Metric Spaces and Termination Analyses

Aziem Chawdhary[1,2] and Hongseok Yang[2]

[1] Durham University
[2] Queen Mary University of London

Abstract. We present a framework for defining abstract interpreters for liveness properties, in particular program termination. The framework makes use of the theory of metric spaces to define a concrete semantics, relates this semantics with the usual order-theoretic semantics of abstract interpretation, and identifies a set of conditions for determining when an abstract interpreter is sound for analysing liveness properties. Our soundness proof of the framework is based on a novel relationship between unique fixpoints in metric semantics and post-fixpoints computed by abstract interpreters. We illustrate the power of the framework by providing an instance that can automatically prove the termination of programs with general (not necessarily tail) recursion.

1 Introduction

Recently, there has been great interest in the automatic verification of program termination. Quite a few techniques for automatically verifying termination or general liveness properties of imperative programs have been proposed [1, 2, 4–8, 16–18], some of which have led to successful tools, such as TERMINATOR [7].

In this paper, we step back from all these technological advances, and re-examine a theoretical foundation of automatic techniques for verifying termination or liveness properties of programs. Most of the proposed techniques are based on abstracting programs (in addition to clever results on well-founded relations such as [3, 19]), but these abstraction methods are justified by rather ad-hoc arguments [4]. This is in contrast with the soundness of abstraction for safety properties, which follows a standard framework of abstract interpretation [10, 11]. Our aim is to develop a theory that provides a similar systematic answer for when an abstraction is sound for proving liveness properties. By doing so, we want to relieve the burden of inventing a new way of proving soundness from designers of liveness analysis.

Our main result is a new framework for developing sound precise abstract interpreters for liveness properties of programs with general recursion. Technically, the key feature of our framework is to use a concrete semantics based on metric space [13, 14, 20] and to spell out a condition under which this concrete metric-space semantics can be related to a usual order-theoretic semantics of abstract interpretation. We illustrate the power of the framework by providing an instance that can automatically prove the termination of recursive procedures.

Our framework uses a metric-space semantics, because such a semantics justifies a novel strategy for computing approximate fixpoints during abstract interpretation for *liveness*. Imagine that we want to develop a sound termination analysis. Our analysis needs to overapproximate the set of all computation traces of a given program and to check whether the overapproximation does not include an infinite trace. In the standard order-theoretic setting, the set of computation traces of a program is defined in terms of the greatest fixpoint of some function F [9], but overapproximating the greatest fixpoint of F precisely wrt. termination is difficult. For instance, a post-fixpoint x of F (i.e., $F(x) \sqsubseteq x$), which is normally computed by an abstract interpreter for safety, does not overapproximate the greatest fixpoint in general. Hence, fixpoint-computation strategies from safety analyses cannot be used for termination analysis without changes. Alternatively, one might consider the following sequence converging to the greatest fixpoint of F (under the assumption of the continuity of F):

$$\top \sqsupseteq F(\top) \sqsupseteq F^2(\top) \sqsupseteq F^3(\top) \sqsupseteq \ldots$$

and want to compute an overapproximating sequence $\{x_n\}$ such that $F^n(\top) \sqsubseteq x_n$ for all n, and $x_m = x_{m+1}$ for some m. In this case, a fixpoint-computation strategy finds this x_m, and returns it as a result. The problem here is that the strategy is very imprecise; it cannot prove termination of most nontrivial programs (especially those whose time complexity is not constant).

The metric-space semantics of our framework resolves this overapproximation issue. It defines the set of computation traces of a program in terms of a *unique fixpoint* of a function G, and then it guarantees that this unique fixpoint can be overapproximated by a post-fixpoint of G, as long as the post-fixpoint lives in a restricted semantic universe, such as the one with the *closed* sets of traces.[3] Thus, when developing a sound termination analysis in our framework, one can re-use fixpoint-computation strategies from existing safety analyses (which compute post-fixpoints), after adjusting the strategies so that computed post-fixpoints live in the restricted universe.

Using a metric-space semantics has another benefit that our framework can hide call stacks, which appear in a small-step operational semantics of recursive procedures. Hence, a user of the framework does not need to worry about abstracting call stacks [15], and can focus on the problem of proving a desired liveness property.

Related Work Among the automatic techniques for proving program termination cited already, we discuss two techniques further [4, 8]. The first is our previous work [4], where we proved the soundness of a termination analysis, by directly relating greatest fixpoints in the concrete trace semantics with post-fixpoints computed by the termination analysis. Our proof relied on the fact that the language contained only tail recursions so that greatest fixpoints could be

[3] A trace set is closed iff all Cauchy sequences in the set have limits in the set. We will explain it further in the main part of the paper.

$$e ::= x \mid r \mid e + e \mid r \times e \qquad b ::= e = e \mid e \neq e \mid e \leq e \mid e < e \mid b \wedge b \mid b \vee b \mid \neg b$$
$$c ::= x := e \mid c; c \mid \texttt{if}\, b\, c\, c \mid f() \mid \texttt{fix}\, f.\, c$$

Fig. 1. Programming Language with General Recursion

rephrased in terms of least fixpoints and infinite iterations. This rewriting is not applicable if a programming language includes non-tail recursions. In contrast, the framework of this paper can handle programs with general recursion.

The second technique is a termination analysis for recursive procedures in [8]. This technique works by replacing each recursive function call by a non-deterministic choice between entering a procedure body (in the case that the procedure does not terminate) or the application of a summary of the procedure (in the case that the procedure does terminate). The instance of our framework in this paper can be seen as a modified version of this technique where program transformations are done on the fly and termination proofs and procedure summarizations are done at the same time.

Recently Cousot et al. [12] defined bi-inductive domains to account for both infinite and finite program properties. They combine a domain for finite behaviours with another for infinite behaviours, and produce a new domain whose order is defined using the orders from the two underlying domains. A least fixpoint on this new domain can overapproximate the union of the least fixpoint in the finite domain and the greatest fixpoint in the infinite domain. However, the semantic functions may not be monotone with respect to the order of the new domain, and so cannot be computed by the usual fixpoint iteration. This limitation means that we once again have to reason about least and greatest fixpoints, a situation that we avoid in this paper by using metric spaces.

2 Programming Language

Let PName be the set of procedures names, ranged over by f, g, and let Var be a finite set of program variables x, y that contain rational numbers in $\mathbb{Q}$. We consider a simple imperative language with parameterless procedures f, g and rational variables x, y. The grammar of the language is given in Fig. 1, where we use r to denote a rational constant.

Most commands in our language are standard. The only unusual case is the definition of recursive procedure $\texttt{fix}\, f.\, c$. It defines a recursive procedure f whose body is c, and then it immediately calls the defined procedure. Note that while loops can be expressed in this language using recursion. We write $\Gamma \vdash c$ for a finite subset Γ of PName, where Γ includes all the free function names in c.

3 Framework

In this section we describe our framework for developing a sound abstract interpreter for liveness properties. Throughout the paper, we will use $\mathbb{N}$ for the set of *positive* integers.

3.1 Review on Metric Spaces

We start with a brief review on metric spaces. For further information on metric semantics, we refer the reader to the standard book and survey on this topic [13, 20].

