Local Higher-Order Fixpoint Iteration

Florian Bruse
School of Electrical Engineering and Computer Science
University of Kassel, Germany

Jörg Kreiker
Institute for Computer Science
University of Applied Sciences Fulda, Germany

Martin Lange
Marco Sälzer
School of Electrical Engineering and Computer Science
University of Kassel, Germany

Local fixpoint iteration describes a technique that restricts fixpoint iteration in function spaces to needed arguments only. It has been studied well for first-order functions in abstract interpretation and also in model checking. Here we consider the problem for least and greatest fixpoints of arbitrary type order. We define an abstract algebra of simply typed higher-order functions with fixpoints that can express fixpoint evaluation problems as they occur routinely in various applications, including program verification. We present an algorithm that realises local fixpoint iteration for such higher-order fixpoints, prove its correctness and study its optimisation potential in the context of several applications.

1 Introduction

Fixpoints are ubiquitous in computer science. They serve to explain the meaning of recursion in programming languages [35], database queries [1], formal languages and automata theory [31]; they are being used as logical quantifiers in descriptive complexity [19] or as specialised operators, for instance in temporal logics [15], etc.

Fixpoints often link a denotational and an algorithmic view onto computational problems, most specifically through Kleene's Fixpoint Iteration Theorem [21]: start with the least, resp. greatest value of the underlying lattice, and then keep applying the function under consideration until the sequence becomes stable. This theorem provides the algorithmic foundations for many applications in which fixpoints play an important role. For instance in model checking, fixpoint operators are used to describe correctness properties [17], and methods based on fixpoint iteration are being used to establish the satisfaction of such properties by models of programs [5]. Fixpoints are used in programming language semantics to explain the meaning of recursive programs. This extends to static analysis methods. For instance in the original formulation of abstract interpretation [13], collecting semantics extends program semantics to powersets of semantic values ordered by subset inclusion [14]. Computing program properties then amounts to solving fixpoint equations over a number of specific (powerset) domains. Fixpoint iteration also provides a standard means for the evaluation of recursive database queries [1].

In many applications, the elements of the lattices in which fixpoints are being sought, are functions themselves. In strictness analyses for functional languages [27, 11] for instance, properties under consideration are sets of functions. Denotational semantics is perhaps the application domain which is most easily seen to need lattices of functions, possibly of higher order, in order to explain the meaning of, for example, functional programs of higher order. Certain infinite-state model checking problems, in particular so-called higher-order model checking [29] are tightly linked to the evaluation of fixpoints in functions spaces as well [22].

J.-F. Raskin and D. Bresolin (Eds.): 11th International Symposium
on Games, Automata, Logics, and Formal Verification (GandALF'20).
EPTCS 326, 2020, pp. 97–113, doi:10.4204/EPTCS.326.7

We are concerned with the problem of finding fixpoints in such a lattice of functions of some higher order. Kleene fixpoint iteration in its pure form can still be employed here, but in many situations it is naïve and inefficient for the following reason. Suppose one is not interested in the entire fixpoint f (which is a function of some type $M \to N$) but only in the value of this f on some particular argument $x \in M$. Naïve fixpoint iteration would start by approximating this function with the least one $f_0 := \_ \mapsto \bot_N$ that maps anything to the least element of the lattice N, and successively compute better approximations $f_1, f_2, \ldots$ until $f_{i+1} = f_i$ for some i. Then it would return $f_i(x)$ which equals $f(x)$.

This procedure has then also computed values $f(y)$ for any $y \in M$. It has been observed that a more efficient approach would be goal-driven and avoid the computation of f on any unnecessary argument. Note that, since f is defined recursively, the value $f(x)$ may depend on *some* but *not all* values $f(y)$ for $y \in M$. The term *neededness analysis* was coined to describe the goal-driven evaluation of fixpoints in function lattices, avoiding the computation of function values on arguments that do not contribute to the computation of the one value of interest.

Neededness analysis has been studied well for lattices of first-order functions as they often arise in abstract interpretation [20]. In groundness analyses for logical programs such as [12], instead of neededness, one rather speaks of a fixpoint computations being *local* [18], when a solver tries to only compute the values of as few variables as possible. Neededness analysis has also been studied in the context of model checking complex program properties which cannot be described in the standard temporal logics of regular expressivity (CTL, μ-calculus) but in extensions using predicate transformers [2]. This can be seen as a notion of *on-the-fly* model checking for fixpoints of order 1. Since "local" is also a synonym for "on-the-fly" in model checking [8], we stick to the term *local* fixpoint iteration here rather than the more cumbersome *neededness analysis*, when referring to a method to avoid the computation of all arguments of fixpoints which are functions themselves.

In this paper we consider the applicability of local fixpoint iteration in function lattices to arbitrary higher orders. To this end, we define a simple abstract and typed *higher-order fixpoint algebra* in Sect. 2 which can be used to describe evaluation problems involving fixpoints in such lattices. We then give a generic local algorithm for evaluating fixpoint terms in higher-order lattices in Sect. 3. It optimises the naïve fixpoint iteration method by localising the evaluation of recursively defined functions at the top order. A formal proof of its correctness is omitted due to space constraints. In Sect. 4 we present some computation problems which are special instances of the evaluation of higher-order fixpoints in various domains and discuss local evaluation's optimisation potential by comparing numbers of iteration, resp. argument computation steps on some hand-crafted examples. In Sect. 5 we briefly sketch limitations to the local approach of fixpoint iteration for higher-order fixpoints, in the form of obstacles to overcome which do not exist in the first-order case. We conclude in Sect. 6 with an outlook onto further work in this area.

2 An Abstract Higher-Order Fixpoint Algebra

Types and higher-order lattices. Let $\blacklozenge$ be some base type. Types are derived from the grammar

$$\tau ::= \blacklozenge \mid \tau^{v} \times \cdots \times \tau^{v} \to \tau \quad , \quad v ::= +, -, \pm$$

where the annotations v are called *variances*, and they specify the dependency of the values of a function of type $\tau_1^{v_1} \times \cdots \times \tau_n^{v_n} \to \tau$ on their arguments. In particular, if $v_i = +$, then this dependency is *monotonic*, if $v_i = -$ then it is *antitonic*, and it $v_i = \pm$ then it is unspecified.

The *order* of a type τ is $\mathsf{ord}(\blacklozenge) := 0$ and $\mathsf{ord}(\tau_1^{v_1} \times \cdots \times \tau_n^{v_n} \to \tau) := \max\{\mathsf{ord}(\tau), \mathsf{ord}(\tau_1)+1, \ldots, \mathsf{ord}(\tau_n)+1\}$

As usual, a function f on a partially ordered set $(M, \leq)$ is *monotonic* if for all $x, y \in M$ with $x \leq y$ we have $f(x) \leq f(y)$. It is *antitonic* if for all such x, y we have $f(y) \leq f(x)$. A lattice is a partial order in which suprema and infima, denoted $x \sqcup y$, $x \sqcap y$, resp. $\bigsqcup X$ and $\bigsqcap X$ for any $X \subseteq M$ exist, for as long as X is finite. A lattice is *complete* if these also exist for arbitrary X. Complete lattices always contain a least and a greatest element, usually denoted $\bot$ and $\top$ here. A finite lattice is trivially complete.

Let $\mathscr{M} = (M, \leq)$ and $\mathscr{M}_i = (M_i, \leq_i)$ for some $i = 1, \ldots, n$ be complete lattices. Remember the following constructions on lattices:

Inverse: $\mathscr{M}^- := (M, \geq)$ where $x \geq y$ iff $y \leq x$. For notational convenience we also let $\mathscr{M}^+ := \mathscr{M}$. It should be clear that these two operations not only preserve the property of being a lattice but also completeness.

Flattening: $\mathscr{M}^\pm := (M, =)$, where $=$ denotes equality as usual. Note that $\mathscr{M}^\pm$ is in general not a lattice anymore, let alone a complete one.

Product: $\prod_{i=1}^n \mathscr{M}_i := (M_1 \times \cdots \times M_n, \sqsubseteq)$ where $(x_1, \ldots, x_n) \sqsubseteq (y_1, \ldots, y_n)$ iff $x_i \leq_i y_i$ for all $i = 1, \ldots, n$. The product lattice is complete if all its components are complete.

Higher-order: $\mathscr{M}_1 \to \mathscr{M}_2 := (\{f \colon M_1 \to M_2 \mid f \text{ is monotonic } \}, \sqsubseteq)$ where $f \sqsubseteq g$ if $f(x) \leq_2 g(x)$ for all $x \in M_1$. The lattice of componentwise ordered monotonic functions from M_1 to M_2 is complete if $\mathscr{M}_2$ is complete. Completeness of $\mathscr{M}_1$ is not required, not even the property of being a lattice since $\leq_1$ is not used in the definition of $\sqsubseteq$.

We can use these constructions to associate, with each type τ, a complete lattice $\mathscr{M}_\tau$, given a complete lattice $\mathscr{M}$ interpreting the ground type $\blacklozenge$:

$$[\![\blacklozenge]\!]^{\mathscr{M}} := \mathscr{M} \quad , \quad [\![\tau_1^{v_1} \times \cdots \times \tau_n^{v_n} \to \tau]\!]^{\mathscr{M}} := \Big(\prod_{i=1}^n ([\![\tau_i]\!]^{\mathscr{M}})^{v_i}\Big) \to [\![\tau]\!]^{\mathscr{M}}$$

Note that each $[\![\tau]\!]^{\mathscr{M}}$ is indeed a complete lattice given the remarks above, as the flattening operation that breaks the lattice property is only used on the argument side of the function operator. Moreover, if $\mathscr{M}$ is finite, then so is $[\![\tau]\!]^{\mathscr{M}}$ for all τ.

