Deciding subset relationship of co-inductively defined set constants

Manfred Schmidt-Schauß[1], David Sabel[1], and Marko Schütz[2]

[1] Institut für Informatik, Johann Wolfgang Goethe-Universität,
Postfach 11 19 32, D-60054 Frankfurt, Germany,
schauss@ki.informatik.uni-frankfurt.de
[2] Dept. of Mathematics and Computing Science,
University of the South Pacific, Suva, Fiji Islands

Technical Report Frank-23
Research group for Artificial Intelligence and Software Technology,
Institut für Informatik,
J.W.Goethe-Universität Frankfurt,

15. Aug. 2005

Abstract. Static analysis of different non-strict functional programming languages makes use of set constants like `Top`, `Inf`, and `Bot` denoting all expressions, all lists without a last Nil as tail, and all non-terminating programs, respectively. We use a set language that permits union, constructors and recursive definition of set constants with a greatest fixpoint semantics. This paper proves decidability, in particular EXPTIME-completeness, of subset relationship of co-inductively defined sets by using algorithms and results from tree automata. This shows decidability of the test for set inclusion, which is required by certain strictness analysis algorithms in lazy functional programming languages.

1 Introduction

Static analysis in lazy functional languages often has to deal with abstract sets of expressions, such as $\top$ for all expressions, $\bot$ for all non-terminating expressions, and `Inf` for all infinite lists (lists without a length). Initially, strictness analysis on non-flat domains was explored using 4 predefined set constants [Wad87]. Subsequently, this has been generalized to strictness analyses providing a language for defining abstract set constants [Nöc92,Nöc93,Sch00,SSPS95], where the analysis requires an inclusion check of set constants in the most powerful version of its loop detection rules. Such a language is also used in the proof of correctness of Nöcker's strictness analysis [SSSS05]. In every case the static analysis requires knowledge of the inclusion of two differently defined sets. In this paper, we show that in a language that allows constructor expressions, unions and recursion,

the decision problem for the inclusion of set constants is EXPTIME-complete by using results from set constraints [Aik94,CP98] and tree automata [CDG+97].

The paper is structured as follows. First the language for forming set constants is given. Set constants can be recursively defined by defining them as a union of $\{\bot\}$ and flat constructor terms. But set constants are not permitted in the union. In section 3 it is shown that the inclusion problem for set constants can be solved using results from tree automata by showing that inclusion w.r.t. the least fixpoint is equivalent to inclusion w.r.t. greatest fixpoint. In section 4 it is shown that the set constant language can be extended to allow set constants at the top-level of unions without changing the complexity of the inclusion problem. In section 5 we show how the results are transferred into extended lambda-calculi, and hence to lazy functional programming languages. It is essential to note that infinite trees are provided by these languages, but not all infinite trees, which means we cannot directly use the least fixpoint of equations, nor arguments using the set of all infinite trees.

2 Syntax of the Language of Values

We use some finite set $\mathcal{A}$ of constants a coming with an arity $\mathrm{ar}(a) \geq 0$. There is one special constant $\bot$ with $\mathrm{ar}(\bot) = 0$. The other constants are called *constructors*. The syntax for expressions E is:

$$E ::= (c\ E_1 \dots E_{\mathrm{ar}(c)}) \text{ where } c \text{ is a constant}$$

We define $\mathcal{L}(\mathcal{A})$ as the set of all expressions E that can be generated using this grammar. We also require the set of all finite and infinite trees $\mathcal{T}(\mathcal{A})$ according to this syntax.
In order to have a rigorous construction, a (possibly) infinite tree $t \in \mathcal{T}(\mathcal{A})$ over $\mathcal{A}$ is a partial function $t : \mathcal{FS}(\mathbb{N}_0) \to \mathcal{A}$, where $\mathcal{FS}(\mathbb{N}_0)$ is the set of all finite sequences over $\mathbb{N}_0$. The domain of the function t is denoted as $D(t)$. Sequences are denoted using the dot-notation. The notation $f.n$ means the sequence f extended with the number n, and $|f|$ is the length of the sequence f. The following properties hold:

1. $f.n \in D(t) \Rightarrow f \in D(t)$.
2. If $\mathrm{ar}(t(f)) = n$, then $f.i \in D(t)$ for $i = 1, \dots, n$.
3. If $\mathrm{ar}(t(f)) = n$ and $f.i \in D(t)$, then $1 \leq i \leq n$.