A **metric space** is a non-empty set X with a function $d_X : X \times X \to [0, \infty)$, called metric, that satisfies the three conditions below:

1. Identity of indiscernible: $\forall x, y \in X.\ d_X(x, y) = 0 \iff x = y$.
2. Symmetry: $\forall x, y \in X.\ d_X(x, y) = d_X(y, x)$.
3. Triangular inequality: $\forall x, y, z \in X.\ d_X(x, z) \leq d_X(x, y) + d_X(y, z)$.

Consider a sequence $\{x_n\}_{n\in\mathbb{N}}$ in a metric space (X, d_X). The sequence $\{x_n\}_{n\in\mathbb{N}}$ is **Cauchy** iff for all real numbers $\epsilon > 0$, there exists some $N \in \mathbb{N}$ such that $\forall m, n \geq N.\ d_X(x_m, x_n) \leq \epsilon$. The sequence $\{x_n\}_{n\in\mathbb{N}}$ **converges to** x **in** X iff for all real numbers $\epsilon > 0$, there exists an $N \in \mathbb{N}$ such that $\forall m \geq N.\ d_X(x_m, x) \leq \epsilon$.

A metric space X is **complete** iff every Cauchy sequence converges to some element in X. In this paper, we will consider only complete metric spaces.

Let (X, d_X) and (Y, d_Y) be metric spaces and let α be a positive real number. A function $F : X \to Y$ is **non-expansive** iff for all $x, x' \in X$, we have that $d_Y(F(x), F(x')) \leq d_X(x, x')$. It is **$\alpha$-contractive** iff $d_Y(F(x), F(x')) \leq \alpha \times d_X(x, x')$ holds for all $x, x' \in X$. Intuitively, the non-expansiveness means that F does not increase the distance between elements, and the contractiveness says that F actually decreases the distance.

In this paper, we use the well-known Banach's unique fixpoint theorem:

Theorem 1 (Banach's Unique Fixpoint Theorem). *Let (X, d_X) be a metric space. If X is complete and a function $F : X \to X$ is α-contractive for some $0 \leq \alpha < 1$, the function F has the unique fixpoint. Furthermore, this unique fixpoint can be obtained as follows: first pick an arbitrary x_1 in X, then construct the sequence $\{x_n\}_{n\in\mathbb{N}}$ with $x_{n+1} = F(x_n)$ and finally take the limit of this sequence.*[4]

We will denote the unique fixpoint of F by $\mathsf{ufix}(F)$.

3.2 Concrete Metric-Space Semantics

Our framework consists of two parts. The first part is a concrete semantics based on metric spaces. It is parameterized by the data below, which should be provided by a user of the framework:

[4] This limit always exists, because the constructed sequence is Cauchy.

1. A pre-ordered complete metric space $(\mathcal{D}, d, \sqsubseteq, \top)$ with the biggest element $\top$. We require that for all Cauchy sequences $\{x_n\}_{n\in\mathbb{N}}$ in $\mathcal{D}$ and all $x \in \mathcal{D}$,
$$(\forall n \in \mathbb{N}.\ x_n \sqsubseteq x) \implies \lim_{n\to\infty} x_n \sqsubseteq x. \tag{1}$$
Elements of $\mathcal{D}$ can be understood as semantic counterparts of syntactic commands; our concrete semantics interprets a command c as an element in $\mathcal{D}$.
2. Monotone non-expansive functions seq, $\mathsf{asgn}_{x,e}$ and if_b for all assignments $x{:=}e$ and all boolean conditions b:
$$\mathsf{seq} : \mathcal{D} \times \mathcal{D} \to \mathcal{D}, \qquad \mathsf{asgn}_{x,e} : \mathcal{D}, \qquad \mathsf{if}_b : \mathcal{D} \times \mathcal{D} \to \mathcal{D}.$$
These functions define the meaning of the sequencing, assignment and conditional statements in our language.
3. A function $\mathsf{proc} : \mathsf{PName} \to \mathcal{D} \to \mathcal{D}$ for modelling the execution of procedures. We write proc_f instead of $\mathsf{proc}(f)$, and require that $\mathsf{proc}_f(-)$ be a monotone $\frac{1}{2}$-contractive function for all $f \in \mathsf{PName}$. Intuitively, an input x to $\mathsf{proc}_f(-)$ denotes all the possible computations by the body of the procedure f, and $\mathsf{proc}_f(x)$ extends each of these computations with steps taken immediately before or after running the procedure body during the call of f.
4. A subset LivProperty of $\mathcal{D}$ that is downward closed with respect to $\sqsubseteq$:
$$x \sqsubseteq y \ \wedge\ y \in \textsc{LivProperty} \implies x \in \textsc{LivProperty}.$$
This subset consists of elements in $\mathcal{D}$ (which are semantic counterparts of commands) satisfying a desired liveness property, such as termination.

Note that the semantic domain $\mathcal{D}$ here has both pre-order and metric-space structures and that the semantic operators respect both structures by being monotone and non-expansive. These two structures are related by the requirement (1) on $\sqsubseteq$ and Cauchy sequences. One important consequence of the relationship is the lemma below, and it will play a crucial role for the soundness of our framework:

Lemma 1. *For all $\frac{1}{2}$-contractive monotone functions $F : \mathcal{D} \to \mathcal{D}$, a post-fixpoint of F overapproximates the unique fixpoint of F. That is, if x satisfies $F(x) \sqsubseteq x$, we have that $\mathsf{ufix}\, F \sqsubseteq x$, where $\mathsf{ufix}\, F$ is the unique fixpoint of F.*

Proof. Let x be a post-fixpoint of F. By the Banach fixpoint theorem, we know that the unique fixpoint $\mathsf{ufix}\, F$ of F exists and is also the limit of the following Cauchy sequence:
$$x,\ F(x),\ F^2(x),\ F^3(x),\ \ldots$$
Since x is a post-fixpoint of F (i.e., $F(x) \sqsubseteq x$) and F is monotone,
$$x \ \sqsupseteq\ F(x) \ \sqsupseteq\ F^2(x) \ \sqsupseteq\ F^3(x) \ \sqsupseteq\ F^4(x)\ \ldots$$
That is, $F^n(x) \sqsubseteq x$ for all n. Thus, the limit $\mathsf{ufix}\, F$ of $\{F^n(x)\}_{n\in\mathbb{N}}$ also satisfies $\mathsf{ufix}\ F \sqsubseteq x$ by the requirement (1) of our framework. We have just proved the lemma. □

$$[\![\Gamma \vdash c]\!] \;:\; [\![\Gamma]\!] \to \mathcal{D}$$

$$[\![\Gamma \vdash f()]\!]\eta = \eta(f) \qquad [\![\Gamma \vdash c_1; c_2]\!]\eta = \mathsf{seq}([\![\Gamma \vdash c_1]\!]\eta, [\![\Gamma \vdash c_2]\!]\eta)$$

$$[\![\Gamma \vdash x{:=}e]\!]\eta = \mathsf{asgn}_{x,e} \qquad [\![\Gamma \vdash \mathtt{if}\, b\, c_1\, c_2]\!]\eta = \mathsf{if}_b([\![\Gamma \vdash c_1]\!]\eta, [\![\Gamma \vdash c_2]\!]\eta)$$

$$[\![\Gamma \vdash \mathtt{fix}\, f.c]\!]\eta = \mathsf{ufix}\, F \qquad (\text{where } F(x) = \mathsf{proc}_f([\![\Gamma, f \vdash c]\!]\eta[f \mapsto x]))$$