Abstract higher-order fixpoint algebra. Let $\mathscr{M}$ be a complete lattice and $\mathsf{Func} = \{f \colon \tau_f, g \colon \tau_g, \ldots\}$ be a set of *computable* and *typed* functions on $\mathscr{M}$, possibly of higher-order. Note that if $\tau_f = \blacklozenge$, then f is not really a function but rather a constant. For simplicity we speak of functions in this case as well.

Let $\mathsf{Var} := \{x \colon \tau_x, y \colon \tau_y, \ldots\}$ be a set of typed variables. We write τ_x, resp. τ_f for the uniquely determined type of variable x, resp. function f. We will also simply write $x \in \mathsf{Var}$ instead of $(x, \tau_x) \in \mathsf{Var}$ and likewise for the members of Func.

Terms of the abstract higher-order fixpoint algebra over Func, $\mu\mathsf{HO}(\mathsf{Func})$ or simply $\mu\mathsf{HO}$ when Func is clear from the context, are built via

$$\varphi, \varphi_1, \ldots, \varphi_n \ ::= \ x \mid f \mid \varphi(\varphi_1, \ldots, \varphi_n) \mid \lambda x_1^{v_1}, \ldots, x_n^{v_n}.\, \varphi \mid \mu x.\, \varphi \mid \nu x.\, \varphi$$

where $x_1, \ldots, x_n \in \mathsf{Var}$, $f \in \mathsf{Func}$ and $v_1, \ldots, v_n \in \{+, -, \pm\}$.

A term φ is *closed* if it contains no free variables, where an occurrence of a variable x is free if it is not under the scope of some $\lambda \ldots x \ldots$ or μx or νx in the syntax tree of φ. In the following, we are mainly interested in closed terms; others will usually only occur as subterms of these. Hence, we will often simply speak of terms when in fact we mean closed terms at syntactic top-level.

$$\frac{}{\Gamma \vdash f\colon \tau_f} \qquad \frac{\Gamma \vdash \varphi\colon \tau_1^{v_1} \times \cdots \times \tau_n^{v_n} \to \tau \qquad \Gamma^{v_1} \vdash \varphi_1\colon \tau_1 \qquad \ldots \qquad \Gamma^{v_n} \vdash \varphi_n\colon \tau_n}{\Gamma \vdash \varphi(\varphi_1,\ldots,\varphi_n)\colon \tau}$$

$$\frac{v \in \{+,\pm\}}{\{x^v,\ldots\} \vdash x\colon \tau_x} \qquad \frac{\Gamma[x_i^{v_i} \mid i=1,\ldots,n] \vdash \varphi\colon \tau}{\Gamma \vdash \lambda x_1^{v_1},\ldots,x_n^{v_n}.\,\varphi\colon \tau_{x_1}^{v_1} \times \cdots \times \tau_{x_n}^{v_n} \to \tau} \qquad \frac{\Gamma[x^+] \vdash \varphi\colon \tau_x}{\Gamma \vdash \sigma x.\,\varphi\colon \tau_x}$$

Figure 1: The typing rules for abstract higher-order fixpoint algebra.

We assume terms to be *well-named*, i.e. each variable is bound at most once. Clearly, any term can always be made well-named by renaming bound variables.

For better readability, we simply write $\sigma x(y_1^{v_1},\ldots,y_n^{v_n}).\,\varphi$ instead of $\sigma x.\,\lambda y_1^{v_1},\ldots,y_n^{v_n}.\,\varphi$, for $\sigma \in \{\mu,\nu\}$.

In order to give terms a well-defined semantics via the Knaster-Tarski Theorem, each φ in a term $\mu x.\varphi$ or $\nu x.\varphi$ needs to denote a function that is monotone in its argument x. Monotonicity is guaranteed for *well-typed terms*, to be explained next, and then formally stated as Lemma 2.1 below. Note that the variances are used to track information about the monotonicity or antitonicity of functions in particular arguments, and that a monotonic function can be built for instance by composing two antitonic ones.

A *typing statement* is a triple $\Gamma \vdash \varphi\colon \tau$ where φ is a term, τ is a type, and Γ is a *typing context* consisting of *typing hypotheses* of the form x^v for $x \in \mathsf{Var}$ and v being a variance. For a typing context Γ, let $\Gamma^+ := \Gamma$; let Γ^- result from Γ by replacing in it every x^+ by x^- and vice-versa; and let $\Gamma^\pm = \Gamma^+ \cap \Gamma^-$, i.e. the context which only contains typing hypotheses of the form $x^\pm$ from Γ. The typing context $\Gamma[x^v]$ is obtained by removing $x^{v'}$ from Γ for any v', and adding x^v instead.

A term φ *has type* τ if the typing statement $\emptyset \vdash \varphi : \tau$ is derivable using the typing rules given in Fig. 1. The rules are standard; they state, for instance, that in function application $\varphi(\varphi_1,\ldots,\varphi_n)$, φ must have a function type with n arguments which are the types of the respective argument terms. Moreover, the arguments themselves have to be typbale in the respective derived typing contexts. For example, if φ is antitonic in its first argument, then φ_1 has to be typable in the typing context Γ^-, where Γ is the context used to type the whole application. This reflects the fact that an antitonic function from some lattice is a monotonic function from the inverse of this lattice (cf. the lattice definitions above and the definition of the semantics below). The rules for fixpoint formulas $\sigma x^\tau.\varphi$ require the term φ to be of the same type as x, since being a fixpoint intuitively means $x = \varphi(x)$, and at the same time ensure that φ is monotonic in x. A term is *well-typed*, if it is of some type.

Variance annotations are only used to guarantee well-typedness (and therefore the existence of fixpoints). We will always assume that terms are well-typed, and therefore often drop typing annotations for better readability. Note that for closed terms, a unique type for each subterm can easily be recovered.

The semantics of terms. Let $\mathscr{M}$ be a complete lattice, and suppose that all base functions $\mathsf{Func} = \{f\colon \tau_f,\ldots\}$ have an interpretation $f^{\mathscr{M}}$ in the family of higher-order lattices over $\mathscr{M}$ according to their types. A term φ of $\mu\mathsf{HO}(\mathsf{Func})$ over $\mathsf{Func} = \{f\colon \tau_f,\ldots\}$ and a set of typed variables $\mathsf{Var} = \{x\colon \tau_x,\ldots\}$ gets interpreted in this family of lattices. In order to explain the value inductively, we need variable interpretations η which assign values in lattices over $\mathscr{M}$ to any variable with free occurrences in subterms: for each $x\colon \tau_x \in \mathsf{Var}$ we have $\eta(x) \in [\![\tau_x]\!]^{\mathscr{M}}$. The value of φ over $\mathscr{M}$ and under η is denoted $[\![\varphi]\!]^{\mathscr{M}}_\eta$ and

is defined inductively as follows.

$$\begin{aligned}
[\![x]\!]^{\mathscr{M}}_{\eta} &:= \eta(x) & [\![\varphi(\varphi_1,\dots,\varphi_n)]\!]^{\mathscr{M}}_{\eta} &:= [\![\varphi]\!]^{\mathscr{M}}_{\eta}([\![\varphi_1]\!]^{\mathscr{M}}_{\eta},\dots,[\![\varphi_n]\!]^{\mathscr{M}}_{\eta}) \\
[\![f]\!]^{\mathscr{M}}_{\eta} &:= f^{\mathscr{M}} & [\![\lambda x_1^{\nu_1},\dots,x_n^{\nu_n}.\,\varphi]\!]^{\mathscr{M}}_{\eta} &:= (f_1,\dots,f_n) \mapsto [\![\varphi]\!]^{\mathscr{M}}_{\eta[x_1\mapsto f_1,\dots,x_n\mapsto f_n]} \\
[\![\mu x.\,\varphi]\!]^{\mathscr{M}}_{\eta} &:= \bigsqcup\{f \in [\![\tau_x]\!]^{\mathscr{M}} \mid [\![\varphi]\!]^{\mathscr{M}}_{\eta[x\mapsto f]} \sqsubseteq f\} & [\![\nu x.\,\varphi]\!]^{\mathscr{M}}_{\eta} &:= \bigsqcap\{f \in [\![\tau_x]\!]^{\mathscr{M}} \mid f \sqsubseteq [\![\varphi]\!]^{\mathscr{M}}_{\eta[x\mapsto f]}\}
\end{aligned}$$

The fourth clause, in particular its right-hand side, denotes the function that maps a tuple $(f_1,\dots,f_n)$ of objects from $[\![\tau_{x_1}]\!]^{\mathscr{M}} \times \cdots \times [\![\tau_{x_n}]\!]^{\mathscr{M}}$ to the value $[\![\varphi]\!]^{\mathscr{M}}_{\eta[x_1\mapsto f_1,\dots,x_n\mapsto f_n]}$ where the subscript index denotes the variable environment that results from η by replacing its bindings for $x_1,\dots,x_n$ accordingly. For the last two clauses, note that the values on the right-hand side are well-defined according to the Knaster-Tarski Theorem [32] since each $[\![\tau_x]\!]^{\mathscr{M}}$ is a complete lattice. Note that the semantics of $\mu\mathsf{HO}$ are easily seen to be invariant under β-reduction.