This implies that if $\mathrm{ar}(t(f)) = 0$, then f is a maximal sequence in $D(t)$, i.e. f corresponds to a leaf in the tree t. The set $D(t)$ is always prefix-closed, i.e. every prefix of a sequence from $D(t)$ is also contained in $D(t)$. If the set $D(t)$ is finite, then we call t a *finite tree*, or a *term over* $\mathcal{A}$, otherwise, if $D(t)$ is an infinite set, we call t an *infinite tree*. If appropriate, we will also use the common notation for trees as terms over a signature.

In the following, instead of the full set $\mathcal{T}(\mathcal{A})$ and in order to support applications, where not all infinite trees are available, we use a set of trees $\mathcal{T}$ with $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{T} \subseteq \mathcal{T}(\mathcal{A})$, which is also *selector-closed*, i.e., with the property that for every $t = (c\ t_1 \dots t_n) \in \mathcal{T}$, also $t_i \in \mathcal{T}$ for all $i = 1, \dots, n$.

In an application of the results, this set $\mathcal{T}$ may be the set of all computable trees in $\mathcal{T}(\mathcal{A})$, or the set of all equivalence classes (by some equivalence) of terminating expressions.

3 Set Constants

Let $\mathcal{U} = \{U_1, \dots, U_K\}$ be a finite set of names of set constants. For every set constant U_i there is a defining rule

$$(Eq_i): \quad U_i = \{\bot\} \cup r_{i,1} \cup \dots \cup r_{i,n_i}$$

where $r_{i,j}$ may be an expression $(c\ u'_1 \dots u'_{\mathrm{ar}(c)})$, and $u'_j \in \mathcal{U}$ are set constants. With $rhs_{Eq}(U_i)$ we denote the right-hand side of Eq_i.

A mapping $\psi : \mathcal{A} \to \mathcal{P}(\mathcal{T})$, where $\mathcal{P}(\cdot)$ denotes the powerset, is called an *sc-interpretation*. For sc-interpretations ψ_1, ψ_2, we write $\psi_1 \leq \psi_2$, iff for all $i = 1, \dots, K : \psi_1(U_i) \subseteq \psi_2(U_i)$.

We define an extension ψ^e for sc-interpretations ψ as follows:

$$\begin{array}{ll} \psi^e(\{\bot\}) & := \{\bot\} \\ \psi^e(c\ u_1\ \dots\ u_{\mathrm{ar}(c)}) & := \{(c\ a_1 \dots a_{\mathrm{ar}(c)}) \mid a_i \in \psi(u_i)\} \\ \psi^e(\{\bot\} \cup r_1 \cup \dots \cup r_n) & := \{\bot\} \cup \psi^e(r_1) \cup \dots \cup \psi^e(r_n) \end{array}$$

The equations Eq_i for the set constants define an operator Ψ on sc-interpretations as follows:

$$\Psi(\psi) := \psi^e \circ rhs_{Eq}.$$

The operator Ψ is monotone on the set of sc-interpretations and hence has a least fixpoint σ and a greatest fixpoint γ, where σ, γ are sc-interpretations.

Definition 1. *For every set constant $u \in \mathcal{U}$ we define the least fixpoint $\sigma(u)$ and the greatest fixpoint as $\gamma(u)$.*

Remark 1. The least fixpoint σ of Ψ can be computed as follows:

Let ϕ_0 be the sc-interpretation with $\phi_0(U_i) = \emptyset$ for $i = 1, \dots, K$. With $\phi_j := \Psi^j(\phi_0)$ for $j > 0$, the j-fold application of Ψ, the equation $\sigma(U_i) = \bigcup_j \phi_j(U_i)$ holds for every $i = 1, \dots, K$. This representation of the least fixpoint allows induction proofs.

For the least fixpoint σ of equations we have $\sigma(u) \subseteq \mathcal{L}(\mathcal{A})$ for all $u \in \mathcal{U}$. So in this case only finite trees are required.

Remark 2. The greatest fixpoint γ of Ψ can be computed as follows. Let ψ_0 be the sc-interpretation with $\psi_0(U_i) = \mathcal{T}$ for $i = 1, \ldots, K$. With $\psi_j := \Psi^j(\psi_0)$ for $j > 0$, the j-fold application of Ψ, the equation $\gamma(U_i) = \bigcap_j \psi_j(U_i)$ holds for every $i = 1, \ldots, K$. This representation of the fixpoint allows co-induction proofs in the style of induction proofs.

The greatest fixpoint γ of equations in general requires infinite trees.

3.1 Properties of Least and Greatest Fixpoints

In this section we show that there is a tight connection between least and greatest fixpoint: The set $\sigma(u)$ is exactly the subset of all finite cuts of trees in $\gamma(u)$.

