Theoretical Informatics and Applications Will be set by the publisher
Informatique Théorique et Applications

DECIDING INCLUSION OF SET CONSTANTS OVER INFINITE NON-STRICT DATA STRUCTURES

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

Abstract. Various static analyses of functional programming languages that permit infinite data structures make use of set constants like `Top`, `Inf`, and `Bot`, denoting all terms, all lists not eventually ending in Nil, 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 in the set of all, also infinite, computable trees, where all term constructors are non-strict. This paper proves decidability, in particular DEXPTIME-completeness, of inclusion of co-inductively defined sets by using algorithms and results from tree automata and set constraints. The test for set inclusion is required by certain strictness analysis algorithms in lazy functional programming languages and could also be the basis for further set-based analyses.

1991 Mathematics Subject Classification. 68N18, 03B40, 68Q25, 68Q45.

1. Introduction

The compilation of programming languages requires static analysis for the purpose of error-detection and for improving the optimization possibilities. In non-strict functional languages like Haskell [29] or Clean [30] the compiler requires so-called strictness analysis that statically determines whether an argument of a function f can be evaluated before evaluating the body of the function. There are different variants of strictness analysis methods, based on abstract interpretation

Keywords and phrases: functional programming languages, lambda calculus, strictness analysis, set constraints, tree automata

[1] Institut für Informatik, Johann Wolfgang Goethe-Universität, Postfach 11 19 32, D-60054 Frankfurt, Germany, e-mail: `schauss@ki.informatik.uni-frankfurt.de`

[2] Dept. of Mathematics and Computing Science, University of the South Pacific, Suva, Fiji Islands, e-mail: `schutz_m@usp.ac.fj`

(e.g. [1, 7, 8, 13, 22, 38]), projections (e.g. [19, 28, 39]), non-standard type systems (e.g. [12, 16–18]) or abstract reduction [24]. A comparison of several approaches can be found in [26, 27].

To find out whether a unary function f is *strict*, one has to check the termination behavior of $f(\bot)$, where $\bot$ is a non-terminating expression. If $f(\bot)$ does not terminate, then f is strict in its argument, and the compiler can exploit this knowledge by generating more efficient code by rearranging the evaluation order by evaluating the argument before entering the body of f. The test whether a binary function f is strict in its second argument means to check $(f\ t\ \bot)$ for non-termination for all arguments t, which will be represented as analyzing $(f\ \top\ \bot)$. We are particularly interested in static analysis methods based on the operational semantics, where the domain of expressions is used. Since non-strict functional languages permit to program with streams, i.e. infinite (lazy) lists, the domain of all computable, finite and infinite, data-terms is a natural choice. Since we address analysis in non-strict functional languages, where all constructors are non-strict, all the data terms are lifted, which means that non-terminating expressions, represented by the symbol $\bot$, can occur everywhere in a data term.

Static analysis in lazy functional languages uses abstract sets, such as `Top` (also denoted as $\top$) for all expressions, `Bot` (also denoted as $\bot$) for all non-terminating expressions, and `Inf` for all infinite lists (lists for which the length-computation does not terminate). There is a proposal for static analyses including structured data based on 4 predefined sets [38]. In [23–25], strictness analysis used a language for abstract sets extending the 4 fixed sets, where the language of sets enforces $\bot \subseteq A \subseteq \top$ for all definable sets A. However, only the inclusions that syntactically follow from $\bot \subseteq A \subseteq \top$ for all A, like $(\texttt{Cons}\ \bot\ u) \subseteq (\texttt{Cons}\ u\ \top)$ are used, but no attempt is made to exploit other valid inclusions. The strictness analysis in [32, 35] used an inclusion check for set constants in the most powerful version of its loop detection rules, but relied on an incomplete decision algorithm to detect these inclusions. A language for sets is also used in the reconstruction and correctness proof of Nöcker's strictness analysis [34]. In most cases, these are recursive definitions of sets with a greatest fixed-point semantics in the domain of computable (also infinite) data terms over non-strict constructors. Strictness analysis usually requires that the set of expressions that are the semantics of set constants are down-closed w.r.t. an ordering inherited from $\bot \leq t$ for all t, since then continuity arguments can be used for correctness proofs. Translated into a term language, this means that $\bot$ can be used for cutting trees.

In this paper we formulate the set constants inclusion problem (SCIP) as follows: Assume given a language that can define sets of (finite and infinite) terms over a signature of constructors including the special constant $\bot$, and given an admissible subset $\mathcal{T}$ of all (including infinite) trees, and two set constants u_1, u_2 together with their recursive definition. The set $\mathcal{T}$ may e.g. be the set of all computable finite and infinite trees. First an interpretation γ for the two sets is computed using the greatest fixpoint of the definitional equations w.r.t. $\mathcal{T}$, and then an inclusion test $\gamma(u_1) \subseteq \gamma(u_2)$ has to be performed. We assume that the set definitions u are restricted to down-closed sets, i.e., if $t \in u$, then also $t[\bot/p] \in u$ for every position

p. This is required for correct application of the inclusion problem within strictness analysis as it is formulated in [34]. In a first order language of terms, the inclusion problem can be reconstructed over finite and infinite terms with $\bot$-entries that play the role of cut-markers.

A main result is that the inclusion problem for set constants can be solved by computing the least fixpoint instead of the greatest fixpoint (Theorem 2.11). The proof method is via cuts of trees and elementary set theory arguments. Since we can use the computation of least fixpoints for SCIP, we can apply tree automata techniques to solve the inclusion problem: A straightforward translation shows that the SCIP is in DEXPTIME (Proposition 2.19). The hardness result for the set constant inclusion problem is derived from the DEXPTIME-hardness of the problem whether all terms are accepted by non-deterministic tree automata [11]. This finally shows that the set constant inclusion problem is DEXPTIME-complete (Theorem 2.21). The technique also allows to prove that the inclusion problem for bot-free SCIP is DEXPTIME-complete (Theorem 2.21), i.e. where no $\bot$ is used in the defining equations, and the greatest fixpoint is computed. Since we can vary the base set $\mathcal{T}$, the results hold for all infinite trees over the signature as well as for all computable trees of the given signature. In the case of all infinite trees, there is a connection to tree languages accepted by special Büchi-automata (see Remark 2.22).