Fig. 2. Concrete Semantics defined by the Framework

The domain $\mathcal{D}$ and the operators above give rise to a metric-space semantics of programs. Let $[\![\Gamma]\!]$ be the domain for procedure environments (i.e., $\Pi_{f \in \Gamma}\mathcal{D}$), pre-ordered pointwise and given the product metric, where the distance between η and η' in $\Pi_{f \in \Gamma}\mathcal{D}$ is given by $\max_{f \in \Gamma} d(\eta(f), \eta'(f))$. The semantics interprets $\Gamma \vdash c$ as a non-expansive map from $[\![\Gamma]\!]$ to $\mathcal{D}$, and it is given in Fig. 2.

Note that the semantics defines $\mathtt{fix}\, f.c$ as the unique fixpoint of a function F modelling the meaning of the procedure body c. To ensure the existence of the fixpoint here, the semantics maintains that all commands denote only non-expansive functions. Then, it defines the function F in terms of non-expansive $[\![\Gamma, f \vdash c]\!]$ and 1/2-contractive proc_f, and ensures that F is 1/2-contractive. Hence, by the Banach fixpoint theorem, F has the unique fixpoint.

Lemma 2. *For all commands $\Gamma \vdash c$, $[\![\Gamma \vdash c]\!]$ is a well-defined non-expansive function from $[\![\Gamma]\!]$ to $\mathcal{D}$. Furthermore, $[\![\Gamma \vdash c]\!]$ is monotone.*

The use of metric spaces means that in order to design an instance of our generic framework one now needs to prove certain properties of the concrete semantics. Firstly, one has to prove that the semantic domain $\mathcal{D}$ for the meaning of commands is a complete metric space, in addition to having a pre-order structure. Secondly, one needs to show that all the semantic operators are non-expansive.

These new proof obligations often make it impossible to re-use an existing concrete semantics. For instance, a naive trace semantics, such as the one in [4], uses the powerset of traces as a semantic universe for commands, but this powerset cannot be used in our framework. This is because it does not form a complete metric space, when it is given a natural notion of distance function. In order to use the framework in this paper, one has to modify the powerset of traces, so that it has a good metric-theoretic structure, as will be done in Sec. 4.

However, these obligations come with a reward—the soundness of an order-theoretic abstract semantics, which is to be presented next.

3.3 Abstract Semantics

The second part of our framework is the abstract semantics. For a function $f : X^n \to X$ and a subset X_0 of X, we say that f can be restricted to X_0 if for all $\boldsymbol{x} \in X_0^n$, we have that $f(\boldsymbol{x}) \in X_0$. Using this terminology, we describe the parameters of our abstract semantics:

1. A set $\mathcal{A}$ with a partition $\mathcal{A}_p \uplus \mathcal{A}_t = \mathcal{A}$. The elements of $\mathcal{A}$ provide abstract meanings of commands. We call elements in $\mathcal{A}_t$ *total* and those in $\mathcal{A}_p$ *partial*. The set $\mathcal{A}$ should come with the additional data below.
   (a) Distinguished elements $\bot$ and $\top$ in $\mathcal{A}$ such that $\top \in \mathcal{A}_t$.
   (b) An algorithm $\mathsf{checktot}$ that answers the membership to $\mathcal{A}_t$ soundly but not necessarily in a complete way. That is, $\mathsf{checktot}(A) = \mathsf{true}$ means that $A \in \mathcal{A}_t$, but $\mathsf{checktot}(A) \neq \mathsf{true}$ does not mean that $A \notin \mathcal{A}_t$.
   (c) A concretization function $\gamma : \mathcal{A}_t \to \mathcal{D}$, such that $\gamma(\top) = \top$. Note that the domain of γ is $\mathcal{A}_t$, not $\mathcal{A}$.
2. Functions $\mathsf{seq}^\sharp$, $\mathsf{asgn}^\sharp_{x,e}$ and $\mathsf{if}^\sharp_b$ for all assignments $x{:=}e$ and booleans b:

$$\mathsf{seq}^\sharp : \mathcal{A} \times \mathcal{A} \to \mathcal{A}, \qquad \mathsf{asgn}^\sharp_{x,e} : \mathcal{A}, \qquad \mathsf{if}^\sharp_b : \mathcal{A} \times \mathcal{A} \to \mathcal{A}.$$

   These functions give the abstract meaning of the sequencing, assignment and conditional statements in our language. We require that these functions can be restricted to $\mathcal{A}_t$, and that they overapproximate their concrete counterparts:

$$\begin{aligned} \forall A_0, A_1 \in \mathcal{A}_t.\ & \mathsf{seq}(\gamma(A_0), \gamma(A_1)) \ \sqsubseteq\ \gamma(\mathsf{seq}^\sharp(A_0, A_1)) \\ \wedge\quad & \mathsf{asgn}_{x,e} \ \sqsubseteq\ \gamma(\mathsf{asgn}^\sharp_{x,e}) \\ \wedge\quad & \mathsf{if}_b(\gamma(A_0), \gamma(A_1)) \ \sqsubseteq\ \gamma(\mathsf{if}^\sharp_b(A_0, A_1)). \end{aligned}$$

   Note that this soundness condition is only relevant for total elements in $\mathcal{A}_t$.
3. A function $\mathsf{proc}^\sharp : \mathsf{PName} \to \mathcal{A} \to \mathcal{A}$ for modelling the execution of procedures. For all $f \in \mathsf{PName}$, we require that $\mathsf{proc}^\sharp_f$ can be restricted to $\mathcal{A}_t$, and that it should overapproximate proc_f:

$$\forall f \in \mathsf{PName}.\quad \forall A \in \mathcal{A}_t.\quad \mathsf{proc}_f(\gamma(A)) \ \sqsubseteq\ \gamma(\mathsf{proc}^\sharp_f(A)).$$

4. A predicate $\textsc{satisfyLiv}^\sharp$ on $\mathcal{A}_t$ such that

$$\forall A \in \mathcal{A}_t.\quad \textsc{satisfyLiv}^\sharp(A) = \mathsf{true} \implies \gamma(A) \in \textsc{LivProperty}.$$

   Intuitively, $\textsc{satisfyLiv}^\sharp$ identifies abstract elements denoting commands with a desired liveness property.
5. A widening operator $\triangledown : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ [10]. This operator needs to satisfy three conditions. Firstly, it can be restricted to a map from $\mathcal{A}_t$. Secondly, it overapproximates an upper bound of its right argument: $\gamma(A_2) \sqsubseteq \gamma(A_1 \triangledown A_2)$ for all $A_1, A_2 \in \mathcal{A}_t$. Finally, it turns any sequences in $\mathcal{A}$ into one with a stable element. That is, for all $\{A_n\}_{n\in\mathbb{N}}$ in $\mathcal{A}$, the widened sequence $\{A'_n\}_{n\in\mathbb{N}}$ with $A'_1 = A_1$ and $A'_{n+1} = A_n \triangledown A_{n+1}$ contains an index m with $A'_m = A'_{m+1}$.