Definition 2. *A tree t' is a* finite cut *of a tree t, if t' is finite, $D(t') \subseteq D(t)$, and $\forall f : t'(f) \neq t(f) \Rightarrow t'(f) = \bot$.*
A tree t' is a finite cut *of a tree t* at depth *k, if t' is a finite cut of t, and $\forall f : t'(f) \neq t(f) \Rightarrow |f| = k$.*

A finite cut can be seen as cutting away subtrees by replacing them with a $\bot$-leaf until the resulting tree is finite.

Lemma 1. *For all set constants $u \in \mathcal{U}$:*

1. $\sigma(u) \subseteq \gamma(u)$.
2. $\sigma(u) = \{t' \mid t' \text{ is a finite cut of a tree in } \gamma(u)\}$.

Proof. 1. For every $u \in \mathcal{U}$, the set $\sigma(u)$ can be described as all finite trees that can be constructed using the equations Eq_i (see Remark 1). Hence for every sc-interpretation ψ that is a fixpoint of Ψ, and for all $u \in \mathcal{U}$ we have $\sigma(u) \subseteq \psi(u)$, and hence $\sigma(u) \subseteq \gamma(u)$.

2. By induction. Suppose there is a tree $t \in \gamma(u)$, and a finite cut t' of t, such that $t' \notin \sigma(u)$. Assume that the depth of t' is minimal with this property. Since $t' \neq \bot$, there is a constructor c with $t = (c\ t_1 \ldots t_n)$, $t' = (c\ t'_1 \ldots t'_n)$, and t'_i is a finite cut of t_i for all $i = 1, \ldots, n$. The fixpoint equations show that there is a component $(c\ u_1 \ldots u_n)$ in the right-hand side of the equation for u with $(c\ t_1 \ldots t_n) \in \gamma(c\ u_1 \ldots u_n)$, and so $t_i \in \gamma(u_i)$. Thus, by induction $t'_i \in \sigma(u_i)$ for all $i = 1, \ldots, n$. The fixpoint equations again show that this implies that $(c\ t'_1 \ldots t'_n) \in \sigma(u)$. This is a contradiction. Hence the claim is proved.

Corollary 1. *For all $u \in \mathcal{U}$: All finite cuts of trees in $\sigma(u)$ are also in $\sigma(u)$.*

Proof. Follows from Lemma 1.

Lemma 2. *Let u be a set constant. Then $\gamma(u) = \{t \in \mathcal{T} \mid$ all finite cuts of t are in $\sigma(u)\}$*

Proof. From Lemma 1 we obtain $\gamma(u) \subseteq \{t \in \mathcal{T} \mid$ all finite cuts of t are in $\sigma(u)\}$.

Let ξ be the sc-interpretation that is defined by $\xi(u) := \{t \in \mathcal{T} \mid$ all finite cuts of t are in $\sigma(u)\}$ for all $u \in \mathcal{U}$. We show that ξ is a fixpoint of Ψ:
First, it is obvious that $\sigma(u) \subseteq \xi(u)$ for all $u \in \mathcal{U}$.
Let $u \in \mathcal{U}$ and let the equation for u be: $u = \{\bot\} \cup r_1 \cup \ldots \cup r_m$. Then $\Psi(\xi)(u) = \{\bot\} \cup \xi^e(r_1) \cup \ldots \cup \xi^e(r_m)$. The goal is to show that $\Psi(\xi)(u) = \xi(u)$:
$\Psi(\xi)(u) \subseteq \xi(u)$: Let $t \in \Psi(\xi)(u)$. The case $t = \bot$ is trivial, so assume $t = (c\ t_1 \ldots t_h)$, and w.l.o.g. $t \in \xi^e(r_1)$ with $r_1 = (c\ u_1\ \ldots u_h)$. We obtain $t_i \in \xi(u_i)$ for all $i = 1, \ldots, h$. Hence all finite cuts of t_i are in $\sigma(u_i)$ for $i = 1, \ldots, h$, and since $\sigma(u) = \{\bot\} \cup \sigma(r_1) \cup \ldots \cup \sigma(r_m)$, we have that all finite cuts of t are in $\sigma(u)$. This means $t \in \xi(u)$.