Lemma 2.1. *Let $\mathscr{M}$ be a lattice, φ be a term of type τ' under the typing assumptions Γ, x^{ν}. If $\nu = +$, then $[\![\varphi]\!]^{\mathscr{M}}_{\eta}$ is monotone in $\eta(x)$; if $\nu = -$ then $[\![\varphi]\!]^{\mathscr{M}}_{\eta}$ is antitone in $\eta(x)$.*

Proof. By a straightforward induction on the syntax tree of φ. □

Remark 2.2. Over *finite* lattices, each of the type lattices is finite as well. According to Kleene's Fixpoint Theorem, the least and greatest fixpoints of a term $\sigma x.\,\varphi$ in $\mu\mathsf{HO}$ under a variable interpretation η can be computed by a sequence of approximations as follows: $x^0_\eta = \top_{\tau_x}$ if $\sigma = \nu$, $x^0_\eta = \bot_{\tau_x}$ if $\sigma = \mu$, and $x^{i+1}_\eta = [\![\varphi]\!]^{\mathscr{M}}_{\eta[x\mapsto x^i_\eta]}$. Then, for each finite lattice $\mathscr{M}$ there is $n \in \mathbb{N}$ such that $[\![\sigma x.\,\varphi]\!]^{\mathscr{M}}_{\eta} = x^n_\eta$. Moreover, these approximations are definable in $\mu\mathsf{HO}$, independently of η: $\hat{\sigma}$ is defined by $x^0_\eta = \sigma x.\,x$, and x^{i+1}_η is defined by the substitution instance $\varphi[x^i_\eta/x]$.

Evaluation problems. We consider the following generic *evaluation* problem: given a (closed) term φ of $\mu\mathsf{HO}(\mathsf{Func})$ with symbols in Func interpreted in the higher-order lattices over a finite $\mathscr{M}$, compute $[\![\varphi]\!]^{\mathscr{M}}$.

This problem is clearly decidable when all basic functions in Func are computable. A naïve algorithm will simply compute the value of each subterm in a bottom-up fashion using Kleene iteration to evaluate fixpoint expressions, and possibly storing function values as tables. Note that if $\mathscr{M}$ is finite, so is $[\![\tau]\!]^{\mathscr{M}}$ for any τ, but the size and height of $[\![\tau]\!]^{\mathscr{M}}$ are k-fold exponential in the size, resp. height of $\mathscr{M}$ when $k = \mathsf{ord}(\tau)$.

Even for low orders, such a naïve algorithm may perform far too many unnecessary computation steps. Consider the following special *local* variant of the evaluation problem: given a finite complete lattice $\mathscr{M}$, a closed term $\varphi_0 := \mu x.\,\varphi$ of type $\tau^{\nu} \to \blacklozenge$ (which is then necessarily the same as τ_x) for some ν, τ, φ, and a term ψ of type τ, compute $[\![\varphi_0(\psi)]\!]^{\mathscr{M}}$.

Note how this problem formulation describes a situation in which naïve fixpoint iteration obviously performs too many evaluation steps in general: it computes $[\![\varphi_0]\!]^{\mathscr{M}}$ using Kleene iteration which results in a function of type $\tau^{\nu} \to \blacklozenge$. Depending on the order of τ, this function is huge in terms of its arguments but still finite. We would then also compute $[\![\psi]\!]^{\mathscr{M}}$. Then we obtain the value $[\![\varphi_0(\psi)]\!]^{\mathscr{M}}$ by application, for instance through a simple look-up in the table representing $[\![\varphi_0]\!]^{\mathscr{M}}$, where $[\![\psi]\!]^{\mathscr{M}}$ occurs as some argument. Clearly, the value of $[\![\varphi_0]\!]^{\mathscr{M}}$ on all other arguments is irrelevant, and the reason for their computation is questionable.

	0	1	2	3	4	5	6	7
F^0:	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
F^1:	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
F^2:	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$
F^3:	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$

	5	0
F^0:	$\bot$	$\bot$
F^1:	$\bot$	$\top$
F^2:	$\top$	$\top$
F^3:	$\top$	

	3	2	1	4
F^0:	$\bot$	$\bot$	$\bot$	$\bot$
F^1:	$\bot$	$\bot$	$\bot$	$\bot$
F^2:	$\bot$	$\bot$	$\bot$	
F^3:	$\bot$	$\bot$		
F^4:	$\bot$			

Figure 2: Global (left) vs. local fixpoint iteration for a first-order function F.

A (first-order) example. Consider the Boolean lattice $\mathbb{B} = \{\top, \bot\}$ and the normal Boolean functions $\mathsf{Func}_{\mathsf{bool}} = \{\wedge, \vee \colon \blacklozenge^+ \times \blacklozenge^+ \to \blacklozenge, \neg \colon \blacklozenge^- \to \blacklozenge, 0, 1 \colon \blacklozenge\}$ interpreted in the standard way. Let $n > 0$ and

$$\varphi_n \;:=\; \mu F(\underbrace{x_0, \ldots, x_{n-1}}_{\vec{x}}).\ \mathsf{null}(\vec{x}) \vee \big(\mathsf{even}(\vec{x}) \wedge F(\mathsf{half}(\vec{x}))\big) \\ \vee \big(\neg\mathsf{even}(\vec{x}) \wedge F(\mathsf{add}(\mathsf{add}(\vec{x}, \mathsf{dbl}(\vec{x})), (1, 0, \ldots, 0)))\big)\ .$$

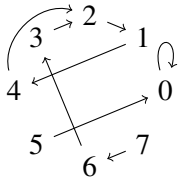
over $\mathsf{Var} = \{x_0, \ldots, x_{n-1} \colon \blacklozenge, F \colon (\blacklozenge^{\pm})^n \to \blacklozenge\}$ where, for $\vec{x} = (x_0, \ldots, x_{n-1})$ and $\vec{y} = (y_0, \ldots, y_{n-1})$,

- $\mathsf{null}(\vec{x})$ returns $\top$ iff $\vec{x}$ encodes the numerical value 0, for instance $\mathsf{null}(\vec{x}) = \bigwedge_{i=0}^{n-1} \neg x_i$;
- $\mathsf{even}(\vec{x})$ returns $\top$ iff $\vec{x}$ encodes an even number, for instance $\mathsf{even}(\vec{x}) = \neg x_0$;
- half and dbl represent the operations "$\div 2$" and "$\cdot 2$" on bit strings;
- $\mathsf{add}(\vec{x}, \vec{y})$ yields a bit string representing the addition of the two values modulo 2^n.

It is not difficult to find Boolean functions realising these operations.

Intuitively, φ_n defines a search procedure. Note that any value of $\vec{x}$ encodes a number in the range $[2^n] = \{0, \ldots, 2^n - 1\}$ which we will simply denote $\vec{x}$ as well. For any value $\vec{x}$, define a sequence $(\vec{x}_k)_{k \geq 0}$ via $\vec{x}_0 = \vec{x}$, $\vec{x}_{k+1} = \vec{x}_k / 2$ if $\vec{x}$ is even, and $\vec{x}_{k+1} = 3 \cdot \vec{x}_k + 1$ if $\vec{x}$ is odd. Hence, suppose $\vec{b} \in \{0,1\}^n$ encodes such a number, then $\varphi_n(\vec{b})$ is true iff this sequence eventually hits the value 0.

Let $n = 3$. The graph on the right depicts the sequence on all values in $[2^3]$. Assuming that the Boolean function $\wedge$ only evaluates its second argument when the first one is not $\bot$, this graph suggests how local fixpoint iteration of this first-order function F can be more efficient when the value of the fixpoint F is only needed on one particular argument. The effect of global fixpoint iteration is depicted in Fig. 2 (left). Here, the iteration starts with the least function $F^0 : \_ \mapsto \bot$, and it terminates when the current approximation equals the last one.



Local fixpoint iteration on the other hand only adds the arguments successively to those tables. Consider the case of evaluating $\varphi_3(1,0,1)$ which means iterating the numerical series beginning at 5. First we only tabulate the approximant F^0 on the value under consideration, i.e. 5. In order to compute $F^1(5)$ we need $F^0(0)$, so 0 gets added as a new argument and receives the initial value $\bot$ there. Then we compute $F^2(5)$ and $F^1(0)$, and so on. The iteration stabilises when no change is being recorded anymore, thus computing values as they are shown in the table of Fig. 2 (middle).

The effect of computing $\varphi_3(1,1,0)$, i.e. beginning the numerical series at 3 is similar. Here, however, 0 is never reached. Hence, all values encountered are $\bot$, and the iteration stabilises when no further arguments are needed, as shown in Fig. 2 (right).

Even though the local iteration computes a value of F^4 while the global one only reaches F^3, it should be clear that local evaluation performs fewer computation steps in general.

Remark 2.3. It is well-known that fixpoint iteration does not need to record the entire history of its computation but, for each variable, merely the value of the last iteration. In the left table (with only one fixpoint variable) in Fig. 2, this corresponds to two successive rows: the upper for the last approximate value of a function, and the lower for the current value. In the two tables on the right using local iteration, this corresponds to diagonals, but the picture is more complicated in general, for instance for fixpoint terms in which the fixpoint variable has multiple occurrences.

