). A technical difference is that our scheduling is driven by the workset—we use top-down evaluation only as an optional mechanism for discovering new nodes—whereas Le Charlier and Van Hentenryck rely purely on top-down evaluation, without a workset. A perhaps deeper difference, and a problematic aspect of their algorithm, is that they do not achieve proper encapsulation of state: two consecutive calls to their version of *request* with identical arguments can produce different results. We believe that this would make it impossible to formally argue that their code satisfies a purely functional specification.

Fecht and Seidl (1999) recall the algorithms by Vergauwen *et al.* (1994) and Le Charlier and Van Hentenryck (1992), and formulate them in ML style: in particular, they use a "spy function". They design a variant of Vergauwen *et al.*'s algorithm that performs depth-first discovery of new nodes and whose workset is a priority queue.

References

Chlipala, Adam, Malecha, Gregory, Morrisett, Greg, Shinnar, Avraham, & Wisnesky, Ryan. 2009 (Sept.). Effective interactive proofs for higher-order imperative programs. *Pages 79–90 of: ACM International Conference on Functional Programming (ICFP).*

Cousot, Patrick, & Cousot, Radhia. 1977 (Jan.). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Pages 238–252 of: ACM Symposium on Principles of Programming Languages (POPL).*

Demers, Alan, Horwitz, Susan, & Teitelbaum, Tim. (1987). An efficient general algorithm for dataflow analysis. *Acta Informatica*, **24**(6), 679–694.

Fecht, Christian, & Seidl, Helmut. (1999). A faster solver for general systems of equations. *Science of Computer Programming*, **35**(2–3), 137–162.

Ford, Bryan. 2002 (Oct.). Packrat parsing: simple, powerful, lazy, linear time. *Pages 36–47 of: ACM International Conference on Functional Programming (ICFP).*

Kam, John B., & Ullman, Jeffrey D. (1976). Global data flow analysis and iterative algorithms. *Journal of the ACM*, **23**(1), 158–171.

Kashiwagi, Yugo, & Wise, David S. 1991 (Apr.). *Graph algorithms in a lazy functional programming language.* Technical Report 330. Indiana University.

Kildall, Gary A. 1973 (Oct.). A unified approach to global program optimization. *Pages 194–206 of: ACM Symposium on Principles of Programming Languages (POPL).*

Le Charlier, Baudouin, & Van Hentenryck, Pascal. 1992 (May). *A universal top-down fixpoint algorithm*. Technical Report CS-92-25. Brown University.

Liu, Xinxin, & Smolka, Scott A. (1998). Simple linear-time algorithms for minimal fixed points. *Pages 53–66 of: International Colloquium on Automata, Languages and Programming*. Lecture Notes in Computer Science, vol. 1443. Springer.

Longley, John. 1999 (Sept.). When is a functional program not a functional program? *Pages 1–7 of: ACM International Conference on Functional Programming (ICFP)*.

Nanevski, Aleksandar, Morrisett, Greg, & Birkedal, Lars. 2006 (Sept.). Polymorphism and separation in Hoare type theory. *Pages 62–73 of: ACM International Conference on Functional Programming (ICFP)*.

Pottier, François. 2008 (June). Hiding local state in direct style: a higher-order anti-frame rule. *Pages 331–340 of: IEEE Symposium on Logic in Computer Science (LICS)*.

Pottier, François. 2009a (Apr.). *Fix*. See `http://gallium.inria.fr/~fpottier/fix/`.

Pottier, François. 2009b (July). *Generalizing the higher-order frame and anti-frame rules*. Unpublished.

Reynolds, John C. (2002). Separation logic: A logic for shared mutable data structures. *Pages 55–74 of: IEEE Symposium on Logic in Computer Science (LICS)*.

Régis-Gianas, Yann, & Pottier, François. (2008). A Hoare logic for call-by-value functional programs. *Pages 305–335 of: International Conference on Mathematics of Program Construction (MPC)*. Lecture Notes in Computer Science, vol. 5133. Springer.

Vergauwen, Bart, Wauman, J., & Lewi, Johan. (1994). Efficient fixpoint computation. *Pages 314–328 of: Static Analysis Symposium (SAS)*. Lecture Notes in Computer Science, vol. 864. Springer.