A formalism related to our set constant inclusion problem are set constraints. Set constraints [2, 3] is a formalism that can express subset relations between sets of terms, and provides methods for solving these constraints. It can be applied in a class of static program analysis methods (set-based analysis). In general, sets of finite terms are considered, but there are also results for infinite trees: the co-definite set constraints in [9], and an extension in [15]. Other work on set constraints using infinite (regular) trees is [20] and [31]. The paper [31] defines a constraint language which allows complements and intersections and shows that the satisfiability problem of set constraints is undecidable in the domain of regular terms. For our $\bot$-free set constants definitions the satisfiability problem in the set of regular terms is decidable, since the set of regular terms is admissible (see Definition 2.1).

The variant of SCIP with a computation of the least fixpoint can be encoded as a set constraint problem, however, in general, our SCIP rather is a "two-stage set constraint": first compute a (unique) solution, then apply the solution to a further constraint, which in our case is an inclusion constraint.

This paper is structured as follows. First the language for defining set constants is given. Subsection 2.2 defines the inclusion problem for set constants. In subsection 2.3 we explore the properties of least and greatest fixpoint in detail. In subsection 2.4 we show the close relationship between bot-closed and bot-free set constants definition for our inclusion problem w.r.t. the greatest fixpoint semantics. Based on these results, in subsection 2.5 we show that the set constant inclusion problem can be solved and its complexity be determined using results from tree automata. In section 3 we show how the results are transferred into and

used in untyped extended lambda-calculi, and also in static analyses, in particular for strictness analysis in lazy functional programming languages.

2. The Language and the Inclusion Problem

2.1. Syntax of the Language of Values

We use a finite signature Σ of function symbols c coming with an arity $\mathrm{ar}(c) \geq 0$. There is one special constant $\bot$ with $\mathrm{ar}(\bot) = 0$. The function symbols in $\Sigma \setminus \{\bot\}$ are called *constructors*. The syntax for terms E is:

$$E ::= (c\ E_1 \ldots E_{\mathrm{ar}(c)}) \text{ where } c \in \Sigma \text{ is a function symbol}$$

We define $\mathcal{T}(\Sigma)$ as the set of all (finite) terms E that can be generated using this grammar. We use *positions* in terms, denoted p, q, as sequences of positive integers following Dewey notation. The concatenation of two sequences p, q is written $p.q$, the subtree of t at position p is denoted as $t_{|p}$, and the label at position p as $t(p)$. The set of all positions of a term t is denoted as $D(t)$, which is a prefix-closed set. This also allows to define and treat infinite terms, where $D(t)$ is an infinite prefix-closed set of positions, and the arities of function symbols are respected. The set of all trees over the signature Σ is denoted as $\mathcal{T}_\infty(\Sigma)$. Maximal positions correspond to leaves in trees. The depth of a finite tree t is the maximal length of a position in $D(t)$.

Definition 2.1. A subset $\mathcal{T} \subseteq \mathcal{T}_\infty(\Sigma)$ is called *subtree-closed*, iff every subtree of a tree $t \in \mathcal{T}$ is also contained in $\mathcal{T}$. A subset $\mathcal{T} \subseteq \mathcal{T}_\infty(\Sigma)$ is called *admissible*, iff $\mathcal{T}(\Sigma) \subseteq \mathcal{T}$, and $\mathcal{T}$ is subtree-closed.

In the following we are particularly interested in the set $\mathcal{T}_{comp}$ of all computable trees, which is admissible. We will also exploit the fact that the set $\mathcal{T}_\infty(\Sigma)$ is admissible.

Definition 2.2. A tree $t' \in \mathcal{T}_\infty(\Sigma)$ is a *cut* of a tree t, if $D(t') \subseteq D(t)$, and $\forall p \in D(t') : t'_{|p} \neq t_{|p} \Rightarrow t'_{|p} = \bot$.
A tree t' is a *finite cut* of a tree t, if t' is a finite tree and a cut of t. A tree t' is a *finite cut* of a tree t *at depth* k, if t' is a finite cut of t, t' has depth k and $\forall p \in D(t') : t'_{|p} \neq t_{|p} \Rightarrow |p| = k$.

A finite cut can be seen as cutting away subtrees by replacing them with a $\bot$-leaf until the resulting tree is finite. Note that for every admissible set $\mathcal{T}$ of trees, all finite cuts are also contained in $\mathcal{T}$, however, arbitrary cuts are not necessarily in $\mathcal{T}$.

2.2. Set Constants

In this and the following subsection the definitions, lemmas and theorems are parameterized by an admissible set $\mathcal{T} \subseteq \mathcal{T}_\infty(\Sigma)$. Later we will show that the

results are independent of the specific set $\mathcal{T}$. If necessary, we will indicate with an index $\mathcal{T}$ in the notation the dependence on $\mathcal{T}$.

Definition 2.3 (Set Constants Definition)**.** A *set constants definition* is a tuple $(\Sigma, \mathcal{T}, \mathcal{U}, \mathcal{EQ})$ where Σ is a finite signature of function symbols, $\mathcal{T}$ is an admissible set of possibly infinite trees, $\mathcal{U} = \{u_1, \ldots, u_K\}$ are finitely many set constants with $\mathcal{U} \cap \Sigma = \emptyset$, and $\mathcal{EQ} = \{Eq_1, \ldots, Eq_K\}$ is a set of defining rules, where for every set constant $u_i \in \mathcal{U}$ there is exactly one rule, named (Eq_i)

$$u_i \quad = \quad r_{i,1} \cup \ldots \cup r_{i,n_i}$$

where $r_{i,j}$ is $\bot$ or an expression $(c\ u_1 \ldots u_{\mathrm{ar}(c)})$, and $u_j \in \mathcal{U}$ are set constants. With $rhs_{Eq}(u)$ we denote the right-hand side of Eq_i, if $u = u_i$.

A set constants definition is called *bot-closed* if $\forall i \in \{1, \ldots, K\}$ there exists $r_{i,j}$ with $r_{i,j} = \bot$. It is called *bot-free* if for all $i, j : r_{i,j} \neq \bot$.