The tables shown in Fig. 2 not only give an idea of how local fixpoint iteration works, their width and height are also good measures for the space, resp. time needed to compute such a higher-order fixpoint.

3 Local Fixpoint Evaluation for Full $\mu\mathsf{HO}$

The algorithm. Procedure EVAL in Alg. 1 solves the evaluation problem for $\mu\mathsf{HO}$ terms of arbitrary higher order and finite lattices using local fixpoint iteration. It takes four parameters: (1) a term $\varphi \in \mu\mathsf{HO}(\mathsf{Func})$ over some Func. It is not necessarily of type $\blacklozenge$, but the algorithm is assumed to be started with a full list of arguments (see below) in order to realise local fixpoint iteration. (2) A finite, and, hence, complete lattice $\mathscr{M}$ with an interpretation of any $f\colon \tau_f \in \mathsf{Func}$ as an object in $[\![\tau_f]\!]^{\mathscr{M}}$. (3) A list $T_1,\dots,T_k$ of arguments to φ. The following invariant is maintained: if the type[1] of φ is $\tau_1 \to \cdots \to \tau_k \to \blacklozenge$, then $T_i \in [\![\tau_i]\!]^{\mathscr{M}}$ for all $1 \le i \le k$. (4) A global variable ENV that is used to interpret free variables. Values of λ-bound variables are stored as full functions[2], values of fixpoint variables may be stored as partial approximations as described at the end of the previous section.

In order to bridge the gap between a variable assignment η, which assigns a value to each variable which is defined at every argument, and the global variable ENV which only stores partial approximations for fixpoint-bound variables, consider the following definition. It turns a state of ENV into a well-defined variable assignment:

$$\eta_{\mathrm{ENV}}(x)(T_1,\dots,T_k) = \begin{cases} \mathrm{ENV}(x)(T_1,\dots,T_k) & \text{, if } x \text{ is } \lambda\text{-bound} \\ \mathrm{ENV}(x)(T_1,\dots,T_k) & \text{, if } x \text{ is fixpoint-bound and } \mathrm{ENV}(x)(T_1,\dots,T_k) \text{ is defined} \\ \hat{\sigma}_x & \text{, otherwise} \end{cases}$$

Here, $\hat{\sigma}_x$ is $\bot_{\blacklozenge}$, resp. $\top_{\blacklozenge}$ for variables that are bound to a least, resp. greatest fixpoint.

Note that, due to the invariants, a state of Alg. 1, i.e. a call of $\mathrm{EVAL}(\varphi,T_1,\dots,T_k)$ with a value of ENV and over some higher-order lattice $\mathscr{M}$, can be thought of as computing the object $[\![\varphi]\!]^{\mathscr{M}}_{\eta_{\mathrm{ENV}}}(T_1,\dots,T_k)$, which is always a member of $[\![\blacklozenge]\!]^{\mathscr{M}}$. The algorithm computes this value recursively by descending through the syntax tree of φ. Fixpoints are resolved by Kleene iteration until the semantics computed stabilises, but the value is only computed at the arguments indicated plus all those arguments that are discovered as necessary to obtain the value for the original argument.

We explain the algorithm's functionality by considering the different cases for its argument φ. Upon reaching a basic function symbol, EVAL simply applies the semantics of this basic function to the arguments in the argument list. When EVAL reaches a variable x and the value of that variable at argument $(T_1,\dots,T_k)$ is defined, then its value is returned. Otherwise, the variable must be fixpoint-bound, and EVAL has discovered a new tuple of arguments at which the value of this fixpoint is needed. This value is initialised as $\hat{\sigma}_x$, which also registers $(T_1,\dots,T_k)$ in ENV. In this case the initial value is returned.

[1] Variances are not important in this section. In order to reduce clutter, we do not display them.

[2] This might appear wasteful at first, but λ-bound variables are never of the highest type (by order) that occurs in the term to be evaluated except in pathological cases, which can be eliminated by β-reduction before calling EVAL.

Algorithm 1 Neededness-based evaluation for abstract higher-order fixpoint algebra.

procedure $\textsc{Eval}(\varphi, T_1, \ldots, T_k)$: ▷ global (partial) $\textsc{env} : \mathsf{Var} \to \bigcup_{x \in \mathsf{Var}} [\![\tau_x]\!]^{\mathscr{M}}$

switch φ:

case f: **return** $f(T_1, \ldots, T_k)$

case x: **if** $\textsc{env}(x)(T_1, \ldots, T_k) = \texttt{undef}$ **then** $\textsc{env}(x) := \textsc{env}(x)[(T_1, \ldots, T_k) \mapsto \hat{\sigma}_x]$
return $\textsc{env}(x)(T_1, \ldots, T_k)$

case $\lambda x_1, \ldots, x_n.\, \varphi'$: $\textsc{env}(x_1) := T_1; \ldots; \textsc{env}(x_n) := T_n$
return $\textsc{Eval}(\varphi', T_{n+1}, \ldots, T_k)$

case $\varphi'(\varphi_1, \ldots, \varphi_n)$: **for** $i = 1, \ldots, n$ **do**
$\text{let } \tau_1 \to \cdots \to \tau_{k'} \to \blacklozenge = \text{type}(\varphi_i)$
$f_i := \{(T'_1, \ldots, T'_{k'}) \mapsto \textsc{Eval}(\varphi_i, T'_1, \ldots, T'_{k'}) \mid T'_i \in [\![\tau_i]\!]^{\mathscr{M}}, i = 1, \ldots, n\}$
return $\textsc{Eval}(\varphi', f_1, \ldots, f_n, T_1, \ldots, T_k)$

case $\sigma x.\, \varphi'$: $\textsc{env}(x) := \{(T_1, \ldots, T_k) \mapsto \hat{\sigma}_x\}$
repeat
$f := \textsc{env}(x)$
for all $(T'_1, \ldots, T'_k) \in \text{dom}(\textsc{env}(x))$ **do**
$\textsc{env}(x) := \textsc{env}(x)[(T'_1, \ldots, T'_k) \mapsto \textsc{Eval}(\varphi', T'_1, \ldots, T'_k)]$
until $f = \textsc{env}(x)$
return $\textsc{env}(x)(T_1, \ldots, T_k)$

At a λ-abstraction, a number of arguments corresponding to the abstracted variables are transferred from the argument list to ENV, i.e. they are now treated as bound variables. In an application $\varphi(\varphi_1, \ldots, \varphi_k)$, EVAL computes, for each argument, its full semantics by a number of recursive calls to EVAL[3]. The obtained values (as functions) are then added to the list of arguments.

Upon reaching a fixpoint binder for variable x, EVAL (re-)sets $\textsc{env}(x)$ to the singleton definition that initialises the value of the fixpoint at $(T_1, \ldots, T_k)$ to the default value of $\hat{\sigma}_x$. Then, for each argument tuple that is already discovered as necessary for the value at $(T_1, \ldots, T_k)$, the algorithm computes a new value. Note that, during this process EVAL can reach the variable case and discover new argument tuples. This procedure of updating the value at all known argument tuples is repeated until both no new arguments are discovered for one round, and the value of the fixpoint at each of the tuples agrees with that of the last round. If this has happened, the value of the last iteration at $(T_1, \ldots, T_k)$ is returned.

Correctness. The formal correctness proof for Alg. 1 uses the following lemma which formalises the converse of Lemma 2.1. Take a term φ that is typed with hypotheses Γ, x^v. Not only is it monotone (if $v = +$), respectively antitone (if $v = -$) in the value of x. If the value of φ also differs genuinely under two variable interpretations that only differ in x, then x must occur freely in φ and there are arguments to the value of x on which this difference manifests itself. We write $x \sqsubset y$ to denote that $x \sqsubseteq y$ but $x \neq y$.

Lemma 3.1. *Let $\mathscr{M}$ be a finite, and hence, complete lattice, η be a variable interpretation, $f_1, f_2 \in [\![\tau']\!]$ with $\tau' = \tau'_1 \to \cdots \to \tau'_k \to \blacklozenge$ for some $\tau'_1, \ldots, \tau'_k$, let $T_1, \ldots, T_n$ be values with $T_i \in [\![\tau_i]\!]$ for $i = 1, \ldots, n$, $v \in \{+, -\}$, and φ be a $\mu\mathsf{HO}$ term such that $\Gamma, x^v \colon \tau' \vdash \varphi \colon \tau_1^{v_1} \to \cdots \to \tau_n^{v_n} \to \blacklozenge$. If*

$$[\![\varphi]\!]^{\mathscr{M}}_{\eta[x \mapsto f_1]}(T_1, \ldots, T_n) \sqsubset_{\blacklozenge} [\![\varphi]\!]^{\mathscr{M}}_{\eta[x \mapsto f_2]}(T_1, \ldots, T_n)$$

[3] This can be done lazily, in case the argument is not needed or has been already computed. We omit the details for this in order to keep the presentation simple.

then x appears freely in φ, and there are $T'_1,\dots,T'_n$ such that

$$f_1(T'_1,\dots,T'_n) \gtrless f_2(T'_1,\dots,T'_n)$$

with $\gtrless = \sqsubset_\blacklozenge$ if $v = +$ and $\gtrless = \sqsupset_\blacklozenge$ if $v = -$.