Note that among the abstract elements in $\mathcal{A}$, only total ones in $\mathcal{A}_t$ have meanings in the concrete domain $\mathcal{D}$ via γ. That is, elements in $\mathcal{A}_p$ need not be concretizable in $\mathcal{D}$. The absence of the concretization relationship between $\mathcal{A}_p$ and $\mathcal{D}$ is intended, because it allows an analysis designer to use a flexible fixpoint strategy during abstract interpretation. Concretely, even though an abstract interpreter aims to compute a value in $\mathcal{D}$ (more precisely, $\{\gamma(A) \mid A \in \mathcal{A}_t\}$)

$$[\![\Gamma \vdash c]\!]^\sharp \ : \ [\![\Gamma]\!]^\sharp \to \mathcal{A}$$

$$[\![\Gamma \vdash f()]\!]^\sharp\eta^\sharp = \eta^\sharp(f) \qquad [\![\Gamma \vdash c_1; c_2]\!]^\sharp\eta^\sharp = \mathsf{seq}^\sharp([\![\Gamma \vdash c_1]\!]^\sharp\eta^\sharp,\ [\![\Gamma \vdash c_2]\!]^\sharp\eta^\sharp)$$

$$[\![\Gamma \vdash x{:=}e]\!]^\sharp\eta^\sharp = \mathsf{asgn}^\sharp_{x,e} \qquad [\![\Gamma \vdash \mathtt{if}\ b\, c_1\, c_2]\!]^\sharp\eta^\sharp = \mathsf{if}^\sharp_b([\![\Gamma \vdash c_2]\!]^\sharp\eta^\sharp,\ [\![\Gamma \vdash c_2]\!]^\sharp\eta^\sharp)$$

$$[\![\Gamma \vdash \mathtt{fix}\, f.c]\!]^\sharp\eta^\sharp = \lceil \mathsf{widenfix}\ F \rceil \quad (\text{where } F(A) = \mathsf{proc}^\sharp_f([\![\Gamma, f \vdash c]\!]^\sharp\eta^\sharp[f \mapsto A]))$$

Fig. 3. Abstract Semantics defined by the Framework

at the end of a fixpoint computation, it can temporarily step outside of $\mathcal{D}$ and use elements in $\mathcal{A}_p$ during the computation, as long as its final result is an element in $\mathcal{D}$. We found this flexibility very useful for achieving high precision in our framework; in order to have a complete metric-space structure, a concrete domain $\mathcal{D}$ often does not include certain semantic elements, such as the empty set, that could serve as the meaning of intermediate results of a precise fixpoint-computation strategy of an abstract interpreter.

The parameters given above are enough to induce an abstract semantics of programs, but to do so, we need to define two operators using the parameters. The first operator is the ceiling $\lceil - \rceil$, which replaces partial elements by $\top$:

$$\lceil A \rceil \ = \ \textbf{if}\ (\mathsf{checktot}(A) = \mathsf{true})\ \textbf{then}\ A\ \textbf{else}\ \top.$$

The second is the widened fixpoint operator $\mathsf{widenfix}$. Given a function $F : \mathcal{A} \to \mathcal{A}$, the operator constructs the sequence $\{A_n\}_{n\in\mathbb{N}}$ with $A_1 = \bot$ and $A_{n+1} = A_n \triangledown F(A_n)$. Then, it returns the first A_m with $A_m = A_{m+1}$. The condition on $\triangledown$ ensures that such A_m exists.

Let $[\![\Gamma]\!]^\sharp$ be the abstract domain for procedure environments (i.e., $[\![\Gamma]\!]^\sharp = \Pi_{f\in\Gamma}\mathcal{A}$). The abstract semantics interprets programs $\Gamma \vdash c$ as functions from $[\![\Gamma]\!]^\sharp$ to $\mathcal{A}$. The defining clauses in the semantics are given in Fig. 3.

The semantics in Fig. 3 are mostly standard, but the abstract semantics of $\mathsf{fix}\, f.c$ deserves attention. After computing a widened fixpoint, $[\![\Gamma \vdash \mathsf{fix}\, f.c]\!]^\sharp$ checks whether the fixpoint is a total element. If not, $[\![\Gamma \vdash \mathsf{fix}\, f.c]\!]^\sharp$ approximates the fixpoint by $\top$, which should be total by the requirement of the framework. This additional step and the requirements of our framework ensure one important property of the semantics:

Lemma 3. *For all $\Gamma \vdash c$ and $\eta^\sharp \in [\![\Gamma]\!]^\sharp$, if $\eta^\sharp(f) \in \mathcal{A}_t$ for every $f \in \Gamma$, we have that $[\![\Gamma \vdash c]\!]^\sharp\eta^\sharp \in \mathcal{A}_t$.*

Intuitively, the lemma says that $[\![\Gamma \vdash c]\!]^\sharp$ can be restricted to total elements. Using this lemma, we express the soundness of the abstract semantics:

$$\forall \eta^\sharp \in [\![\Gamma]\!]^\sharp.\ \ (\forall f \in \Gamma.\ \ \eta^\sharp(f) \in \mathcal{A}_t) \implies [\![\Gamma \vdash c]\!]\gamma(\eta^\sharp) \sqsubseteq \gamma([\![\Gamma \vdash c]\!]^\sharp\eta^\sharp). \qquad (2)$$

In $\gamma(\eta^\sharp)$ above, we use the componentwise extension of γ to procedure environments. Note that although γ is not defined on partial elements, the soundness claim above is well-formed, because Lemma 3 ensures that $[\![\Gamma \vdash c]\!]^\sharp\eta^\sharp$ is total. We prove the soundness in the next theorem:

Theorem 2. *The abstract semantics is sound. That is,* (2) *holds for all* $\Gamma \vdash c$.