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

We extend interpretations ψ to ψ^e as follows:

$$\begin{array}{lcl} \psi^e(\bot) & := & \{\bot\} \\ \psi^e(c\ u_1\ \ldots\ u_{\mathrm{ar}(c)}) & := & \{(c\ a_1 \ldots a_{\mathrm{ar}(c)}) \mid a_i \in \psi(u_i)\} \\ \psi^e(r_1 \cup r_2) & := & \psi^e(r_1) \cup \psi^e(r_2) \end{array}$$

In abuse of notation, we write ψ instead of ψ^e in the following.
For all i the equations Eq_i for set constants define an operator Ψ on interpretations as follows:

$$\Psi(\psi)(u_i) := \psi(rhs_{Eq}(u_i)).$$

Let $\mathcal{SC}_\mathcal{T}$ be the set of all interpretations. Then $(\mathcal{SC}_\mathcal{T}, \leq)$ is a partially ordered set. Furthermore, every $\mathcal{S} \subseteq \mathcal{SC}_\mathcal{T}$ has a least upper bound $lub(\mathcal{S}) \in \mathcal{SC}_\mathcal{T}$ as well as a greatest lower bound $glb(\mathcal{S}) \in \mathcal{SC}_\mathcal{T}$ given by the definitions:

$$lub(\mathcal{S})(u_i) = \bigcup_{\psi \in \mathcal{S}} \psi(u_i) \qquad \text{and} \qquad glb(\mathcal{S})(u_i) = \bigcap_{\psi \in \mathcal{S}} \psi(u_i)$$

The partially ordered set $(\mathcal{SC}_\mathcal{T}, \leq)$ is a complete lattice, isomorphic to $(\mathcal{T}, \subseteq)^K$. The operator Ψ is monotone, hence we can apply the Knaster-Tarski fixpoint theorem and thus the least and the greatest fixpoint of Ψ exist. We define $\sigma_\mathcal{T}$ and $\gamma_\mathcal{T}$ to be respectively the least and greatest fixed points of Ψ. The index $\mathcal{T}$ will be omitted if it is clear from the context.
Now we can state our inclusion problem:

Definition 2.4 (Set Constant Inclusion Problem (SCIP))**.** Given a set constants definition $(\Sigma, \mathcal{T}, \mathcal{U}, \mathcal{EQ})$ and two set-constants $u_i, u_j \in \mathcal{U}$. The *set constant inclusion problem* is the question whether or not $\gamma_\mathcal{T}(u_i) \subseteq \gamma_\mathcal{T}(u_j)$.

The set constant $\top$ with definition $\top = \bot \cup (c_1\ \top \ldots \top) \cup \ldots (c_N\ \top \ldots \top)$ for $\Sigma = \{\bot, c_1, \ldots, c_N\}$ is the full set $\mathcal{T}$ under the greatest fixpoint semantics and the set of finite trees under the least fixed point semantics. I.e. $\sigma(\top) = \mathcal{T}(\Sigma)$, and $\gamma(\top) = \mathcal{T}$.

In the following we distinguish two kinds of SCIPs: A SCIP is called *bot-closed* (*bot-free*, respectively) if the corresponding set constants definition is bot-closed (bot-free, respectively).

Note that the bot-free case is the general one if the special properties of $\bot$ as cut-marker are not used, which is the case if $\mathcal{T} = \mathcal{T}_\infty(\Sigma)$.

We extend the application of Ψ to sets as usual as $\Psi(\mathcal{S}) := \{\Psi(\psi) \mid \psi \in \mathcal{S}\}$.

Lemma 2.5. *The operator Ψ is (upper) continuous [14], i.e. for every directed set $\mathcal{S} \subseteq \mathcal{SC}_\mathcal{T}$: $lub(\Psi(\mathcal{S})) = \Psi(lub(\mathcal{S}))$*

Proof. Let $\mathcal{S} \subseteq \mathcal{SC}_\mathcal{T}$ be a directed set, then $lub(\Psi(\mathcal{S})) \leq \Psi(lub(\mathcal{S}))$ follows from monotonicity of Ψ. For the other direction let $\psi \in \mathcal{S}$ and $u_i \in \mathcal{U}$ be arbitrary but fixed. Transforming $\Psi(\psi)(u_i)$ and $\Psi(lub(\mathcal{S}))(u_i)$ gives the equations:

$$\begin{aligned}\Psi(\psi)(u_i) &= \textstyle\bigcup_{k\in\{1,\ldots,n_i\}}\{c_{i,k}\ a_{i,k,1} \ldots a_{i,k,\mathrm{ar}(c_{i,k})} \mid a_{i,k,l} \in \psi(u_{i,k,l})\}\\ \Psi(lub(\mathcal{S}))(u_i) &= \textstyle\bigcup_{k\in\{1,\ldots,n_i\}}\{c_{i,k}\ a_{i,k,1} \ldots a_{i,k,\mathrm{ar}(c_{i,k})} \mid a_{i,k,l} \in lub(\mathcal{S})(u_{i,k,l})\}\end{aligned}$$

Now let $t \in \Psi(lub(\mathcal{S}))(u_i)$. If $t = \bot$ then obviously $t \in \Psi(\psi)(u_i)$. Otherwise $t = c_{i,k}\ a_{i,k,1}\ \ldots a_{i,k,\mathrm{ar}(c_{i,k})}$ where $a_{i,k,l} \in lub(\mathcal{S})(u_{i,k,l})$ for $l = 1, \ldots, \mathrm{ar}(c_{i,k})$, i.e. $a_{i,k,l} \in \bigcup_{\psi\in\mathcal{S}} \psi(u_{i,k,l})$. I.e., there exist $\psi_1, \ldots \psi_l$ in $\mathcal{S}$ with $a_{i,k,l} \in \psi_l(u_{i,k,l})$. Since $\mathcal{S}$ is directed, there exists $\psi_m \in \mathcal{S}$ with $\psi'_k \leq \psi_m$ for $k' = 1, \ldots, l$. Thus, $a_{i,k,l} \in \psi_m(u_{i,k,l})$ for $l = 1, \ldots, \mathrm{ar}(c_{i,k})$. Hence, $t \in \Psi(\psi_m)(u_i)$. Since $\Psi(\psi_m)(u_i) \subseteq \bigcup_{\psi\in\mathcal{S}} \Psi(\psi)(u_i)$, we have $t \in lub(\Psi(\mathcal{S}))(u_i)$. Thus we have shown $\Psi(lub(\mathcal{S}))(u_i) \subseteq lub(\Psi(\mathcal{S}))(u_i)$. Since this holds for all $u_i \in \mathcal{U}$, we have $\Psi(lub(\mathcal{S})) \leq lub(\Psi(\mathcal{S}))$. □

Corollary 2.6. *By Kleene's fixpoint theorem and since Ψ is continuous, the least fixpoint σ of Ψ can be computed as follows: Let ϕ_0 be the interpretation with $\phi_0(u_i) = \emptyset$ for $i = 1, \ldots, K$. With $\Psi^j(\phi_0)$ for $j > 0$, the j-fold application of Ψ, the equation $\sigma(u_i) = \bigcup_j \Psi^j(u_i)$ holds for every $i = 1, \ldots, K$.*

This representation of the least fixpoint allows induction proofs. For the least fixpoint σ of equations we have $\sigma(u) \subseteq \mathcal{T}(\Sigma)$ for all $u \in \mathcal{U}$. So in this case only finite trees are required.