$\xi(u) \subseteq \Psi(\xi)(u)$: Let $t \in \xi(u)$. If $t = \bot$, then there is nothing to prove. If $t = (c\ t_1 \ldots t_n)$, then consider a sequence of finite cuts as follows: For all $i = 0, 1, 2, \ldots$ let s_i be the finite cut of $(c\ t_1 \ldots t_n)$ at depth i. Since $\sigma(v) \subseteq \xi(v)$ for all $v \in \mathcal{U}$, we have $s_i \in \sigma(u) \subseteq \xi(u)$ for all i. Then there is a component $(c\ u_1 \ldots u_n)$ among the $r_1, \ldots, r_m$, such that for infinitely many indices i, the tree s_i is also in $\sigma(c\ u_1 \ldots u_n)$. This, however, means that all the finite cuts of s_i are also $\sigma(c\ u_1 \ldots u_n)$ by Corollary 1, and so all the finite cuts of t are in $\sigma(c\ u_1 \ldots u_n)$. Hence all the finite cuts of t_i are in $\sigma(u_i)$ for all $i = 1, \ldots, n$. Since $\mathcal{T}$ is selector-closed, $t_i \in \mathcal{T}$. This implies $t_i \in \xi(u_i)$.

Summarizing, ξ is a fixpoint of Ψ, and hence $\xi(u) \subseteq \gamma(u)$ for all set constants u.

Note that Lemma 1 and Corollary 1 do not require the selector-closedness of $\mathcal{T}$, whereas this is required in the proof of Lemma 2.

Theorem 1. *For all u_1, u_2: $\gamma(u_1) \subseteq \gamma(u_2)$ iff $\sigma(u_1) \subseteq \sigma(u_2)$.*

Proof. $\Leftarrow$: From $\sigma(u_1) \subseteq \sigma(u_2)$ the relation $\gamma(u_1) \subseteq \gamma(u_2)$ follows, by Lemma 2, since the construction $\{t \mid$ all finite cuts of t are in $\sigma(u)\}$ is monotone in u.

$\Rightarrow$: Let $\gamma(u_1) \subseteq \gamma(u_2)$. Since $\sigma(u_1) \subseteq \gamma(u_1)$, we obtain $\sigma(u_1) \subseteq \gamma(u_2)$. Lemma 1 shows that $\sigma(u_2)$ is the subset of finite cuts of trees in $\gamma(u_2)$, hence $\sigma(u_1) \subseteq \sigma(u_2)$.

It is possible to define a set constant $\top$ that contains all trees under the greatest fixpoint semantics and all finite trees under the least fixed point semantics. The definition is: $\top ::= \{\bot\} \cup (c_1\ \top \ldots \top) \cup \ldots (c_K\ \top \ldots \top)$.

The set constant $\top$ contains all trees (or finite, respectively), since the set of all trees (finite, respectively) is a greatest fixpoint (least fixpoint, respectively) for this equation. I.e. $\sigma(\top) = \mathcal{L}(\mathcal{A})$, and $\gamma(\top) = \mathcal{T}$.

3.2 Decidability and Complexity

From Theorem 1, we obtain the corollary.

Proposition 1. *The subset relationship of set constants w.r.t. the greatest fixpoint semantics is decidable.*

Proof. Theorem 1 shows that this is equivalent to the subset relationship w.r.t. the least fixpoint semantics. The subset problem reduces to the set constraint problem via a direct encoding of each equation as two set constraints. Induction on the number of iterations of Ψ can be used to prove both directions for the claim: the solutions to the encoded subset problem decode exactly to those sc-interpretations satisfying the subset relationship

The subset relationship w.r.t. least fixpoint semantics can be decided using the results on solvability of set constraints, see the overview [Aik94].

Remark 3. The methods in the paper [CP98] on co-definite set constraints also use infinite trees. However, the results in [CP98] are not applicable, since the encoding above results in constraints that are not covered by the syntactic restrictions in [CP98], since our set constraints permit left hand sides e_l in $e_l \subseteq e_r$ of the form $(c\ u_1 \dots\ u_n)$ where $\mathrm{ar}(c) \geq 2$.

Remark 4. The special form of our defining equation allows to apply the tool of tree automata. The equations Eq_i define a non-deterministic tree automaton with states $U_1, \dots, U_K$. The rules are $U_i(\bot)$ for all $i = 1, \dots, K$, and if $(c\ u_1 \dots u_m)$ is a component in the right-hand side of Eq_i, then $(c\ u_1(t_1)\ \dots\ u_n(t_n)) = U_i(c\ t_1 \dots t_n)$. The sets $\sigma(U_i)$ are exactly the finite trees accepted by the tree automaton with the rules corresponding to the equations $Eq_j, j = 1, \dots, K$, and with accepting state U_i.