Proof (Sketch). Our proof is by induction on the structure of c. Here we focus on the most interesting case that $c \equiv \mathtt{fix}\, f.c_1$, where we can see the interaction between the metric structure and the pre-order structure of $\mathcal{D}$. Let $F\colon \mathcal{D} \to \mathcal{D}$ and $G\colon \mathcal{A} \to \mathcal{A}$ be functions defined by

$$F(x) = \mathsf{proc}_f([\![\Gamma, f \vdash c_1]\!]\gamma(\eta^\sharp)[f \mapsto x]),\ \ G(A) = \mathsf{proc}^\sharp_f([\![\Gamma, f \vdash c_1]\!]^\sharp\eta^\sharp[f \mapsto A]).$$

We need to prove that

$$(\mathsf{ufix}\ F) \ \sqsubseteq\ \gamma(\lceil \mathsf{widenfix}\ G \rceil). \tag{3}$$

If $\mathsf{checktot}(\mathsf{widenfix}\ G) \neq \mathsf{true}$, then $\gamma(\lceil \mathsf{widenfix}\ G \rceil) = \gamma(\top) = \top$. Thus, (3) holds. Suppose that $\mathsf{checktot}(\mathsf{widenfix}\ G) = \mathsf{true}$, which implies that $\mathsf{widenfix}\ G \in \mathcal{A}_t$. In this case, it is sufficient to prove that $\gamma(\mathsf{widenfix}\ G)$ is a post-fixpoint of F. Because then, the inequality (3) follows from Lemma 1. By the definition of $\mathsf{widenfix}$, $(\mathsf{widenfix}\ G) = (\mathsf{widenfix}\ G) \triangledown G(\mathsf{widenfix}\ G)$. Because of the condition on $\triangledown$, this implies that

$$\gamma(G(\mathsf{widenfix}\ G)) \ \sqsubseteq\ \gamma(\mathsf{widenfix}\ G). \tag{4}$$

The LHS of (4) is greater than or equal to $F(\gamma(\mathsf{widenfix}\ G))$ as shown below:

$$\begin{array}{rcl} \gamma(G(\mathsf{widenfix}\ G)) & = & \gamma\big(\mathsf{proc}^\sharp_f\,[\![\Gamma, f \vdash c_1]\!]^\sharp\eta^\sharp[f \mapsto (\mathsf{widenfix}\ G)]\big)\big) \\ & \sqsupseteq & \mathsf{proc}_f\big(\gamma\big([\![\Gamma, f \vdash c_1]\!]^\sharp\eta^\sharp[f \mapsto (\mathsf{widenfix}\ G)]\big)\big) \\ & \sqsupseteq & \mathsf{proc}_f\big([\![\Gamma, f \vdash c_1]\!]\gamma(\eta^\sharp)[f \mapsto \gamma(\mathsf{widenfix}\ G)]\big) \\ & = & F(\gamma(\mathsf{widenfix}\ G)). \end{array} \tag{5}$$

The first inequality holds because $\mathsf{proc}^\sharp$ overapproximates proc. The second follows from the induction hypothesis and the monotonicity of proc_f. The inequalities in (4) and (5) imply the desired $F(\gamma(\mathsf{widenfix}\ G)) \sqsubseteq \gamma(\mathsf{widenfix}\ G)$. □

3.4 Generic Analysis

Let $\eta^\sharp_*$ be the unique abstract environment for the empty context $\Gamma = \emptyset$. Our generic analysis takes a command c with no free procedures, and computes the function: LivAnalysis(c) = satisfyLiv$^\sharp([\![c]\!]^\sharp\eta^\sharp_*)$. The result is a boolean value, indicating whether c satisfies a liveness property specified by LivProperty.

Theorem 3. *Let* η_* *be the unique concrete environment for the empty context* $\Gamma = \emptyset$. *Then, for all commands* c *with no free procedures, if* LivAnalysis$(c) =$ true, *we have that* $[\![c]\!]\eta_* \in$ LivProperty.

4 Instance of the Framework

In this section, we instantiate the framework and define a sound abstract interpreter for proving the termination of programs with general recursion.

4.1 Concrete Semantics

Our instance of concrete semantics of the framework interprets commands as sets of traces satisfying certain healthiness conditions. The notion of traces here is slightly unusual, because the traces are sequences of *tagged states* and they need to meet our well-formedness conditions. In this section, we will explain the meanings of tagged states and traces, and provide parameters necessary for instantiating a concrete semantics from the framework.

Tagged States, Pre-traces and Traces We start with the definition of traces. A **state** is a map from program variables to rational numbers, and a **tagged state** is a pair of state and tag:

$$\mathsf{Tag} = \{\mathit{none}\} \cup \mathsf{PName} \times \{\mathit{call}, \mathit{ret}\}, \quad \mathsf{State} = \mathsf{Var} \to \mathbb{Q}, \quad \mathsf{tState} = \mathsf{State} \times \mathsf{Tag}.$$

The tag of a tagged state indicates whether the state is the initial or the final state of a procedure call, or just a normal one not related to a call. The (f, call) and (f, ret) tags mean that the state is, respectively, the initial and the final state of the call $f()$, and the *none* tag indicates that the state is a normal state, i.e., it is neither the initial nor the final state of a procedure call. We use symbol σ to denote elements in tState, and use s to denote elements in State.

A **pre-trace** τ is a nonempty finite or infinite sequence of tagged states, such that τ starts with a *none*-tagged state and if it is finite, it ends with a *none*-tagged state.

$$\mathsf{nState} = \mathsf{State} \times \{\mathit{none}\}, \quad \mathsf{preTrace} = \mathsf{nState}(\mathsf{tState}^*)\mathsf{nState} \cup \mathsf{nState}(\mathsf{tState}^\infty),$$

where tState^∞ means the set of (countably) infinite sequences of tagged states.

A trace τ is a pre-trace that satisfies well-formedness conditions. To define these conditions, we consider the sets $\mathcal{W}, \mathcal{O}$ of sequences of tagged states that are the least fixpoints of the below equations:

$$\mathcal{W} = \mathsf{nState}^* \cup \mathcal{W}\mathcal{W} \cup \big(\bigcup_{f \in \mathsf{PName}, s, s_1 \in \mathsf{State}} \{(s, (f, \mathit{call}))\}\, \mathcal{W}\, \{(s_1, (f, \mathit{ret}))\}\big),$$
$$\mathcal{O} = \mathcal{W} \cup \mathcal{O}\mathcal{O} \cup \big(\bigcup_{f \in \mathsf{PName}, s \in \mathsf{State}} \{(s, (f, \mathit{call}))\}\mathcal{O}\big).$$

Intuitively, $\mathcal{W}$ describes sequences where every procedure call has a matching return and calls and returns are well-bracketed. The other set $\mathcal{O}$ defines a bigger set; in each trace in $\mathcal{O}$, some procedure calls might not have matching returns, but calls and returns should be well-bracketed.

Definition 1. *A pre-trace τ is a* **trace** *iff τ is finite and belongs to $\mathcal{W}$, or τ is infinite and all of its finite prefixes are in $\mathcal{O}$. We write* Trace *for the set of traces.*

For $\tau \in \mathsf{Trace}$ and $n \in \mathbb{N} \cup \{\infty\}$, the projection $\tau[n]$ is the n-prefix of τ; in case that $|\tau| < n$, $\tau[n] = \tau$.[5] Using this projection, we define the distance function on traces as follows:

$$d(\tau, \tau') = 2^{-\max\{n \mid \tau[n] = \tau'[n]\}} \qquad \text{(where we regard } 2^{-\infty} = 0\text{)}.$$

[5] $\tau[n]$ does not necessarily belong to Trace or even to $\mathsf{preTrace}$, but this will not cause problems for our results.