Remark 2.7. By the dual of Kleene's fixpoint theorem for the greatest fixpoint, and since lower continuity for Ψ for all countably infinite descending chains $\xi_1 \geq \xi_2 \geq \xi_3, \ldots$ in $\mathcal{SC}_\mathcal{T}$ can be proved[1], the greatest fixpoint γ of Ψ can be computed as follows. Let ψ_0 be the 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 $i = 1, \ldots, K$.

The representation $\bigcap_j(\psi_j)(u)$ of $\gamma(u)$ allows co-induction proofs in the style of induction proofs. However, note that the greatest fixpoint γ of equations in general contains also infinite trees.

[1] we omit the straightforward proof, since it is not used in the paper

2.3. Properties of Least and Greatest Fixpoints for bot-closed SCIPs

In this subsection we show for bot-closed set constants definitions that there is a tight connection between least and greatest fixpoint:

The set $\sigma_{\mathcal{T}}(u)$ is exactly the subset of all finite cuts of trees in $\gamma_{\mathcal{T}}(u)$, which has as corollary that the inclusion problem for the greatest fixpoint can be translated into an inclusion problem for the least fixpoint. The latter problem is independent of the particular choice of $\mathcal{T}$.

Lemma 2.8. *Let $(\Sigma,\mathcal{U},\mathcal{T},\mathcal{EQ})$ be a bot-closed set constant definition. Then for all $u \in \mathcal{U}$ the following holds:*

(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 inductively constructed using the equations Eq_i (see Corollary 2.6). Hence for every 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) The direction "$\subseteq$" is trivial, since it follows from (1)
We show $\supseteq$ 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 2.9. *Let $(\Sigma,\mathcal{U},\mathcal{T},\mathcal{EQ})$ be a bot-closed set constants definition. Then for all $u \in \mathcal{U}$: All finite cuts of trees in $\sigma(u)$ are also in $\sigma(u)$.*

Lemma 2.10. *Let $(\Sigma,\mathcal{U},\mathcal{T},\mathcal{EQ})$ be a bot-closed set constant definition and $u \in \mathcal{U}$. Then $\gamma(u) = \{t \in \mathcal{T} \mid$ all finite cuts of t are in $\sigma(u)\}$.*

Proof. From Lemma 2.8 we obtain $\gamma(u) \subseteq \{t \in \mathcal{T} \mid \text{all finite cuts of } t \text{ are in } \sigma(u)\}$.

Let ξ be the interpretation that is defined for all $u \in \mathcal{U}$ by
$\xi(u) := \{t \in \mathcal{T} \mid \text{all finite cuts of } t \text{ are in } \sigma(u)\}$. 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 $\Psi(\xi)(u) = \{\bot\} \cup \xi(r_1) \cup \ldots \cup \xi(r_m)$. The goal is to show that $\Psi(\xi)(u) = \xi(u)$:

(1) $\Psi(\xi)(u) \subseteq \xi(u)$: 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. $t \in \xi(r_1)$ 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 $t \in \xi(u)$.
(2) $\xi(u) \subseteq \Psi(\xi)(u)$: 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 $\sigma(u_j) \subseteq \xi(u_j)$ for all $u_j \in \mathcal{U}$, 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 in $\sigma(c\ u_1 \ldots u_n)$ by Corollary 2.9, 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 subtree-closed, $t_i \in \mathcal{T}$. This implies $t_i \in \xi(u_i)$. Hence $t = (c\ t_1 \ldots t_n) \in \xi(c\ u_1 \ldots u_n)$, and thus $t \in \Psi(\xi)(u)$.

Summarizing, ξ is a fixpoint of Ψ, and hence $\xi(u) \subseteq \gamma(u)$ for all $u \in \mathcal{U}$. □

Note that Lemma 2.8 and Corollary 2.9 do not require admissibility of $\mathcal{T}$, whereas this is required in the proof of Lemma 2.10.

Theorem 2.11. *Let $(\Sigma, \mathcal{U}, \mathcal{T}, \mathcal{EQ})$ be a bot-closed set constant definition. Then for all $u_1, u_2 \in \mathcal{U}$: $\gamma(u_1) \subseteq \gamma(u_2)$ 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 2.10, since the construction $\{t \mid \text{ all finite cuts of } t \text{ are in } \sigma(u)\}$ 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 2.8 shows that $\sigma(u_2)$ is the subset of finite cuts of trees in $\gamma(u_2)$, hence $\sigma(u_1) \subseteq \sigma(u_2)$. □

We have the following immediate consequences:

Corollary 2.12. *Given two admissible sets $\mathcal{T}_1$, $\mathcal{T}_2$. Then the inclusion problem for a bot-closed set constants definition $(\Sigma, \mathcal{U}, \mathcal{T}_1, \mathcal{EQ})$ is equivalent to the inclusion problem for the set constants definition $(\Sigma, \mathcal{U}, \mathcal{T}_2, \mathcal{EQ})$. In particular the following hold:*

- *Assume given a bot-closed set constants definition. Then the inclusion problem for set constants is equivalent to the inclusion problem for set constants w.r.t. the least fixed-point semantics.*
- *Assume given a bot-closed set constants definition. Then the inclusion problem for set constants is equivalent to the inclusion problem for set constants w.r.t. the greatest fixed-point semantics where $\mathcal{T} = \mathcal{T}_\infty(\Sigma)$.*

2.4. Reducing bot-free SCIPs to bot-closed SCIPs

In this subsection we show the relationship between bot-closed and bot-free set constants definitions.

To ease reading, we assume in this subsection that we have a bot-free SCIP and consider also the corresponding bot-closed SCIP with equations $Eq_{\perp,i} \in \mathcal{EQ}_\perp$ that are determined from $Eq_i \in \mathcal{EQ}$ by adding a $\perp$-component. We use Ψ, γ as notations for the operator and the greatest fixpoint, respectively, and $\Psi_\perp, \gamma_\perp$ for the notions for the corresponding bot-closed SCIP.

Lemma 2.13. *Let* $(\Sigma,\mathcal{U},\mathcal{T},\mathcal{EQ})$ *be a bot-free set constants definition,* $(\Sigma,\mathcal{U},\mathcal{T},\mathcal{EQ}_\bot)$ *be the corresponding bot-closed set constants definition, and let* $u \in \mathcal{U}$*. Then* $\gamma(u) = \{t \mid t \in \gamma_\bot(u)$ *and* t *has no occurrence of* $\bot\}$

Proof. Let γ' be the interpretation defined as

$$\gamma'(u) := \{t \mid t \in \gamma_\bot(u) \text{ and } t \text{ has no occurrence of } \bot\}.$$

Then it is easy to see that γ' is a fixpoint of Ψ. This follows from the fixpoint-property of $\gamma_\bot$ and the fact that Ψ does not introduce $\bot$'s in the interpretations. We have shown that $\gamma' \leq \gamma$.