Proof. By induction on the structure of φ. Details are omitted. □

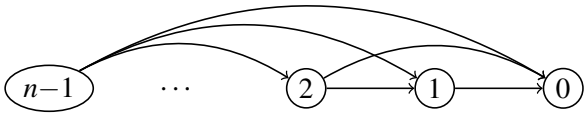
Next we state correctness of Alg. 1. It is not hard to imagine local fixpoint iteration to be sound (for least fixpoints, resp. complete for greatest ones) since it clearly only performs part of a global fixpoint iteration that is sound and complete according to Kleene's Theorem. For completeness one has to see that the value of a fixpoint function on some argument is determined solely by the value of that function on its dependent arguments. Hence, it suffices to iterate on these until stability is reached.

We write $\textsc{Eval}_\eta(\varphi,[T_1,\dots,T_k])$ for the result of the call of Alg. 1 on $\mathscr{M}$ with arguments φ and $[T_1,\dots,T_k]$ such that ENV satisfies $\eta = \eta_{\text{ENV}}$.

Theorem 3.2. *Let $\mathscr{M}$ be a finite, and, hence, complete lattice, η be a variable interpretation, φ be a term of type $\tau_1 \to \cdots \to \tau_k \to \blacklozenge$, let $T_1,\dots,T_k$ be values with $T_i \in [\![\tau_i]\!]^{\mathscr{M}}$, and* ENV *be such that $\eta = \eta_{\text{ENV}}$. Then $\textsc{Eval}_\eta(\varphi,[T_1,\dots,T_k]) = [\![\varphi]\!]^{\mathscr{M}}_\eta(T_1,\dots,T_k)$.*

Proof. The proof is, again, by induction on the structure of φ. Details are omitted. □

A natural question that arises is the one after the time and space complexity of local higher-order fixpoint iteration. Two aspects need to be considered here. First of all, it should be obvious that local evaluation cannot improve the worst-case. It is in fact not hard to construct examples which a fixpoint term of higher-order such that its evaluation causes all argument values to be explored. Consider the (order-1) term $\big(\nu F(x).F((x \wedge \Diamond\neg x) \vee (\neg x \wedge \Box x))\big)(\mathtt{ff})$ with $\wedge,\vee,\neg,\Diamond,\Box$ interpreted in the usual way known from modal logic, over the powerset lattice induced by the Kripke structure



Even though the term evaluates to $\{0,\dots,n-1\}$, local fixpoint iteration will successively discover all 2^n arguments to the first-order function F before termination. It is also possible to extend this example to an arbitrary higher order.

Second, the question after the space and time complexity of Alg. 1 cannot be answered without making assumptions on the representation of the lattice and the complexity of evaluating base functions. So far, no assumptions have been made explicitly, even though it is clear that such functions should at least be computable for otherwise Alg. 1 would not be well-defined. A reasonable assumption is that each base function of order k can be evaluated in time and space that is at most $(k-1)$-fold exponential in the size of the underlying lattice, with 0-fold meaning polynomial and (-1)-fold meaning logarithmic. Logarithmic bounds may seem highly restrictive at first glance, but they make sense in cases where the underlying lattice is obtained as the powerset lattice of some other structure, see the example above. If this assumption is met, then it is not too hard to see that Alg. 1 runs in time and space that is at most k-fold exponential with k being the order of the input term. This also assumes that the lattice is given in a logarithmically sized representation. Otherwise, the complexity drops by one exponential.

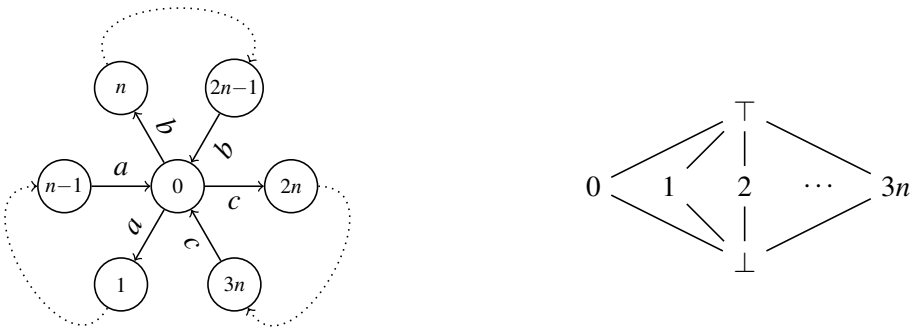


Figure 3: Directed graph G_n with $n > 1$ (left) for the language-constrained reachability problem example and corresponding base lattice (right).

4 Applications

We present four applications of fixpoint evaluation in higher-order lattices using local fixpoint evaluation, and estimate how many computation steps can be saved compared to a naïve bottom-up and global fixpoint iteration.

Constrained reachability problems. Reachability problems – to decide whether some node is reachable from another in a directed graph – are ubiquitous in computer science. In some applications, simple reachability is too coarse; instead one wants to put constraints on the form of path under which the target node can be reached from the source, for instance in terms of distance, weight, shape or allowed sequence of edges. The latter can easily be formalised as a reachability problem constrained by some formal language. This has been investigated thoroughly for regular [6] and context-free languages [7, 3, 24] for applications in database theory, model checking, or in static program analysis for heap-manipulating programs [25, 16]. Little has been done for larger classes of languages.

We consider the context-sensitive language $L_{\mathsf{abc}} := \{a^n b^n c^n \mid n \geq 1\}$ over the alphabet $\Sigma := \{a,b,c\}$, and the problem whether for some given nodes s,t of a directed, edge-labelled graph $G = (V,E)$ there is a path from s to t whose edge labels form a word in L_{abc}. Reachability problems can be interpreted in powerset lattices $(2^V, \subseteq)$ as they can be seen as least fixpoints of functions from sets of nodes to sets of nodes. However, if G is backwards-deterministic, i.e. for all $v,u,w \in V$, $a \in \Sigma$ we have $(v,a,u) \in E \wedge (w,a,u) \in E \Rightarrow v = w$, it is possible to formalise such problems over a smaller lattice.

Consider the graph G_n depicted in Fig. 3 on the left. It contains a central state 0 and around this three loops: an a-loop with $n-1$ states, a b-loop with n states and a c-loop with $n+1$ states. It is backwards-deterministic. Let $\blacklozenge$ be interpreted by the lattice shown in Fig. 3 on the right. Intuitively, $\bot$ can be read as "a path from source to the target has not been found yet", and $\top$ signals that such a path has been found.

We use base functions $\mathscr{F} := \{0\colon \blacklozenge,\ a,b,c\colon \blacklozenge^+ \to \blacklozenge,\ \mathsf{ite}\colon \blacklozenge^\pm \times \blacklozenge^+ \to \blacklozenge\}$ as follows. The constant 0 denotes the state 0. For any $v \in V$, $a(v)$ is the a-predecessor of v; likewise for b and c. The value is $\bot$ if no such predecessor exists, in particular when applied to $\bot$. The value on $\top$ is $\top$ itself. For instance, $c(0) = 3n$, $c(2n) = 0$, $c(v) = v-1$ if $2n < v \leq 3n$, $c(\top) = \top$ and $c(v) = \bot$ otherwise.

In the powerset lattice $(2^V, \subseteq)$, ite could simply be interpreted as set union. However, here we

$$
\begin{aligned}
P &\to \mathtt{b}\cdot B(S)\\
S &\to \mathtt{␣} \mid \mathtt{␣}\cdot S\\
B(I) &\to \varepsilon\\
&\mid I\cdot \mathtt{c}\cdot B(I)\\
&\mid I\cdot \mathtt{b}\cdot B(S\cdot I)
\end{aligned}
\qquad
\begin{array}{l}
\mathtt{b}\\
\mathtt{␣␣c}\\
\mathtt{␣␣b}\\
\mathtt{␣␣␣c}\\
\mathtt{␣␣␣c}\\
\mathtt{␣␣c}
\end{array}
$$

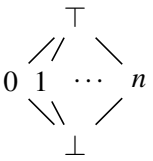


Figure 4: Higher-order grammar $\mathscr{G}_{\mathsf{ind}}$ (left); example word (middle); lattice $\mathscr{M}_w$ (right).

interpret it as an *if-then-else* in the following way. Note that it is only monotonic in its second argument.

$$\mathsf{ite}(x,y) := \begin{cases} \top & \text{, if } x = 0 \\ y & \text{, otherwise} \end{cases}$$

Now let $\mathsf{Var} = \{f,g\colon \blacklozenge^+ \to \blacklozenge,\ x\colon \blacklozenge,\ F\colon (\blacklozenge^+ \to \blacklozenge)^\pm \times (\blacklozenge^+ \to \blacklozenge)^\pm \times \blacklozenge^\pm \to \blacklozenge\}$ and consider the term

$$\varphi_{\mathsf{reach}} \ := \ \Big(\mu F(f,g,x).\,\mathsf{ite}\big(f(g(x)),F(a\circ f,b\circ g,c(x))\big)\Big)\,\big(a,b,c(0)\big)$$

where $\psi\circ\chi := \lambda x.\psi(\chi(x))$.

Using fixpoint unfolding and β-reduction one can see that the value of φ_{reach} becomes $\top$ when $a(b(c(0)) = 0$ or $a(a(b(b(c(c(0)))))) = 0$ or $a^3(b^3(c^3(0))) = 0$ and so on. Hence, evaluating φ_{reach} solves the reachability question "is there a path from 0 to 0 under some word in L_{abc}?"

We analyse how much computation power is being saved when computing the answer to the question of whether there is an L_{abc}-path from 0 to 0 in G_n. We compare four situations arising from the use of the standard powerset lattice vs. the optimised flat lattice of Fig. 3, as well as local vs. global enumeration of all arguments to higher-order functions.