$$
\begin{aligned}
\mathsf{seq}(T, T') &= \{\tau\sigma\tau' \mid (\tau\sigma \in T \cap \mathsf{tState}^+) \wedge (\sigma\tau' \in T')\} \cup (T \cap \mathsf{tState}^\infty) \\
\mathsf{asgn}_{x,e} &= \{\sigma\sigma' \mid \sigma, \sigma' \in \mathsf{nState} \wedge \mathsf{first}(\sigma') = \mathsf{first}(\sigma)[x \mapsto [\![e]\!]\mathsf{first}(\sigma)]\} \\
\mathsf{if}_b(T_0, T_1) &= \{\sigma\tau \mid (\sigma\tau \in T_0 \wedge [\![b]\!](\mathsf{first}(\sigma)) = \mathsf{true}) \vee (\sigma\tau \in T_1 \wedge [\![b]\!](\mathsf{first}(\sigma)) = \mathsf{false})\} \\
\mathsf{proc}_f(T) &= \{\sigma\sigma^{(f,call)}\tau \mid \sigma\tau \in (T \cap \mathsf{tState}^\infty)\} \cup \{\sigma\sigma^{(f,call)}\sigma^{(f,ret)}\sigma \mid \sigma \in T\} \\
&\quad \cup \{\sigma\sigma^{(f,call)}\tau\sigma_1^{(f,ret)}\sigma_1 \mid \sigma\tau\sigma_1 \in (T \cap \mathsf{tState}^+)\}
\end{aligned}
$$

Here $\mathsf{first}(\sigma)$ is the first component of the tagged state σ, and $\sigma^{(f,call)}$ and $\sigma^{(f,ret)}$ are, respectively, $(\mathsf{first}(\sigma), (f, \mathit{call}))$ and $(\mathsf{first}(\sigma), (f, \mathit{ret}))$. And $[\![b]\!]$ and $[\![e]\!]$ are the standard interpretation of booleans and expressions as functions from (untagged) states to $\{\mathsf{true}, \mathsf{false}\}$ and $\mathbb{Q}$.

Fig. 4. Semantic Operators for the Instance Concrete Semantics

Lemma 4. (Trace, d) *is a complete metric space.*

Full Closed Sets of Well-formed Traces A subset $T_0 \subseteq \mathsf{Trace}$ of traces is **closed** if for all Cauchy sequences of traces in T_0, their limits belong to T_0 as well. A trace set $T_0 \subseteq \mathsf{Trace}$ is **full** if for every *none*-tagged state $\sigma \in \mathsf{nState}$, there is a trace $\tau \in T_0$ starting with σ.

The semantic domain $(\mathcal{D}, d)$ for interpreting commands in our concrete semantics is the set $\mathcal{P}_{fcl}(\mathsf{Trace})$ of *full closed* sets of traces:

$$\mathcal{D} \;=\; \mathcal{P}_{fcl}(\mathsf{Trace}), \qquad d^\dagger(T, T') \;=\; 2^{-\max\{n \mid T[n]=T'[n]\}}.$$

Here $T[n]$ is the result of taking the prefix of every trace in T (i.e., $T = \{\tau[n] \mid \tau \in T\}$). The closedness ensures that the $d^\dagger$ just defined satisfies the axioms for being a complete metric space. Also, the condition about being full allows us to meet the non-expansiveness requirement for seq in our framework.

Our domain $\mathcal{D}$ is ordered by the subset relation $\subseteq$. With respect to this $\subseteq$ order, $\mathcal{D}$ has the top element, which is the set Trace of all traces.

Lemma 5. $(\mathcal{D}, d^\dagger)$ *is a complete metric space. Furthermore, the requirement* (1) *of our framework in Sec. 3 holds for* $\subseteq$ *and this metric space.*

Semantic Operators So far we have defined the semantic domain for commands, the first required parameter of the framework. The next four parameters are operators working on this domain, and we describe them in Fig. 4. In the figure, the sequencing operator seq concatenates traces from T and T', while treating infinite traces from T specially. And the operator proc_f duplicates initial and final states, and tags the duplicated states with information about procedure call and return.

Lemma 6. *All the operators are well-defined, and satisfy the monotonicity and non-expansiveness or* $\frac{1}{2}$*-contractiveness requirements of our framework.*

$$E \;::=\; r \mid x \mid {\lq}x \mid x' \mid E + E \mid r \times E \qquad P \;::=\; E = E \mid E \neq E \mid E < E \mid E \leq E$$
$$\varphi \;::=\; P \mid \mathsf{true} \mid \varphi \wedge \varphi \mid \mathsf{false} \mid \varphi \vee \varphi \mid \exists x'.\, \varphi$$

Fig. 5. Syntax for Linear Constraints

Liveness Property The only remaining parameter is LivProperty, which describes a desired liveness property on trace sets. Here we use a property such that if we restrict our attention to $T = [\![c]\!]\eta$ of some command c with no free procedure names, the membership of T to this property implies that T consists of finite traces only.

We say that a trace τ includes an infinite subsequence of open calls iff there exists $\{\tau_n \sigma_n\}_{n \in \mathbb{N}}$ such that

1. $\tau \;=\; \tau_1 \sigma_1 \tau_2 \sigma_2 \tau_3 \sigma_3 \ldots \tau_n \sigma_n \ldots$,
2. for all $i \in \mathbb{N}$, there exists some $f \in \mathsf{PName}$ such that $\mathsf{second}(\sigma_i) = (f, \mathit{call})$,
3. for all $i \in \mathbb{N}$, the corresponding return for σ_i does not appear in τ after σ_i, i.e., the return does not occur in the sequence $\tau_{i+1} \sigma_{i+1} \tau_{i+2} \sigma_{i+2} \ldots$.

We specify a desired liveness property of (semantic) commands, using the following subset LivProperty of $\mathcal{D}$: T is in LivProperty iff no traces in T include an infinite subsequence of open calls.

4.2 Abstract Semantics with Linear Ranking Relations

Our abstract semantics uses formulas φ for linear constraints. The syntax of these formulas is given in Fig. 5. Note that a formula φ can use three kinds of variables: normal program variables x; pre-primed ones ${\lq}x$ for denoting the value of x before running a program; primed ones x' that can be existentially quantified. We assume that the set Var of normal variables and the set ${\lq}\mathsf{Var}$ of pre-primed variables are finite and that there is an one-to-one correspondence between Var and ${\lq}\mathsf{Var}$, which maps x to ${\lq}x$.

Let Form be the set of formulas φ that do not contain free primed variables. Each $\varphi \in \mathsf{Form}$ defines a relation from (untagged) states ${\lq}s$ with pre-primed variables (i.e., ${\lq}s \in {\lq}\mathsf{Var} \to \mathbb{Q}$) to (untagged) states s with normal variables:

$$({\lq}s, s) \;\models\; \varphi,$$

where $\models$ is the standard satisfaction relation from the first-order logic. Let TForm be a subset of Form consisting of *total* formulas in the sense below:

$$\mathsf{TForm} \;=\; \{\varphi \in \mathsf{Form} \mid \forall {\lq}s \in ({\lq}\mathsf{Var} \to \mathbb{Q}).\, \exists s \in (\mathsf{Var} \to \mathbb{Q}).\, ({\lq}s, s) \models \varphi\}.$$

The abstract semantics in this section assumes a sound but possibly incomplete theorem prover that can answer queries of the two kinds: $\varphi \vdash \psi$ and $\vdash \forall {\lq}X.\, \exists X.\, \varphi$. Here ${\lq}X$ and X are the sets of free pre-primed variables and

normal variables in φ. Note that by asking the query of the second kind, we can use a prover to check, soundly, whether a formula φ belongs to TForm.