For all set constants u_i we obviously have $\gamma(u_i) \subseteq (\Psi_\bot \circ \gamma)(u_i)$, since $\gamma(u_i) = (\Psi \circ \gamma)(u_i) \subseteq (\Psi_\bot \circ \gamma)(u_i)$. Hence, by the Knaster-Tarski fixpoint theorem, we have $\gamma(u) \subseteq \gamma_\bot(u)$ for all u, since $\gamma_\bot(u)$ is a greatest fixpoint of $\Psi_\bot$. This implies $\gamma(u) = \gamma'(u)$ for all set constants u. □

Lemma 2.14. *Let* $(\Sigma,\mathcal{U},\mathcal{T},\mathcal{EQ})$ *be a bot-free set constants definition,* $(\Sigma,\mathcal{U},\mathcal{T},\mathcal{EQ}_\bot)$ *be the corresponding bot-closed set constants definition, and let* $u \in \mathcal{U}$*. Then* $\gamma_\bot(u) = \{t' \in \mathcal{T} \mid t \in \gamma(u)$ *and* t' *is cut of* $t\}$*.*

Proof. Let γ' be the interpretation defined as $\gamma'(u_i) := \{t' \in \mathcal{T} \mid t \in \gamma(u_i)$ and t' is cut of $t\}$ for all u_i. Then γ' is a fixpoint of $\Psi_\bot$, which follows from the fixpoint-property of γ. Hence $\gamma' \leq \gamma_\bot$.

Since γ' is the maximal extension of γ by adding cuts, and using Lemma 2.13, we obtain that $\gamma' = \gamma_\bot$. □

Theorem 2.15. *Let* $(\Sigma,\mathcal{U},\mathcal{T},\mathcal{EQ})$ *be a bot-free set constants definition and* $u_1, u_2 \in \mathcal{U}$*. Then* $\gamma(u_1) \subseteq \gamma(u_2)$ *iff* $\gamma_\bot(u_1) \subseteq \gamma_\bot(u_2)$*.*

Proof. This follows from Lemmas 2.13 and 2.14 . □

This theorem allows to test for inclusion in bot-free SCIP using the greatest fixpoint by reducing this problem to the corresponding bot-closed SCIP.

Example 2.16. Theorem 2.15 cannot be extended to SCIPs that are neither bot-closed nor bot-free. Consider the set constants definition $(\{\bot, \texttt{cons}, c\}, \mathcal{T}_\infty(\Sigma), \{u, \texttt{Inf}, \texttt{Inf}'\}, \{u = c, \texttt{Inf} = \bot \cup (\texttt{cons}\ u\ \texttt{Inf}), \texttt{Inf}' = (\texttt{cons}\ u\ \texttt{Inf}')\})$. Then $\gamma_\bot(\texttt{Inf}) = \gamma_\bot(\texttt{Inf}')$, since the definitions of `Inf` and `Inf′` are identical after adding $\bot$-components, but $\gamma(\texttt{Inf}) \neq \gamma(\texttt{Inf}')$.

Example 2.17. We show that in Theorem 2.15 the greatest fixpoint cannot be replaced by the least fixpoint: Let $\Sigma := \{\bot, c_1, c_2\}$. We consider the set constants definition $(\Sigma, \mathcal{T}_\infty(\Sigma), \{u_1, u_2\}, \{u_1 = \bot \cup (c_1\ u_2), u_2 = \bot \cup (c_2\ u_1)\})$. Then $\gamma(u_1)$ consists of the infinite tree $c_1\ (c_2\ (c_1\ (\ldots)))$ and all its finite cuts, and $\gamma(u_2)$ consists of the infinite tree $c_2\ (c_1\ (c_2\ (\ldots)))$ and all its finite cuts, which makes u_1, u_2 different w.r.t. the greatest fixpoint semantics. The according bot-free set constants definition is $(\{\bot, c_1, c_2\}, \mathcal{T}_\infty(\Sigma), \{u_1, u_2\}, \{u_1 = (c_1\ u_2), u_2 = (c_2\ u_1)\})$. Using the least fixpoint semantics u_1, u_2 are the empty set, and thus equal.

2.5. Decidability and Complexity

In this subsection we show DEXPTIME-completeness as complexity of the (bot-closed as well as bot-free) set constant inclusion problem by using results from tree automata.

Our set constant inclusion problem can be solved as an inclusion problem of the accepted languages of non-deterministic bottom-up tree automata [11].

A non-deterministic bottom-up tree automata A consists of finitely many states, some are accepting ones, and rules of the form $c(q_1, \ldots, q_n) \to q$, where $c \in \Sigma$, and q, q_i are states. A finite tree t is accepted by A, if it is possible, in a bottom-up fashion, to reach $q(t)$ for an accepting state q, where a single application of a rule is like $c(q_1(t_1), \ldots, q_n(t_n)) \to q(c(t_1, \ldots, t_n))$, where $c(q_1, \ldots, q_n) \to q$ is a rule in A, and $c(t_1, \ldots, t_n)$ is a subterm of t. The accepted language of A is the set of all trees t, such that $q(t)$ can be reached for some accepting state q of A.

By a straightforward encoding where set-constants are the states of the automata and the definitions of set constants are translated into the rules of the automata, and using Theorem 2.11 we obtain:

Proposition 2.18. *Every bot-closed SCIP can be solved by encoding it in linear time as an inclusion problem of the languages accepted by non-deterministic tree automata.*

Proposition 2.19. *The bot-closed as well as bot-free SCIP is in DEXPTIME.*

Proof. This follows from Proposition 2.18, since the inclusion problem of the accepted languages of two non-deterministic tree automata is in DEXPTIME [11]. □

Remark 2.20. If in the inclusion test $u_1 \subseteq u_2$, the set constant u_2 corresponds to a top-down deterministic tree automaton, then the language inclusion test can be done using a fixpoint iteration in polynomial time. This case may also occur in practice and potentially alleviates the high worst case complexity of the SCIP.

A lower bound on the complexity can be obtained using the problem whether a bottom-up nondeterministic tree automata accepts the full language of closed terms. Note that an encoding of the inclusion problem for tree automata into a set constant inclusion problem is not easily possible, since we require $\bot$ as part of every defining equation in bot-closed SCIP.