Note that the three cycles in G_n have lengths n, $n+1$ and $n+2$ respectively, which are always coprime for each $n \geq 2$. Hence, the shortest L_{abc}-path from 0 to 0 is the one that performs $(n+1)(n+2)$ many rounds on the a-cycle, then $n(n+2)$ rounds on the b-cycle and then $n(n+1)$ rounds on the c-cycle.

The following table shows the computational effort needed to evaluate φ_{reach} in terms of the number of arguments, resp. width of the table representing the function F. It also shows the space that is needed in order to represent one argument, i.e. a triple (f,g,x) where f,g are first-order functions and x is a lattice element. Finally, in all cases the height of the table, i.e. the number of fixpoint iterations needed until F stabilises, is in $\mathscr{O}(n^3)$.

	powerset lattice		flat lattice	
evaluation	global	local	global	local
width of table for F	$2^{2^{\mathscr{O}(n)}}$	$\mathscr{O}(n^3)$	$2^{\mathscr{O}(n\cdot\log n)}$	$\mathscr{O}(n^3)$
size of arguments	$2^{\mathscr{O}(n)}$		$\mathscr{O}(n\log n)$	

Parsing of programming languages with indentation. Some programming languages like HASKELL or PYTHON use indentation as a syntax element. Such an effect can be described conveniently by the higher-order grammar $\mathscr{G}_{\mathsf{ind}}$ shown in Fig. 4 (left), over the terminal alphabet $\Sigma = \{\mathtt{b},\mathtt{c},\mathtt{␣}\}$ (for "block",

"code" and "space"). We refrain from defining higher-order grammars [26, 34] formally, since the technicalities needed for this small example are quite intuitive: $B(I)$ generates a block of code at indentation level I. The block can either be empty, contain one line followed by the rest of this block at the same indentation level, or start a new block at a greater indentation level. The symbol S is used to generate a sequence of space characters '␣'. Finally, P generates a program as a block at some initial indentation level. An example word, generated from this grammar and formatted in order to visualise indentation best, is shown in the middle of Fig. 4.

Suppose a word $w = a_0 \dots a_{n-1} \in \Sigma^*$ is given. This gives rise to an interpretation of the symbols in the grammar above as follows. Let $\mathscr{M}_w$ be the lattice shown in Fig. 4 on the right. We use the following base functions, derived from the terminal symbols and the constructors in the higher-order grammar.

$$\mathsf{Func}_{\mathsf{ind}} = \{\mathtt{b}, \mathtt{c}, \mathtt{␣}, \varepsilon \colon \underbrace{\blacklozenge^{\pm} \times \blacklozenge^{\pm} \to \blacklozenge}_{\tau},\ \cdot, | \colon \tau^+ \times \tau^+ \to \tau,\ \ \mathsf{start}, \mathsf{end} \colon \blacklozenge\}$$

Intuitively, $\mathtt{b}$, $\mathtt{c}$, $\mathtt{␣}$ and ε are used to mark sections of the word in which the corresponding symbols (resp. the empty word) occur. The top and bottom element of the lattice are used to signal true/false. The interpretation of these functions is therefore simply

$$\varepsilon(i,j) = \begin{cases} \top & \text{, if } i = j \\ \bot & \text{, otherwise} \end{cases} \quad \text{and} \quad a(i,j) = \begin{cases} \top & \text{, if } i+1 = j \text{ and } a_i = a \\ \bot & \text{, otherwise} \end{cases}$$

for $a \in \Sigma$. The two constructors $|$ and $\cdot$ denoting disjunctive choice and concatenation in the higher-order grammar are interpreted as follows.

$$(f|g)(i,j) = \begin{cases} \top & \text{, if } f(i,j) = \top \text{ or } g(i,j) = \top \\ \bot & \text{, otherwise} \end{cases}$$

and

$$(f \cdot g)(i,j) = \begin{cases} \top & \text{, if there is } h \text{ s.t. } f(i,h) = \top = g(h,j) \\ \bot & \text{, otherwise} \end{cases}$$

Finally, we need two constants start and end which are interpreted as 0 and n, respectively.

The nonterminals in the higher-order grammar can be seen as (fixpoint) variables, hence we have $\mathsf{Var} = \{P, S, I \colon \tau, B \colon \tau^+ \to \tau\}$. Then $\mathscr{G}_{\mathsf{ind}}$ immediately becomes a second-order term of $\mu\mathsf{HO}$ over $\mathsf{Func}_{\mathsf{ind}}$ and Var, since recursion in grammars is captured by least fixpoints. The problem of evaluating $P(\mathsf{start}, \mathsf{end})$ over $\mathscr{M}_w$ is then equivalent to parsing w w.r.t. $\mathscr{G}_{\mathsf{ind}}$.

Clearly, the space and time needed to evaluate $P(\mathsf{start}, \mathsf{end})$ is dominated by the fixpoint iteration for B as the only second-order variable. The number of possible arguments to it is $2^{\mathscr{O}(n^2)}$. Local fixpoint iteration only discovers a fraction of these, though. Note that B is initially evaluated on S, and – when the recursive is called on argument I – it needs the values of B on I itself as well as on $S \cdot I$. Hence, it only ever discovers $S, S^2, S^3, \dots$. Moreover, it is not hard to see that S^k maps two positions (i,j) of an underlying word $w = a_0 \dots a_{n-1}$ to $\top$, if $j - 1 - i \geq k$ and $a_h = \mathtt{␣}$ for $h = i, \dots, j-1$. Hence, the number of possible arguments to B discovered in this way is bounded by $n-1$ and so, again, local higher-order fixpoint iteration realises an exponential reduction in space complexity in this example.

Model checking Higher-Order Fixpoint Logic. Fixpoints play a fundamental role in model checking, where properties of the runtime behaviour of programs are typically expressed in temporal logics, the

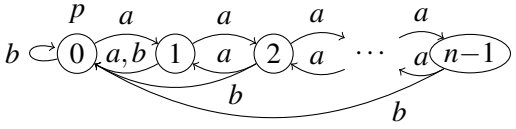
most prominent of which are LTL and CTL. Fixpoints are used there to express limit behaviour as in reachability, safety and fairness [17]. The true power of fixpoints is unleashed in logics that extend modal logic by extremal fixpoint quantifiers like the well-known modal μ-calculus [23]. A lesser known extension of this is Higher-Order Fixpoint Logic (HFL) [33], a highly expressive specification logic that mixes modal logic, a typed λ-calculus and fixpoint quantifiers. Its model checking problem is decidable over finite transition systems, albeit k-fold exponential in the order of involved function types [4, 10].

We refer to the literature for a self-contained definition of HFL [33]. With the preliminary work on abstract higher-order fixpoint algebra in Sect. 2 we can simply present HFL as a special instantiation of this algebra. The base type $\blacklozenge$ is interpreted as the powerset lattice $(2^{\mathscr{S}}, \subseteq)$ of the state set of a transition system with edge labels from some set $\mathscr{A}$ and propositional labels from some set $\mathscr{P}$. This gives rise to an interpretation of all higher-order types as functions on sets of states; a function of type $\blacklozenge^+ \to \blacklozenge$ for instance is known as a (monotonic) *predicate transformer*.

The set of ground functions then is $\mathscr{F} := \{\wedge, \vee \colon \blacklozenge^+ \times \blacklozenge^+ \to \blacklozenge,\ \neg \colon \blacklozenge^- \to \blacklozenge\} \cup \{\langle a \rangle, [a] \colon \blacklozenge^+ \to \blacklozenge \mid a \in \mathscr{A}\} \cup \{p \colon \blacklozenge \mid p \in \mathscr{P}\}$ reflecting the Boolean, modal and propositional parts of the logic. Well-typed terms of the higher-order fixpoint algebra over this $\mathscr{F}$ are exactly the formulas of HFL; and the standard semantics is the same as the one derived from the generic semantics in Sect. 2 for this set of terms.

Consider the following formula describing the property $\varphi =$ "there is an infinite b-path such that the i-th node on this path is the start of an a-path of length 2^i ending in a p-node, for any $i \geq 0$", as well as the family of transition systems $\mathscr{T}_n$ on the right.

$$\varphi := \left(\nu F.\, \lambda f.\, (f\, p) \wedge \langle b \rangle (F(f \circ f))\right) (\lambda x. \langle a \rangle x)$$



where $\varphi \circ \psi := \lambda x. \varphi(\psi x)$, over $\mathsf{Var} = \{x \colon \blacklozenge,\ f \colon \blacklozenge^+ \to \blacklozenge,\ F \colon (\blacklozenge^+ \to \blacklozenge)^+ \to \blacklozenge\}$. We use F in the following to abbreviate the subformula $\nu F.\lambda f.(f\, p) \wedge \langle b \rangle (F\,(f \circ f))$.