Theorem 2. *The complexity of the problem whether two set constants are in the subset relation is EXPTIME-complete.*

Proof. By Theorem 1, it is sufficient to use least fixpoint semantics. The complexity of satisfiability of set constraints, and hence by Proposition 1 set constant-subset problem is in NEXPTIME, see [BGW93] and [AKVW93], which gives an upper bound on the complexity. However, the form of our definitions is syntactically restricted, which allows to derive a sharper bound. The subset-problem for set-constants is reducible to the inclusion problem of the languages accepted by tree automata via the translation in Remark 4, where the states of the tree automata correspond to the set constants, and the accepted language is the set of finite trees defined by the set constants using least fixed points (see Theorem 1). The latter is EXPTIME-complete (see [Sei90,CDG$^+$97]). This allows us to

conclude that the subset-problem for set constants is in EXPTIME. For hardness, we have to argue that every tree automaton can be encoded as a set of equations for set constants using least fixed-point semantics. This, however, follows directly from Remark 4, where we have to add the treatment of the constant $\bot$, which does not make a difference in complexity.

Hence, the subset-problem for set-constants is EXPTIME-complete.

This also implies that the algorithms from tree automata to solve the inclusion problem, can be transferred to our problem.

4 An Extended Form of Equations for Set Constants

In this section we show that extending the syntax for set constants by allowing top-level set constants on the right-hand side does not increase the expressive power of set constants.

Assume the defining equations for set constants are allowed to be of the extended form

$$(Eqx_i): \quad U_i = \{\bot\} \cup r_{i,1} \cup \ldots \cup r_{i,n_i}$$

where $r_{i,j}$ may be a set constant, or an expression $(c\ u'_1 \ldots u'_{\mathrm{ar}(c)})$, where u'_j are set constants. With $rhsx_{Eq}(U_i)$ we denote the right-hand side of Eqx_i. We can assume that every set constant occurs at most once as a top-level component in each of the right-hand sides.

Definition 3. *We define the relation $\rightarrow_{eq}$ as follows:*
If there is an equation $u_i = \{\bot\} \cup r_{i,1} \cup \ldots \cup r_{i,k_i}$, and some $r_{i,h}$ is the set constant u_j, then $u_i \rightarrow_{eq} u_j$. The transitive closure of $\rightarrow_{eq}$ is denoted as $\rightarrow^+_{eq}$.

The following transformation steps transforms the set of equations Eqx_i into an equivalent set of equations for set constants w.r.t. the greatest fixpoint semantics. The first transformation identifies unrestricted set constants, and replaces them by an equation yielding all trees, whereas the second transformation removes the occurrences of set constants in the right-hand side by instantiating them.

R1 Let u be a set constant, such that there is a cycle. i.e., $u \rightarrow^+_{eq} u$. Replace the equation for u by the $\top$-equation, i.e. by $u ::= \{\bot\} \cup (c_1\ u \ldots u) \cup \ldots \cup (c_K\ u \ldots u)$.

R2 This rule is applicable only if there are no $\rightarrow^+_{eq}$-cycles.
Select a set constant u that occurs at top-level in some right-hand side of an equation. Replace all these occurrences of u as component in the right-hand sides in all equations by $rhs(u)$. After the replacement, there is a simplification as follows: Eliminate double occurrences of components in right-hand sides.

Pragmatically, a good strategy for replacement is to instantiate in a bottom-up fashion w.r.t. $\rightarrow_{eq}^{+}$.

Lemma 3. *The transformation steps terminate.*

Proof. This is obvious, since every step removes an occurrence of a set constant in the right-hand side of equations.

Proposition 2. *The greatest fixpoint semantics $\gamma(u_i)$ is unchanged by the transformation steps*

Proof. If for some set constant u_i, we have $u_i \rightarrow_{eq}^{+} u_i$, then by the greatest fixpoint semantics, we have $\gamma(u_i) = \mathcal{T}$, hence $\gamma(u_i)$ remains the same, if we transform the equation for u_i according to (R1). For other set constants u_i occurring in the right hand side, the value under the greatest fixed point is not changed by the (R1)-transformation, since $\gamma(u_i)$ remains the same.

The replacement in the second transformation step does not change the greatest fixpoint γ.

Theorem 3. *The complexity of the decision problem for the inclusion problem for set constants defined in the extended language is EXPTIME-complete.*

Proof. An exhaustive application of the first transformation rule can be done in polynomial time: The replacements lead to a linear space increase. Every rule application requires linear time for the cycle-detection, hence the overall time is $O(n^2)$.

The exhaustive application of the second transformation rule is polynomial, since the right-hand sides are bounded by the total size of all right-hand sides, which is a quadratic upper bound for the size increase. The worst case time complexity is $O(n^3)$. After the transformation, we apply Theorem 2.