Theorem 2.21. *If the signature contains at least one function symbol c with* $\mathrm{ar}(c) \geq 2$*, then the bot-closed as well as bot-free set-constant inclusion problem is DEXPTIME-complete.*

Proof (Sketch). Since $\top$ can be encoded, the SCIP "$\top \subseteq u$" is equivalent to the problem whether an appropriately encoded tree automaton, where u corresponds to the accepting state, accepts all terms. DEXPTIME-hardness follows since the problem of acceptance of all ground terms by a non-deterministic tree automaton is DEXPTIME-hard [11], provided the signature contains at least one function

symbol of arity at least 2. DEXPTIME-completeness now follows from Proposition 2.19. □

The theorem above on the relation between bot-free and bot-closed SCIPs is also correct under the slight generalization that bot-free set constants definitions may have an empty right hand side. However, we do not provide an explicit proof since it only obscures the arguments by several extra case distinctions in the proofs.

Remark 2.22. Theorem 2.21 restricted to the set of all infinite trees $\mathcal{T}_\infty(\Sigma)$ and for bot-free SCIP shows that the inclusion of infinite tree languages that are accepted by Büchi-automata for infinite trees [37], in which every state is accepting, is also in DEXPTIME. This holds perhaps after some recoding of signature constants as infinite trees. We do not expand on this, since it is beyond the scope of this paper to explore the exact relationship to the inclusion problem of infinite tree languages and of languages accepted by automata for infinite trees [37].

Remark 2.23. The bot-closed set constant inclusion problem can also be solved by encoding it as satisfiability of a set-constraint consisting of the straightforward encoding of the equations together with the constraint $u_1 \subseteq u_2$. The encoding of the equations ensures that every solution must be the least fixpoint, and the constraint $u_1 \subseteq u_2$ then can only contribute a yes/no decision. The derived complexity is NEXPTIME [2, 3].

Note that the "natural" encoding of bot-closed SCIP into co-definite set-constraints [9] does not work, since our two-stage solution process is not directly encodable.

Remark 2.24. It is no problem to also allow definitional equations of the form $u_1 = u_2$ in bot-closed SCIP, since they can be treated as the so-called ε-transitions for tree automata. Their removal is a polynomial action. Note, however, that there is a difference during removal of set-constants whether the greatest fixpoint or the least fixpoint is considered. Constants in cyclic definitions are to be set to $\top$. If the least fixpoint is considered, the constants in cyclic definitions are to be set to $\bot$ instead.

Remark 2.25. An alternative way to obtain the equivalence of least fixpoint and greatest fixpoint would be to use notions from cpos and continuity: It may roughly work as follows: The set of terms $\mathcal{T}_\infty(\Sigma)$ can be made a cpo by using an ordering inherited from $\bot \leq t$ for all t, and $s_i \leq t_i$ for $i = 1, \ldots, n \Rightarrow c\ s_1\ \ldots\ s_n \leq c\ t_1\ \ldots\ t_n$. Then the continuity of all constructors can be shown. This would allow to switch between least fixpoint and greatest fixpoint semantics.

3. Application to an Extended Lambda-Calculus with Case and Constructors

In this section we illustrate and explain strictness analysis as a static analysis method and show the use of the set constant inclusion problem and the consequences of the solution algorithms and the complexity results.

First we will give a sketch of an extended untyped lambda calculus and how the set constant inclusion problem arises. In the second subsection, instead of expressions, a set of terms is given together with a relation to the expressions, hence we will be more rigorous.

3.1. Example of an Untyped Call-By-Need Lambda-Calculus

There are several extended call-by-name and call-by-need lambda calculi with case-expressions and constructors with a thorough investigation of their operational semantics (see e.g. [4, 5, 10, 21, 24, 25, 34, 35]).

We explicitly define the syntax of a deterministic, untyped call-by-need lambda-calculus with case and constructors.
There is a finite set of constructors as before, and an infinite set $\mathcal{V}$ of variables. The syntax of expressions E and patterns P, where V, V_i are non-terminals for variables, is as follows:

$$\begin{array}{lcl} E & ::= & V \mid \lambda V.E \mid (E\ E) \mid (c\ E_1 \ldots\ E_{\mathrm{ar}(c)}) \mid (\texttt{case}\ E\ (P_1\texttt{->}E_1) \ldots (P_n\texttt{->}E_n)) \\ & & \mid (\texttt{letrec}\ \{V_1 = E_1; \ldots; V_n = E_n\}\ \texttt{in}\ E) \\ P & ::= & (c\ V_1 \ldots V_{\mathrm{ar}(c)}) \end{array}$$

There is the usual notion of free and bound variables in expressions, where the binders are λ, `letrec` and pattern in case-expressions. An expression is closed if it has not any free variables. The set of all expressions is denoted as Λ, and the set of all closed expressions as Λ_0. The reduction rules are variants of beta- and case-reductions and several reduction rules for `letrec`-expressions.

There is a notion of evaluation, which is a sequence of particularly chosen reductions (so-called normal order reductions) to a weak head normal form (WHNF), which is defined as follows.

Definition 3.1. A *constructor expression* is an expression $(c\ t_1 \ldots t_{\mathrm{ar}(c)})$. A *constructor weak head normal norm* (*CWHNF*) is a constructor expression, or a `letrec`-expression (`letrec` *Env* `in` $(c\ t_1 \ldots t_{\mathrm{ar}(c)})$), or a `letrec`-expression (`letrec` $x = (c\ t_1 \ldots t_{\mathrm{ar}(c)}), Env$ `in` x). A *functional weak head normal form* (*FWHNF*) is either an abstraction $\lambda x.t$ or an expression of the form (`letrec` *Env* `in` $\lambda x.t$). An expression is a *weak head normal form* (*WHNF*) if it is a CWHNF or an FWHNF. If an expression t evaluates to a WHNF, then this is denoted as $t{\Downarrow}$.

Equality of expressions is defined using the contextual preorder:

$$s \leq_c t \text{ iff for all contexts } C : C[s]{\Downarrow} \Rightarrow C[t]{\Downarrow}$$
$$\text{and}$$
$$s \sim_c t \text{ iff } s \leq_c t \wedge t \leq_c s.$$

Here a context C means an expression with a single hole, where an expression can be plugged in.