Only state 1 satisfies φ. Now note that F is a second-order fixpoint taking as arguments a term interpreted as a first-order function of the kind $2^{[n]} \to 2^{[n]}$. Hence, even for $n = 2$, there already are 256 of them, and naïve fixpoint iteration would tabulate all of them first before computing the values of F on them. On the other hand, all that is needed is F's value on functions $\langle\langle a \rangle\rangle^{2^i}$ where $\langle\langle a \rangle\rangle(S) = \{t \in [n] \mid \exists t \in [n] \text{ s.t. } s \xrightarrow{a} t\}$. The following puts the number of such different functions which are being discovered by local fixpoint iteration in relation to the number of otherwise possible function argument.

n	2	3	4	5	6	7	…
possible arguments to F	256	$1.6 \cdot 10^6$	$1.8 \cdot 10^{19}$ $1.5 \cdot 10^{48}$	$3.9 \cdot 10^{115}$	…	…	…
discovered in local iteration	2	2	2	3	3	4	…

The numbers can be verified either through manual computation of the functions $\langle\langle a \rangle\rangle^{2^i}$ for $i = 0, 1, \ldots$ on each $\mathscr{T}_n$ or using the implementation of Alg. 1 mentioned in the conclusion below.

Abstract interpretation of functional languages. Strictness analysis for (lazy) functional languages tries to figure out whether an argument to a function must always be evaluated. In this case compilers may force the evaluation of the argument thus saving space and time to create closures and allowing for parallelisation. Strictness analysis may be formulated as an abstract interpretation as e.g. in [11]. A function $f \colon D \times D \times \cdots \times D \to D$ is *strict* in its i-th argument, when $f(d_1, d_2, \ldots, d_{i-1}, \bot_D, d_{i+1}, \ldots d_k) = \bot_D$ for a concrete base domain D. As this may be uncomputable, in [11], functions are interpreted *abstractly* over the domain $\mathbf{2} := \{\mathbf{0}, \mathbf{1}\}$ (with $\mathbf{0} \sqsubseteq \mathbf{1}$), where $\mathbf{0}$ means *definitely undefined*, and where $\mathbf{1}$ means *might be defined*. Examples of abstract interpretations of common base values are (for $x, y, z \in \mathbf{2}$),

- constants of base domains such as integers or boolean values are abstracted to $\mathbf{1}$ (not *undefined*);
- first-order functions such as addition are strict in all arguments, e.g., $x+y=\mathbf{0}$ unless $x=y=\mathbf{1}$;
- if-then-else: $\mathsf{ite}(x,y,z)=x\wedge(y\vee z)$, where elements of $\mathbf{2}$ are read as boolean values. If-then-else might only be defined if both the condition and at least one of the then-else arguments might be defined. Otherwise it is definitely undefined. Note that, in this example, we use ite in the traditional sense of functional programming.

As an example of the application of EVAL to the abstract interpretation of functional languages, we choose $\mathscr{F} := \{\mathsf{ite}\colon \blacklozenge^+\times\blacklozenge^+\times\blacklozenge^+\to\blacklozenge\}$ and $\mathsf{Var}=\{x\colon \blacklozenge,\ f\colon \blacklozenge^+\to\blacklozenge, p\colon \blacklozenge^+\to\blacklozenge,\ I\colon (\blacklozenge^+\to\blacklozenge)^+\times(\blacklozenge^+\to\blacklozenge)^+\times\blacklozenge^+\to\blacklozenge\}$. Consider the term $\varphi \ := \ \mu I(f,p,x).\,\mathsf{ite}(p(x),I(f,p,f(x)),x)$. It essentially describes an iterated application of some function f until a predicate p holds. In order to show that φ is strict in x for given functions f_0 and p_0, one needs to evaluate $\varphi(f_0,p_0,\mathbf{0})$ by fixpoint unfolding and β-reduction. If $p_0(\mathbf{0})=\mathbf{0}$, that is, the termination predicate is itself strict, then EVAL terminates in one step proving strictness of φ in its third argument. If $p_0(\mathbf{0})=\mathbf{1}$, that is, p is essentially a constant true or constant false predicate, we need to evaluate $\mathbf{1}\wedge(\varphi(f_0,p_0,f_0(\mathbf{0}))\vee\mathbf{0})=\varphi(f_0,p_0,f_0(\mathbf{0}))$ next. If $f_0(\mathbf{0})=\mathbf{0}$, that is, f_0 is strict itself, we have reached a fixpoint and can conclude strictness in x as well. If, however, $f_0(\mathbf{0})=\mathbf{1}$, we obtain an overall result of $\mathbf{1}$, not showing strictness in x. This is plausible for constant functions f and p. Using local iteration this is in fact the only computation that takes place, whereas a naïve global fixpoint computation would start by tabulating all possible triples of type $(\blacklozenge^+\to\blacklozenge)^+\times(\blacklozenge^+\to\blacklozenge)^+\times\blacklozenge^+$, which, for the lattice $\mathbf{2}$, amounts to $4\cdot4\cdot2=32$ table columns.

5 Limitations of Neededness Analysis and Optimisation

As mentioned in the introduction to Sect. 3, Algorithm EVAL does not use local evaluation on operand-side subterms but rather computes their value fully. If such an operand has a function type, its value on all its arguments might not be needed either. Consider the first example from Sect. 4 about formal-language constrained reachability problems. Clearly, the values of the order-1-functions stored in the parameters f and g are not needed at most arguments. Hence, computing their value fully appears to be wasteful.

Algorithm EVAL computes values of operand-side subterms fully due to the termination criterion for the computation of fixpoint terms: iteration stops when both no new argument tuples have been discovered during a round of the repeat-loop computing the semantics, and the value of the fixpoint in question is stable on all existing tuples. This, of course, requires some way of deciding whether a discovered argument is actually new. Going back to the example in Sect. 4, Algorithm EVAL successively discovers the argument tuples $[a,b,c(0)],[a^2,b^2,c^2(0)],\ldots$ Eventually, these argument tuples begin to repeat, which is when the loop terminates. However, deciding whether e.g. $[a^i,b^i,c^i(0)]$ is the same argument tuple as $[a^j,b^j,c^j(0)]$ requires knowing the value of the function type arguments at all *their* arguments. One could assume that it is enough to know just their value on arguments actually needed in the iteration, but this approach fails readily: already for $[a,b,c(0)]$ and $[a^2,b^2,c^2(0)]$, for $n\geq 2$, we see that $c^i(0)=3n-(i+1)$ for $i\in\{1,2\}$, whence $a^i(b^i(c^i(0)))=\bot$ for either i, and, in fact, all $i\leq n(n-1)(n-2)$, since these differ on hitherto undiscovered arguments. Hence, any algorithm that tests equality of function type arguments only on tuples already identified as necessary for the computation must fail here. Moreover, since the base functions a,b,c are actually interpreted, instead of e.g. tree constructors as in the case of higher-order model checking, a simple flow analysis (e.g. 0-CFA) fails to detect which functions are duplicates unless one also inspects the behavior of the base functions. Hence, safe approach to avoid the error sketched above is to compute values of argument-side functions – which are necessarily not of the highest type order occurring in the term under consideration – in full.

However, this does not mean that this is always necessary. In the example from Sect. 4, one can readily see that the value of e.g. f will always be $\bot$ on all arguments that are not in $\{0,\dots,n-1\}$, since f only contains powers of a. This kind of domain-specific approach, together with e.g. flow analysis and the choice of an appropriate lattice, could be used to cut down the amount of computations necessary.

6 Conclusion

We have lifted the notion of local fixpoint iteration, resp. neededness analysis, for the evaluation of first-order fixpoint functions to fixpoint functions at arbitrary higher order. For generality purposes we have defined an abstract algebra $\mu\mathsf{HO}$ combining a simply typed λ-calculus over (possibly higher-order) base functions with fixpoints at arbitrary type orders. The examples in Sect. 4 show that this can vastly reduce the number of values that are being computed in fixpoint iterations, compared to the naïve global approach.

A conceptual implementation of $\mu\mathsf{HO}$ and Alg. 1 is available.[4] It does not compete with specialised tools like higher-order model checkers but rather focuses on displaying the effect that local fixpoint iteration has in comparison to global iteration for higher-order fixpoints.

Work on fixpoint iteration for higher-order functions can be continued in several directions. The most pressing issue is an extension to *fully* local fixpoint iteration, which would also employ local evaluation at orders beneath the top one, bearing in mind the obstacles to overcome which have been discussed in Sect. 5. Significant progress on this front likely requires giving up the full genericity of the algorithm. For example, many intersection-type based HORS model checkers (e.g. [9, 30]) require backwards reasoning alongside the base functions. For example, acceptance of an automaton in a node of a tree depends on its children, i.e. the arguments to the tree constructor in question, and the relationship is readily available. Conversely, in the present form, our algorithm makes no assumptions on the (behavior of) the base functions, whence it can not infer which values of a given argument might yield a desired function value.

The acute reader may have wondered why $\mu\mathsf{HO}$ does not feature operators $\sqcup,\sqcap$ for suprema and infima at arbitrary types. It would in fact be possible to add these, and algorithm EVAL can be extended accordingly to handle them just like other base functions are being handled. They are not included in the syntax of $\mu\mathsf{HO}$ here for the following reason: when $\sqcup,\sqcap$ are present in the syntax one would expect the distributivity laws like $\varphi\sqcup(\psi\sqcap\chi)\equiv(\varphi\sqcap\psi)\sqcup(\varphi\sqcap\chi)$ to hold. But in arbitrary lattices, such laws do not necessarily hold; they only do in distributive latttices. In order not to confuse the issue or make false assumptions we therefore prefer to introduce $\sqcup,\sqcap$ as base functions when necessary and appropriate. This prevents us from restricting the semantics of $\mu\mathsf{HO}$ terms to distributive lattices only. Note that the lattices depicted in Figs. 3 and 4 are not distributive.