5 Application to Extended Lambda-Calculi with Case and Constructors

We apply the results to two different extended lambda calculi. Their syntax is different, however, at the level of interfacing with the question of inclusion of set constants, the properties are the same.

The first calculus is a call-by-name lambda-calculus with case and constructors. A slight variation of the syntax and equality as used in [Sch00] is as follows. There is a finite set of constructors. Every constructor c comes with an arity $\mathrm{ar}(c) \in \mathbb{N}_0$. There is also a set of types, which provides a partitioning of the

constructors into types, $A_1, \ldots, A_k$. The syntax of expressions E and patterns P is as follows:

$$E ::= V \mid \lambda V.E \mid (E\ E) \mid (c\ E_1 \ldots\ E_{\mathrm{ar}(c)}) \mid (\mathtt{case}_{\mathtt{A}}\ E\ (P_1\mathtt{->}E_1) \ldots (P_n\mathtt{->}E_n))$$
$$P ::= (c\ V_1 \ldots V_{\mathrm{arc}})$$

The constructors in the patterns in a case-expression starting with $\mathtt{case}_A$ must be in the type A. The symbol V denotes variables. The reduction rules are beta- and case-reduction.

The second calculus is a call-by-need lambda calculus (see [SSSS05]). The syntax is as follows. There is a set of constructors, denoted as c, coming with an arity $\mathrm{ar}(c) \in \mathbb{N}_0$. There is a partitioning of the constructors into types, $A_1, \ldots, A_k$. The syntax of expressions E and patterns P is as follows:

$$E ::= V \mid \lambda V.E \mid (E\ E) \mid (c\ E_1 \ldots\ E_{\mathrm{ar}(c)}) \mid (\mathtt{case}_{\mathtt{A}}\ E\ (P_1\mathtt{->}E_1) \ldots (P_n\mathtt{->}E_n))$$
$$\mid (\mathtt{seq}\ E\ E) \mid (\mathtt{letrec}\ \{V_1 = E_1; \ldots; V_n = E_n\}\ \mathtt{in}\ E$$
$$P ::= (c\ V_1 \ldots V_{\mathrm{ar}(c)})$$

The constructors in the patterns in a case-expression starting with $\mathtt{case}_A$ must belong to the type A, V denotes variables. The reductions rules are variants of beta- and case-reductions, a `seq`-reduction, and several reduction rules for the `letrec`.

From now on we do not distinguish the two calculi, and formulate the following properties in an independent way, the proofs are either straightforward or in the respective papers [SSSS05]. In both calculi there is a notion of evaluation, which is a normal order reduction to a weak head normal form. If this evaluation terminates with a WHNF, then this is denoted as $t{\Downarrow}$. Equality of expressions is defined using the contextual preorder:
$s \leq_c t$ iff for all contexts $C : C[s]{\Downarrow} \Rightarrow C[t]{\Downarrow}$, and $s \sim_c t$ iff $s \leq_c t \wedge t \leq_c s$.
Here a context means an expression with a single hole, where a term can be plugged in.

The relations $\leq_c$ and $\sim_c$ are compatible with contexts; the latter relation can be seen as maximal sensible equality relation on the expressions. There are two kinds of values, constructor values of the form $(c\ t_1 \ldots t_{\mathrm{ar}(c)})$, and abstractions $(\mathtt{letrec}\ E\ \mathtt{in}\ \lambda x.t)$ with a surrounding environment.

The following classification of expressions holds:

Proposition 3. *For every closed expression t one of the following holds:*

1. $t \sim_c \Omega$, *where* $\Omega = (\lambda x.(x\ x))\ (\lambda y.(y\ y))$.
2. *There exists a closed expression* $(c\ t_1 \ldots t_{\mathrm{ar}(c)})$, *such that* $t \sim_c (c\ t_1 \ldots t_{\mathrm{ar}(c)})$.
3. *There exists a closed expression* $(\mathtt{letrec}\ E\ \mathtt{in}\ \lambda x.t')$ *such that* $t \sim_c (\mathtt{letrec}\ E\ \mathtt{in}\ \lambda x.t')$.

Furthermore, $(c\ t_1 \ldots t_n) \not\sim_c \Omega \not\sim_c (\mathtt{letrec}\ E\ \mathtt{in}\ \lambda x.t) \not\sim_c (c\ t_1 \ldots t_n)$ *and* $(c_1\ s_1 \ldots s_n) \sim_c (c_2\ t_1 \ldots t_m) \Leftrightarrow {}_1 = c_2, n = m$ *and* $\forall i = 1, \ldots, n : s_i \sim_c t_i$.