Using what we have defined or assumed so far, we define an abstract domain $\mathcal{A}$ and its subset $\mathcal{A}_t$ of total abstract elements as follows:

$$\mathcal{A} = \mathsf{Form} \times \mathsf{Form} \times \mathsf{Form}, \quad \mathcal{A}_t = \mathsf{TForm} \times \mathsf{Form} \times \mathsf{Form}, \quad \mathcal{A}_p = \mathcal{A} - \mathcal{A}_t.$$

The element $(\mathsf{false}, \mathsf{false}, \mathsf{false})$ in $\mathcal{A}$ serves the role of $\bot$, and $(\mathsf{true}, \mathsf{true}, \mathsf{true})$ the role of $\top$. The algorithm for soundly checking the totality of abstract elements is defined using the assumed prover:

$$\mathsf{checktot}(A) \;=\; \textbf{if}\;\; (\vdash \forall\, `X.\exists X.\, A_1)\;\; \textbf{then}\;\; \mathsf{true}\;\; \textbf{else}\;\; \mathsf{unknown}$$

where A_i is the i-th component of A and $`X$ and X are the sets of free pre-primed and free normal variables in A_1.

Next, we define the concretization map γ, which will provide the intuitive meaning of abstract elements in $\mathcal{A}$. To do this, we need to introduce some additional notations. Firstly, for a (untagged) state s, we write $`s$ for the state obtained from s by renaming normal variables by corresponding pre-primed ones. Secondly, we write $\sigma \in \tau$ to mean that σ is a tagged state appearing in τ, and $\mathsf{iscall}(\sigma)$ to mean that the tag for σ is a procedure call:

$$\mathsf{iscall}(\sigma) \iff \exists f \in \mathsf{PName}.\, (\mathsf{second}(\sigma) = (f, \mathit{call})).$$

Finally, for all tagged states $\sigma_1, \sigma_2 \in \tau$, we say that σ_1 is an open call with respect to σ_2 in τ, denoted $\mathsf{open}(\sigma_1, \sigma_2, \tau)$, if both σ_1 and σ_2 are tagged with procedure calls, σ_1 appears strictly before σ_2 in τ, but the corresponding return for σ_1 does not appear before σ_2. The concretization is defined as follows:

$$\begin{aligned}\gamma(A) = \{\tau \in \mathsf{Trace} \mid\; & (\tau \in \mathsf{tState}^{+} \implies (`\mathsf{first}(\mathsf{first}(\tau)), \mathsf{first}(\mathsf{last}(\tau))) \models A_1) \;\wedge \\ & (\forall \sigma \in \tau.\, \mathsf{iscall}(\sigma) \implies (`\mathsf{first}(\mathsf{first}(\tau)), \mathsf{first}(\sigma)) \models A_2) \;\wedge \\ & (\forall \sigma_1, \sigma_2.\, \mathsf{open}(\sigma_1, \sigma_2, \tau) \implies (`\mathsf{first}(\sigma_1), \mathsf{first}(\sigma_2)) \models A_3)\}.\end{aligned}$$

Here A_i is the i-th component of A. According to this concretization, A_1 relates the initial and final states of a trace τ, and A_2 and A_3 describe the relationship between certain intermediate states in τ; A_2 relates the initial state and a call state in τ, and A_3 relates states at two open calls in τ. Tracking the relationship between intermediate states is crucial for the precision of our analysis. If the abstract domain included only the first component (as in our previous work [4]), the concretizations of its elements would contain traces violating LivProperty, or they would not belong to $\mathcal{D}$.

Lemma 7. *For every $A \in \mathcal{A}_t$, the set $\gamma(A)$ is in $\mathcal{D}$, i.e., it is full and closed.*

Abstract Operators For $\varphi, \psi \in \mathsf{Form}$, let $\varphi; \psi$ be their relational composition defined by

$$\varphi; \psi \;\equiv\; \exists Y'.(\varphi[Y'/X] \wedge \psi[Y'/`X]).$$

$$\mathsf{seq}^\sharp(A, A') = (\ A_1; A'_1,\ \ A_2 \vee (A_1; A'_2),\ \ A_3 \vee A'_3\ )$$
$$\mathsf{asgn}^\sharp_{x,e} = (\ eq_{\mathsf{Var}-\{x\}} \wedge (e[`x/x] = x),\ \mathsf{false},\ \mathsf{false}\ )$$
$$\mathsf{if}^\sharp_b(A, A') = \textbf{let}\ b_1 = \mathsf{preprime}(b)\ \text{ and }\ b_2 = \mathsf{preprime}(\mathsf{neg}(b))$$
$$\textbf{in}\ (\ (b_1 \wedge A_1) \vee (b_2 \wedge A'_1),\ \ (b_1 \wedge A_2) \vee (b_2 \wedge A'_2),\ \ A_3 \vee A'_3\ )$$
$$\mathsf{proc}^\sharp_f(A) = (\ A_1,\ eq_{\mathsf{Var}} \vee A_2,\ A_2 \vee A_3\ )$$

Here $\mathsf{preprime}(b)$ renames all the variables with the corresponding pre-primed variables, and $\mathsf{neg}(b)$ is the negation of b where $\neg$ is removed by being pushed all the way down to atomic predicates using logical equivalences. For instance, $\mathsf{neg}(x{=}y \vee z{<}3)$ is $x{\neq}y \wedge 3{\leq}z$.

Fig. 6. Semantic Operators for the Instance Abstract Semantics

Here X and $`X$ respectively contain normal variables in φ and pre-primed variables in ψ, Y' is the set of fresh primed variables, and the cardinalities of these three sets are the same so that the substitution in $\varphi; \psi$ is well-defined. Also, for a set X of normal variables, define the formula eq_X to be the equality on the variables in X and the corresponding pre-primed ones: $eq_X \equiv \bigwedge_{x \in X}(`x = x)$.

Using these notations, we present abstract operators in Fig. 6. Note that the abstract sequencing $\mathsf{seq}^\sharp(A, A')$ is not simply the relational composition of formulas; it also describes relationships between intermediate states of a trace. For instance, the second component $A_2 \vee (A_1; A'_2)$ relates the initial state of a trace with states at procedure call in the trace. The first disjunct A_2 considers the case that a call state is from the first argument A of the sequencing, and the second $A_1; A'_2$ is for the other case that a call is from the second argument A'.

Lemma 8. *The operators in Fig. 6 meet all the requirements of our framework.*

Widening Operator Our widening operator is parameterized by three elements. The first is a positive integer k, which bounds the number of outermost disjuncts in formulas appearing in the results of widening. We will write $\triangledown_k$ to make this parameterization explicit. The second is a function lower that overapproximates a formula φ in Form by the conjunction of lower bounds on some pre-primed variables (i.e., the conjunction of formulas of the form $r \leq `x$ for some *pre-primed* variable $`x$ and rational number r):

$$\mathsf{lower}(\varphi) \quad = \quad (r_1 \leq `x_1\ \wedge\ r_2 \leq `x_2\ \wedge\ \ldots r_n \leq `x_n)$$

such that φ entails $\mathsf{lower}(\varphi)$ semantically. The third is the dual of the second function. It is a function upper that overapproximates a formula φ in Form by the conjunction of formulas of the form $`x \leq r$.