We define $\Omega := (\lambda x.(x\ x))\ (\lambda y.(y\ y))$, which will later be represented by $\bot$. Note that based on the definition of normal order reduction the cyclic expression ($\texttt{letrec}\ x = x\ \texttt{in}\ x$) is $\sim_c$-equivalent to Ω, and that also other expressions like $((c\ t_1 \ldots t_{\mathrm{ar}(c)})\ t')$ and $(\texttt{case}\ (\lambda x.t)\ \ldots)$ are $\sim_c$-equivalent to Ω.
The following classification of expressions holds [34]:

Proposition 3.2. *For every closed expression $t \in \Lambda_0$ one of the following holds:*

(1) $t \sim_c \Omega$
(2) $t \sim_c t'$ *where t' is a closed constructor expression.*
(3) $t \sim_c t'$*, where t' is a closed FWHNF.*

Furthermore, the constructors are "free": I.e. w.r.t. equivalence $\sim_c$, the following inequalities hold: $(c\ t_1 \ldots t_n) \not\sim_c \Omega \not\sim_c \lambda x.t \not\sim_c (c\ t_1 \ldots t_n)$*, and* $(c_1\ s_1 \ldots s_n) \sim_c (c_2\ t_1 \ldots t_m) \Leftrightarrow c_1 = c_2, n = m$ *and* $\forall i = 1, \ldots, n : s_i \sim_c t_i$.

A consequence is that we can reason over sets of finite and infinite terms instead of closed expressions, where terms are built from constructors, $\bot$, and abstractions, where an outer let-environment can be ignored due to the classification property.

In order to apply the results from subsection 2.5, there is one last simplifying step: we have to put all closed FWHNFs into one set, represented by the new 0-ary constant *Fun*. In addition, the expression Ω is represented by the symbol $\bot$, which is a 0-ary constant, but not a constructor in the language Λ_0.

3.2. Consequences for Extended Lambda Calculi

The final set of terms $\mathcal{T}_{comp}$ is the set of all closed finite and infinite computable terms according to the syntax $E ::= \bot \mid (c\ E_1\ \ldots\ E_{\mathrm{ar}(c)}) \mid \mathit{Fun}$. As mentioned above, the set $\mathcal{T}_{comp}$ can also be obtained as a quotient of Λ_0 by $\sim_c$, and then by putting all closed FWHNFs into one set, represented by the constant *Fun*. Hence every expression in Λ_0 is represented.

The set $\mathcal{T}_{comp}$ contains all finite trees and also the computable infinite trees over the constructors, *Fun* and $\bot$, and it is admissible. This follows easily from the properties of expressions in Λ_0, and since Λ is a programming language. Note that the set $\mathcal{T}_{comp}$ does not contain all infinite trees. An example for an expression equivalent to an infinite tree in $\mathcal{T}_{comp}$ is ($\texttt{letrec}\ x = \texttt{cons}\ 1\ x\ \texttt{in}\ x$), for the binary list-constructor `cons`, which corresponds to an infinite list of 1s as entries.

Now the set constants can be defined as in subsection 2.2, however, as bot-closed. Using the greatest fixpoint to compute the contained equivalence classes is appropriate, since a non-strict semantics is used. Note that since the constructors are non-strict also infinite trees are included, where looping expressions are represented by $\bot$. Theorem 2.21 justifies the use of algorithms from tree automata to solve the set constant inclusion problem. Since in general the binary list-constructor is available, the set constant inclusion problem is DEXPTIME-complete. In lambda-calculi where the classification property (see Proposition 3.2) holds, there is a potential to apply more static analyses, since every expression t is equivalent to some term in $\mathcal{T}_{comp}$.

For example, the following set constants that are analogous to constants in [25, 38] can be defined as:

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

An example for a nontrivial inclusion is $\texttt{BotElem} \subseteq \texttt{List}$, which can be proved using the methods from tree automata as well as by directly using co-induction. Note that this is not an example for a hard inclusion test, since it falls into the category that can be solved in polynomial time, see Remark 2.20.

Example 3.3. A slightly more complex example is a tree data structure, given in the typed Haskell notation by

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

where `Tree a` is a type with a type variable `a`, and `Leaf` and `Node` are constructors. We assume that only Booleans and the trees are available. Assuming typed trees of Haskell-type `(Tree Boolean)`, some set constants are:

$$\begin{array}{lcl}
\mathrm{T} & = & \bot \cup (\texttt{Leaf}\ \mathrm{B}) \cup (\texttt{Node}\ \mathrm{T}\ \mathrm{T}) \\
\mathrm{B} & = & \bot \cup \texttt{True} \cup \texttt{False} \\
\mathrm{BF} & = & \bot \cup \texttt{False} \\
\mathrm{BT} & = & \bot \cup \texttt{True} \\
\mathrm{TF} & = & \bot \cup (\texttt{Leaf}\ \mathrm{BF}) \cup (\texttt{Node}\ \mathrm{TF}\ \mathrm{TF}) \\
\mathrm{TT} & = & \bot \cup (\texttt{Leaf}\ \mathrm{BT}) \cup (\texttt{Node}\ \mathrm{TT}\ \mathrm{T}) \cup (\texttt{Node}\ \mathrm{TF}\ \mathrm{TT}) \\
\mathrm{TO} & = & \bot \cup (\texttt{Leaf}\ \mathrm{B}) \cup (\texttt{Node}\ \mathrm{TO}\ \mathrm{TO}) \\
\mathrm{TW} & = & \bot \cup (\texttt{Leaf}\ \bot) \cup (\texttt{Node}\ \mathrm{TW}\ \mathrm{T}) \cup (\texttt{Node}\ \mathrm{TO}\ \mathrm{TW})
\end{array}$$

TT corresponds to a function that scans the tree from left to right and loops, if a leaf with value `True` is encountered, and TW corresponds to a function that completely scans the tree from left to right. The test could be whether an inclusion $\mathrm{TT} \subseteq \mathrm{TW}$ holds, which is not covered by the polynomial subcases.

The strictness analyses [25, 32, 34, 35] employ abstract reduction. This is a method to extend expressions by set constants and to evaluate them in all possible ways, where also loop detection rules are applied. This kind of strictness analysis has a non-termination analysis as its core, and the used set constants are defined by recursive definitions, exactly as bot-closed set constants definition in subsection 2.2. E.g. to check whether a defined binary function f (i.e. an expression) is strict in its second argument, one can show that there are no successful evaluations of a term matching $(f\ \top\ \bot)$. Also other questions could be answered by this kind of analysis: If $(g\ \texttt{Inf})$ results in nontermination, then during evaluation of an expressions $(g\ s)$, g can safely evaluate the spine of the input-list before evaluating the function call, i.e. first compute the length without evaluating the elements.

If (g `BotElem`) results in nontermination, then during evaluation of (g s), g can safely evaluate the spine of the input-list, and also the elements before evaluating the expression (g s).