We construct $\mathcal{T}$ as follows: First let $\mathcal{T}_0 = \Lambda/\sim_c$, where Λ is the set of expressions. Using this quotient, the set constants w.r.t. Λ are defined.

To explain the definition of the constants, we follow [SSSS05]. The corresponding notion of demands in [Sch00] is more general and permits as a fragment the language of set constants.

The Λ-set constants $U_{\Lambda,i}$ are defined over $\mathcal{T}_0$ using a greatest fixpoint of the corresponding sc-interpretations. For every set constant $U_{\Lambda,i}$ there is a defining rule

$$(Eq_{\Lambda,i}): \quad U_{\Lambda,i} = \{\bot\} \cup r_{i,1} \cup \ldots \cup r_{i,n_i}$$

where $r_{i,j}$ may be *Fun*, or an expression $(c\ u'_1 \ldots u'_{\mathrm{ar}(c)})$, where u'_j are Λ-set constants. The set constant *Fun* represents all expressions that can be reduced to an expression of the form $(\texttt{letrec}\ E\ \texttt{in}\ \lambda x.t)$, or equivalently, that are contextually equivalent to a term of the form $(\texttt{letrec}\ E\ \texttt{in}\ \lambda x.t)$.

For a translation into the set constant mechanism, we put all abstractions into one further equivalence class.

Let $\approx$ be the greatest relation on $\mathcal{T}_0$ with

- $(\texttt{letrec}\ E\ \texttt{in}\ \lambda x.t) \approx (\texttt{letrec}\ E'\ \texttt{in}\ \lambda x'.t')$ for all expressions $(\texttt{letrec}\ E\ \texttt{in}\ \lambda x.t), (\texttt{letrec}\ E'\ \texttt{in}\ \lambda x'.t')$.
- $(c\ t_1 \ldots t_n) \approx (c\ s_1 \ldots s_n) \Rightarrow \forall i : t_i \approx s_i$.
- $(c_1\ t_1 \ldots t_n) \not\approx (c_2\ s_1 \ldots s_m)$ for all s_i, t_i if $c_1 \neq c_2$
- $(c_1\ t_1 \ldots t_n) \not\approx (\texttt{letrec}\ E\ \texttt{in}\ \lambda x.t)$ for all t_i, t, E.
- $(c_1\ t_1 \ldots t_n) \not\approx \Omega$ for all t_i.
- $(\texttt{letrec}\ E\ \texttt{in}\ \lambda x.t) \not\approx \Omega$ for all t, E.

The relation $\approx$ can be constructed as greatest fixpoint and is an equivalence relation. We define $\mathcal{T} := (\mathcal{T}_0)/\approx$. We denote $\lambda = [(\texttt{letrec}\ E\ \texttt{in}\ \lambda x.t)]_\approx$ and $\bot := [\Omega]_\approx$. The set $\mathcal{T}$ contains all finite trees over the constructors, λ, and $\bot$, and is selector-closed, which follows easily from the properties of expressions in Λ. Note that the set $\mathcal{T}$ does not contain all possible infinite trees, since $\mathcal{T}$ is countable, however, there are also infinite trees, e.g. $(\texttt{letrec}\ x = \texttt{cons}\ 1\ x\ \texttt{in}\ x)$, for a binary construct `cons`, which results in an infinite list of 1s as entries.

The translation τ from $\mathcal{T}_0$ into $\mathcal{T}$ can now be used for elements and the set constants.

Note that the treatment of $\bot$ in the contextual preorders is compatible with the definition and use of $\bot$ for trees and set constants in the value language, since it implies that $\bot \leq_c t$ for all expressions t and that a set constant $\bot$ defined as $\{\bot\}$ is contained in all other set constants.

Let γ_0 be the greatest fixpoint of sc-interpretations w.r.t. the equations in $\mathcal{T}_0$.

Proposition 4. *For all Λ-set constants u_1, u_2 : $\gamma(\tau(u_1)) \subseteq \gamma(\tau(u_2))$ iff $\gamma_0(u_1) \subseteq \gamma_0(u_2)$*

Proof. Let $\gamma_0(u_1) \subseteq \gamma_0(u_2)$, and let $t \in \gamma(\tau(u_1))$. By the construction of the set $\mathcal{T}$, there is an expression e_t with $\tau(e_t) = t$. Using the translation and co-induction, it is clear that $e_t \in \gamma_0(u_1)$, and hence also $e_t \in \gamma_0(u_2)$. Again, using the translation, we obtain that $t \in \gamma(\tau(u_1))$.