Algorithm EVAL makes no assumptions on the order in which needed arguments are evaluated. In data flow analyses, giving precedence to the arguments in the form of heuristics has turned out to be beneficial for efficiency purposes, c.f. [28, Chp. 6]. It remains to be seen whether such heuristics can be extended to higher orders as well.

Most static program analyses in abstract interpretation work with rather rich lattices as base domains which cannot be cast into the scheme of a simply typed λ-calculus over a single base type $\blacklozenge$ as it is used here. We remark, though, that an extension to a many-sorted logic over several base types is straightforward, not only regarding the type system but, most importantly, algorithm EVAL. The same holds for product types on the right of function arrows. It then remains to be seen how far the type system can be enriched without seriously interfering with the ability to evaluate higher-order fixpoints locally.

[4] `https://github.com/muldvarp/LocalHOFPIter`

References

[1] S. Abiteboul, R. Hull & V. Vianu (1995): *Foundations of Databases*. Addison-Wesley.

[2] R. Axelsson & M. Lange (2007): *Model Checking the First-Order Fragment of Higher-Order Fixpoint Logic*. In: *Proc. 14th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR'07*, *LNCS* 4790, Springer, pp. 62–76, doi:10.1007/978-3-540-75560-9_7.

[3] R. Axelsson & M. Lange (2011): *Formal Language Constrained Reachability and Model Checking Propositional Dynamic Logics*. In: *Proc. 5th Workshop on Reachability Problems, RP'11*, *LNCS* 6945, Springer, pp. 45–57, doi:10.1007/978-3-642-24288-5_6.

[4] R. Axelsson, M. Lange & R. Somla (2007): *The Complexity of Model Checking Higher-Order Fixpoint Logic*. *Logical Methods in Computer Science* 3, pp. 1–33, doi:10.2168/LMCS-3(2:7)2007.

[5] C. Baier & J.-P. Katoen (2008): *Principles of model checking*. MIT Press.

[6] P. Barceló, L. Libkin, A. W. Lin & P. T. Wood (2012): *Expressive Languages for Path Queries over Graph-Structured Data*. *ACM Trans. Database Syst* 37(4), pp. 31:1–31:46, doi:10.1145/2389241.2389250.

[7] C. Barrett, R. Jacob & M. Marathe (2000): *Formal-Language-Constrained Path Problems*. *SIAM Journal on Computing* 30(3), pp. 809–837, doi:10.1137/S0097539798337716.

[8] G. Bhat & R. Cleaveland (1996): *Efficient Local Model-Checking for Fragments of the Modal μ-Calculus*. In: *Proc. 2nd Int. Workshop on Tools and Algorithms for Construction and Analysis of Systems, TACAS'96*, *LNCS* 1055, Springer, pp. 107–126, doi:10.1109/LICS.1996.561358.

[9] C. H. Broadbent & N. Kobayashi (2013): *Saturation-Based Model Checking of Higher-Order Recursion Schemes*. In S. Ronchi Della Rocca, editor: *Computer Science Logic 2013 (CSL 2013), CSL 2013, September 2-5, 2013, Torino, Italy*, *LIPIcs* 23, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 129–148, doi:10.4230/LIPIcs.CSL.2013.129.

[10] F. Bruse, M. Lange & É. Lozes (2017): *Space-Efficient Fragments of Higher-Order Fixpoint Logic*. In: *Proc. 11th Workshop on Reachability Problems, RP'17*, *LNCS* 10506, Springer, pp. 26–41, doi:10.1007/978-3-319-67089-8_3.

[11] G. L. Burn, C. Hankin & S. Abramsky (1986): *Strictness analysis for higher-order functions*. *Science of computer programming* 7, pp. 249–278, doi:10.1016/0167-6423(86)90010-9.

[12] B. Le Charlier & P. Van Hentenryck (1993): *Groundness analysis for Prolog: implementation and evaluation of domain prop*. In: *Proc. ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'93*, pp. 99–110, doi:10.1145/154630.154641.

[13] P. Cousot & R. Cousot (1977): *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*. In: *Proc. 4th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages, POPL'77*, pp. 238–252, doi:10.1145/512950.512973.

[14] P. Cousot & R. Cousot (1994): *Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages)*. In: *Proc. Int. Conf. on Computer Languages, ICCL'94*, IEEE, pp. 95–112, doi:10.1109/ICCL.1994.288389.

[15] S. Demri, V. Goranko & M. Lange (2016): *Temporal Logics in Computer Science*. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, doi:10.1017/CBO9781139236119.

[16] D. Distefano, P. W. O'Hearn & H. Yang (2006): *A local shape analysis based on separation logic*. In: *Proc. 12th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'06*, Springer, pp. 287–302, doi:10.1007/11691372_19.

[17] E. A. Emerson & E. M. Clarke (1980): *Characterizing Correctness Properties of Parallel Programs Using Fixpoints*. In J. W. de Bakker & J. van Leeuwen, editors: *Proc. 7th Int. Coll. on Automata, Languages and Programming, ICALP'80*, *LNCS* 85, Springer, pp. 169–181, doi:10.1007/3-540-10003-2_69.

[18] C. Fecht & H. Seidl (1996): *An even faster solver for general systems of equations*. In: *Proc. 3rd Int. Static Analysis Symp., SAS'96*, Springer, pp. 189–204, doi:10.1007/3-540-61739-6_42.

[19] N. Immerman (1999): *Descriptive complexity*. Graduate texts in computer science, Springer, doi:10.1007/978-1-4612-0539-5.

[20] N. Jørgensen (1994): *Finding Fixpoints in Finite Function Spaces using Neededness Analysis and Chaotic Iteration*. In: *Proc. 1st Int. Static Analysis Symposium, SAS'94*, *LNCS* 864, Springer, pp. 329–345, doi:10.1007/3-540-58485-4_50.

[21] S. C. Kleene (1938): *On Notation for Ordinal Numbers*. *Journal Symbolic Logic* 3(4), pp. 150–155, doi:10.2307/2267778.

[22] N. Kobayashi, É. Lozes & F. Bruse (2017): *On the relationship between higher-order recursion schemes and higher-order fixpoint logic*. In: *Proc. 44th ACM SIGPLAN Symp. on Principles of Programming Languages, POPL'17*, ACM, pp. 246–259, doi:10.1145/3093333.3009854.

[23] D. Kozen (1983): *Results on the Propositional μ-calculus*. *TCS* 27, pp. 333–354, doi:10.1016/0304-3975(82)90125-6.

[24] M. Lange & É. Lozes (2015): *Conjunctive Visibly-Pushdown Path Queries*. In: *Proc. 20th Int. Symp. on Fundamentals of Computation Theory, FCT'15*, *LNCS* 9210, Springer, pp. 327–338, doi:10.1007/978-3-319-22177-9_25.

[25] T. Lev-Ami & M. Sagiv (2000): *TVLA: A system for implementing static analyses*. In: *Proc. 7th Int. Static Analysis Symp., SAS'00*, Springer, pp. 280–301, doi:10.1007/978-3-540-45099-3_15.

[26] A. N. Maslov (1974): *The hierarchy of indexed languages of an arbitrary level*. *Dokl. Akad. Nauk SSSR* 217, pp. 1013–1016.

[27] A. Mycroft (1980): *The theory and practice of transforming call-by-need into call-by-value*. In: *Proc. 4th Int. Symp. on Programming*, Springer, pp. 269–281, doi:10.1007/3-540-09981-6_19.

[28] F. Nielson, H. R. Nielson & C. Hankin (1999): *Principles of program analysis*. Springer, doi:10.1007/978-3-662-03811-6.

[29] C.-H. L. Ong (2006): *On Model-Checking Trees Generated by Higher-Order Recursion Schemes*. In: *Proc. 21st IEEE Symp. on Logic in Computer Science, LICS'06*, IEEE Computer Society, pp. 81–90, doi:10.1109/LICS.2006.38.

[30] S. J. Ramsay, R. P. Neatherway & C.-H. Luke Ong (2014): *A type-directed abstraction refinement approach to higher-order model checking*. In: *Proc. 41st Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'14*, ACM, pp. 61–72, doi:10.1145/2535838.2535873.

[31] D. Simovici & R. L. Tenney (1999): *Theory of Formal Languages with Applications*. World Scientific, doi:10.1142/3991.

[32] A. Tarski (1955): *A Lattice-theoretical Fixpoint Theorem and its Application*. *Pacific Journal of Mathematics* 5, pp. 285–309, doi:10.2140/pjm.1955.5.285.

[33] M. Viswanathan & R. Viswanathan (2004): *A Higher Order Modal Fixed Point Logic*. In: *Proc. 15th Int. Conf. on Concurrency Theory, CONCUR'04*, *LNCS* 3170, Springer, pp. 512–528, doi:10.1007/978-3-540-28644-8_33.

[34] M. Wand (1974): *An algebraic formulation of the Chomsky hierarchy*. In E. G. Manes, editor: *Category Theory Applied to Computation and Control, Proceedings of the First International Symposium, San Francisco, CA, USA, February 25-26, 1974, Proceedings*, *Lecture Notes in Computer Science* 25, Springer, pp. 209–213, doi:10.1007/3-540-07142-3_84.

[35] G. Winskel (1993): *The Formal Semantics of Programming Languages: An Introduction*. Foundations of Computing series, MIT Press, doi:10.7551/mitpress/3054.001.0001.