The widening operator uses three subroutines. The first is toDNF that transforms a formula $\varphi \in \mathsf{Form}$ to a disjunctive normal form, where all existential quantifications are placed right before each conjunct. The second is the function $\mathsf{bound}_k \colon \mathsf{Form} \to \mathsf{Form}$ for bounding the number of outermost disjuncts to k:

$$\mathsf{bound}_k(\varphi)\ =\ \textbf{if}\ \ (\text{at most } k \text{ outermost disjuncts are in } \varphi)\ \ \textbf{then}\ \ \varphi\ \ \textbf{else}\ \ \mathsf{true}$$

The third is an algorithm RFS that synthesizes a linear ranking function from φ. such as RankFinder in [18]. Semantically, unless RFS returns fail, it computes an overapproximation of a disjunction-free formula $\varphi \in \mathsf{Form}$, and the overapproximation expresses a linear ranking relation, such as $10 < `x \wedge x \leq `x - 1$ for the ranking function x.

Using these parameters and subroutines, we can now define the widening operator:

$$\begin{aligned} A \triangledown_k A' = \textbf{let } (\textstyle\bigvee_{j\in J_i} \kappa^i_j) &= \mathsf{toDNF}(A'_i) \qquad (i = 1,2,3 \text{ here and below}) \\ \chi^i_j &= \textstyle\bigwedge \{`x = x \mid x \in \mathsf{Var} \text{ and } \kappa^i_j \vdash `x = x\} \\ \xi^i_j &= \textbf{if } \big(\ \mathsf{RFS}(\kappa^i_j) = \zeta^i_j \text{ for some formula } \zeta^i_j\ \big) \\ &\quad\ \textbf{then } \big(\ \zeta^i_j \wedge \mathsf{lower}(\kappa^i_j) \wedge \mathsf{upper}(\kappa^i_j) \wedge \chi^i_j\ \big) \\ &\quad\ \textbf{else } \big(\ \mathsf{lower}(\kappa^i_j) \wedge \mathsf{upper}(\kappa^i_j) \wedge \chi^i_j\ \big) \\ \delta_i &= \mathsf{bound}_k(A_i \vee \textstyle\bigvee_{j\in J_i} \{\xi^i_j \mid \kappa^i_j \not\vdash A_i\}) \\ \textbf{in } (\delta_1, \delta_2, \delta_3). \end{aligned}$$

Lemma 9. *The operator $\triangledown_k : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ is a widening operator.*

Abstract Liveness Predicate The abstract semantics uses the following predicate satisfyLiv$^\sharp$ on $\mathcal{A}_t$ and checks whether an analysis result implies the desired liveness property:

$$\begin{aligned} \textsc{satisfyLiv}^\sharp(A) \ = \ & \textbf{let } (\textstyle\bigvee_{i\in I} \delta_i) = \mathsf{toDNF}(A_3) \\ & \textbf{in } \big(\textbf{if } (\mathsf{RFS}(\delta_i) \neq \mathsf{fail} \text{ for all } i \in I) \textbf{ then } \mathsf{true} \textbf{ else } \mathsf{false}\big). \end{aligned}$$

The predicate satisfyLiv$^\sharp$ first transforms A_3 to a disjunctive normal form. Then, it checks whether each disjunct δ_i is well-founded using the function RFS. Hence, if the predicate returns true, it means that A_3 is disjunctively well-founded. The below lemma is an easy consequence of the disjunctively well-foundedness of A_3, the result of Podelski and Rybalchenko [19] and the definition of γ.

Lemma 10. *For all $A \in \mathcal{A}_t$, if* satisfyLiv$^\sharp(A) = \mathsf{true}$*, we have that $\gamma(A) \in$* LivProperty*.*

5 Conclusion

In this paper, we have presented a framework for designing a sound abstract interpreter for liveness properties. The framework incorporates the theory of metric spaces in the concrete semantics. By doing so, it justifies a new strategy for approximating fixpoints for an abstract interpreter for liveness, and relieves the burden of abstracting low-level details from an analysis designer. We hope that our results help the program analysis community to exploit metric space semantics and other unexplored areas of the semantics research for developing effective program analysis algorithms.

Acknowledgments We would like to thank Martin Escardo, Xavier Rival and Andrey Rybalchenko for helpful suggestions, comments and discussion. Peter O'Hearn has always been supportive and provided helpful suggestions. Chawdhary was supported by a Microsoft Research PhD Scholarship and ESPRC grant EP/G042322/1, and Yang was supported by EPSRC grant EP/E053041/1.

References

1. I. Balaban, A. Pnueli, and L. Zuck. Ranking abstraction as companion to predicate abstraction. In *FORTE'05*, 2005.
2. A. Bradley, Z. Manna, and H. Sipma. Termination of polynomial programs. In *VMCAI'05*, 2005.
3. A. R. Bradley, Z. Manna, and H. B. Sipma. The polyranking principle. In *ICALP'05*, 2005.
4. A. Chawdhary, B. Cook, S. Gulwani, M. Sagiv, and H. Yang. Ranking abstractions. In *ESOP'08*, 2008.
5. M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *JLP*, 41(1):103–123, 1999.
6. B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Y. Vardi. Proving that programs eventually do something good. In *POPL'07*, 2007.
7. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI'06*, 2006.
8. B. Cook, A. Podelski, and A. Rybalchenko. Summarization for termination: no return! *Formal Methods in System Design*, 35(3):369–387, 2009.
9. P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theor. Comput. Sci.*, 277(1-2):47–103, 2002.
10. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.
11. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.
12. P. Cousot and R. Cousot. Bi-inductive structural semantics. *Information and Computation*, 207(2):258–283, 2009.
13. J. de Bakker and E. de Vink. *Control flow semantics*. MIT Press, Cambridge, MA, USA, 1996.
14. M. Escardó. A metric model of PCF. *unpublished research note*, 1998.
15. B. Jeannet and W. Serwe. Abstracting call-stacks for interprocedural verification of imperative programs. In *Int. Conf. on Algebraic Methodology and Software Technology, AMAST'04*, volume 3116 of *LNCS*, July 2004.
16. D. Kroening, N. Sharygina, A. Tsitovich, and C. Wintersteiger. Termination analysis with compositional transition invariants. In *CAV'10*, 2010. To appear.
17. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. *SIGPLAN Not.*, 36(3):81–92, 2001.
18. A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI'04*, 2004.
19. A. Podelski and A. Rybalchenko. Transition invariants. In *LICS'04*, 2004.
20. F. van Breugel. An introduction to metric semantics: operational and denotational models for programming and specification languages. *Theoretical Computer Science*, 258(1-2):1 – 98, 2001.