Let $\gamma(\tau(u_1)) \subseteq \gamma(\tau(u_2))$ and let $e \in \gamma_0(u_1)$. The same reasoning as above shows that the tree $\tau(e)$ is contained in $\gamma(\tau(u_1))$, and hence $\tau(e) \in \gamma(\tau(u_2))$. Via the translation and using co-induction, we obtain $e \in \gamma_0(u_2)$.

Since all the preconditions are satisfied, the following holds:

Theorem 4. *The inclusion problem for set constants in both lambda-calculi is EXPTIME-complete.*

For example, the set constants $\top$, `Bot`, `List`, and `BotElem` can be defined as

$$\begin{array}{ll} \top & ::= \{\bot\} \cup \mathit{Fun} \cup (c_1\ \top \ldots \top) \cup \ldots \cup (c_k\ \top \ldots \top) \\ \texttt{Bot} & ::= \{\bot\} \\ \texttt{List} & := \{\bot\} \cup \texttt{Nil} \cup \texttt{cons}\ \top\ \texttt{List} \\ \texttt{BotElem} & := \{\bot\} \cup (\texttt{cons}\ \texttt{Bot}\ \top) \cup (\texttt{cons}\ \top\ \texttt{BotElem}) \end{array}$$

A nontrivial inclusion being $\texttt{BotElem} \subseteq \texttt{List}$, which can be proved using the methods from tree automata as well as directly using co-induction.

Theorem 2 justifies the use of algorithms from tree automata to solve the inclusion problem, which can be transferred to the inclusion problem of set constants in the respective languages.

6 Conclusion and Further Research

We have proved how methods from tree automata can be used to solve the inclusion-problem for co-inductively defined set constants in static analysis of lazy functional programming languages. The algorithm for the set constants we considered is EXPTIME-complete and we are sure that for most cases arising in practice this will not result in the worst case running time.

Future work may investigate the set-inclusion problem for more expressive value-languages, e.g. including intersections, or even for the full demand language in [Sch00].

References

[Aik94] A. Aiken. Set constraints: Results, applications, and future directions. In *Second Workshop on the Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*, pages 171–179, Orcas Island, Washington, May 1994. Springer-Verlag.

[AKVW93] Alexander Aiken, Dexter Kozen, Moshe Y. Vardi, and Edward L. Wimmers. The complexity of set constraints. In *Proc. CSL 1993*, pages 1–17, Swansea, Wales, 1993.

[BGW93] Leo Bachmair, Harald Ganzinger, and Uwe Waldmann. Set constraints are the monadic class. In *Proc. 8th Proc Symp. Logic in Computer Science*, pages 75–83, Swansea, Wales, 1993.

[CDG+97] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 1997. release October, 1rst 2002.

[CP98] Witold Charatonik and Andreas Podelski. Co-definite set constraints. In Tobias Nipkow, editor, *Proceedings of the 9th International Conference on Rewriting Techniques and Applications*, volume 1379 of *LNCS*, pages 211–225. Springer-Verlag, 1998.

[Nöc92] Eric Nöcker. Strictness analysis by abstract reduction in orthogonal term rewriting systems. Technical Report 92-31, University of Nijmegen, Department of Computer Science, 1992.

[Nöc93] Eric Nöcker. Strictness analysis using abstract reduction. In *Functional Programming Languages and Computer Architecture*, pages 255–265. ACM Press, 1993.

[Sch00] Marko Schütz. *Analysing demand in nonstrict functional programming languages*. Dissertation, J.W.Goethe-Universität Frankfurt, 2000. available at `http://www.ki.informatik.uni-frankfurt.de/papers/marko`.

[Sei90] Helmut Seidl. Deciding equivalence of finite tree automata. *SIAM Journal of Computing*, 19:424–437, 1990.

[SSPS95] Manfred Schmidt-Schauß, Sven Eric Panitz, and Marko Schütz. Strictness analysis by abstract reduction using a tableau calculus. In *Proc. of the Static Analysis Symposium*, number 983 in Lecture Notes in Computer Science, pages 348–365. Springer-Verlag, 1995.

[SSSS04] Manfred Schmidt-Schauß, Marko Schütz, and David Sabel. On the safety of Nöcker's strictness analysis. Technical Report Frank-19, Institut für Informatik. J.W.Goethe-University, 2004.

[SSSS05] Manfred Schmidt-Schauß, Marko Schütz, and David Sabel. A complete proof of the safety of Nöcker's strictness analysis. Technical Report Frank-20, Institut für Informatik. J.W.Goethe-University, 2005.

[Wad87] Phil Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 12. Ellis Horwood Limited, Chichester, 1987.