There are at least two situations during the run of a strictness analyzer using abstract reduction, where an algorithm to solve the set constant inclusion problem is useful:

(1) If during abstract reduction, an expression of the form $C[u_1, \ldots, u_n]$ has a successor in the (non-deterministic) abstract reduction of the form $C[u'_1, \ldots, u'_n]$, where u_i, u'_i are set constants and $u_i \subseteq u'_i, i = 1, \ldots, n$ can be proved, then the branch where $C[u'_1, \ldots, u'_n]$ occurs, can be stopped by loop detection.
This loop detection is the main use of the inclusion problem.

(2) If for a defined function f, it is already known that $f(u_1)$ does not terminate, then also $f(u_2)$ does not terminate, provided $u_2 \subseteq u_1$. There is a trade-off between again checking non-termination (which is undecidable), or using the algorithm for the set inclusion problem, which is DEXPTIME and thus has a high worst case complexity.

Note that the loop detection accounts for much of the strength of the strictness analyzers using abstract reduction, and that the set constants inclusion check contributes to increasing their power. The strictness problem as such is undecidable, and the hope is that the increase in power using the algorithms from tree automata outweighs the added exponential worst case complexity. From a practical point of view, this is no real problem, since a strictness analyzer performs under resource restrictions.

4. 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 set constants inclusion problem is shown to be DEXPTIME-complete. Practical examples have to be analyzed to check whether the worst-case running time shows up in practice.

Future work may investigate the set constants inclusion problem for more expressive languages, e.g. also for non-bot-closed set constants definitions, or if intersections and/or complements are permitted, or for the full demand language in [35]. The latter is a notable exception to the down-closed condition, and has to define semantics by using an alternation between fixpoint computation and continuity and closure properties w.r.t. an approximation ordering.

Another line of research would be to investigate the connection of Theorem 2.15 with the inclusion problem for infinite tree languages and for languages accepted by automata for infinite trees [37].

We thank the referees and Joachim Niehren for many valuable remarks and comments, which helped to avoid errors and to greatly improve the paper.

References

[1] S. Abramsky and C. Hankin. *Abstract interpretation of declarative languages*. Ellis Horwood, 1987.

[2] 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.

[3] 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.

[4] Zena M. Ariola and Stefan Blom. Cyclic lambda calculi. In *TACS*, pages 77–106, 1997. Sendai, Japan.

[5] Zena M. Ariola and Jan Willem Klop. Lambda calculus with explicit recursion. *Information and Computation*, 139(2):154–233, 1997.

[6] 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.

[7] G. L. Burn, C. L. Hankin, and S. Abramsky. The theory for strictness analysis for higher order functions. In H. Ganzinger and N. D. Jones, editors, *Programs as Data Structures*, number 217 in Lecture Notes in Computer Science, pages 42–62. Springer, 1985.

[8] Geoffrey Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation*. Pitman, London, 1991.

[9] 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.

[10] David Clark, Chris Hankin, and Sebastian Hunt. Safety of strictness analysis via term graph rewriting. In *SAS 2000*, pages 95–114, 2000.

[11] 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.

[12] M. Coppo, F. Damiani, and P. Giannini. Strictness, totality, and non-standard type inference. *Theoretical Computer Science*, 272(1-2):69–112, 2002.

[13] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 252–252. ACM Press, 1977.

[14] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, 1992.

[15] Philippe Devienne, Jean-Marc Talbot, and Sophie Tison. Co-definite set constraints with membership expressions. In *JICSLP'98: Proceedings of the 1998 joint international conference and symposium on Logic programming*, pages 25–39, Cambridge, MA, USA, 1998. MIT Press.

[16] Kirsten Lackner Solberg Gasser, Hanne Riis Nielson, and Flemming Nielson. Strictness and totality analysis. *Sci. Comput. Program.*, 31(1):113–145, 1998.

[17] Thomas P. Jensen. Inference of polymorphic and conditional strictness properties. In *Symposium on Principles of Programming Languages*, pages 209–221, San Diego, January 1998. ACM Press.

[18] Tsun-Ming Kuo and Prateek Mishra. Strictness analysis: A new perspective based on type inference. In *Functional Programming Languages and Computer Architecture*, pages 260–272. ACM Press, 1989.

[19] John Launchbury and Simon Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(2):293–341, 1995.

[20] Laurent Mauborgne. Improving the representation of infinite trees to deal with sets of trees. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, volume 1782 of *LNCS*, pages 275–289. Springer, 2000.

[21] Andrew K.D. Moran, David Sands, and Magnus Carlsson. Erratic fudgets: A semantic theory for an embedded coordination language. *Sci. Comput. Program.*, 46(1-2):99–135, 2003.

[22] Alan Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs*. PhD thesis, University of Edinburgh, 1981.

[23] Eric Nöcker. Strictness analysis using abstract reduction. Technical Report 90-14, Department of Computer Science, University of Nijmegen, 1990.

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

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

[26] Dirk Pape. Higher order demand propagation. In K. Hammond, A.J.T. Davie, and C. Clack, editors, *Implementation of Functional Languages (IFL '98) London*, volume 1595 of *Lecture Notes in Computer Science*, pages 155–170. Springer-Verlag, 1998.

[27] Dirk Pape. *Striktheitsanalysen funktionaler Sprachen*. PhD thesis, Fachbereich Mathematik und Informatik, Freie Universität Berlin, 2000. in German.

[28] Ross Paterson. Compiling laziness using projections. In *Static Analysis Symposium*, volume 1145 of *LNCS*, pages 255–269, Aachen, Germany, September 1996. Springer.

[29] Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003. `www.haskell.org`.

[30] R. Plasmeijer and M. van Eekelen. The concurrent Clean language report: Version 1.3 and 2.0. Technical report, Dept. of Computer Science, University of Nijmegen, 2003. `http://www.cs.kun.nl/~clean/`.

[31] Pawel Rychlikowski and Tomasz Truderung. Set constraints on regular terms. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic, 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 20-24, 2004, Proceedings*, volume 3210 of *LNCS*, pages 458–472. Springer, 2004.

[32] 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.

[33] 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.

[34] 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.

[35] 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`.

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

[37] Wolfgang Thomas. Automata on infinite objects. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Formal Models and Semantics (B)*, volume B, pages 133–192. Elsevier, 1990.

[38] 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.

[39] Philip Wadler and John Hughes. Projections for strictness analysis. In *Functional Programming Languages and Computer Architecture*, number 274 in Lecture Notes in Computer Science, pages 385–407. Springer, 1987.

Communicated by (The editor will be set by the publisher).
September 18, 2006.