Formal Language, Grammar and Set-Constraint-Based Program Analysis by Abstract Interpretation

Patrick COUSOT

LIENS, École Normale Supérieure
75230 Paris cedex 05, France
cousot@dmi.ens.fr

Radhia COUSOT

LIX, CNRS & École Polytechnique
91140 Palaiseau cedex, France
rcousot@lix.polytechnique.fr

Abstract

Grammar-based program analysis à la Jones and Muchnick and set-constraint-based program analysis à la Aiken and Heintze are static analysis techniques that have traditionally been seen as quite different from abstract-interpretation-based analyses, in particular because of their apparent non-iterative nature. For example, on page 18 of [17], it is alleged that "The finitary nature of abstract interpretation implies that there is a fundamental limitation on the accuracy of this approach to program analysis. There are decidable kinds of analysis that *cannot* be computed using abstract interpretation (even with widening and narrowing). The set-based analysis considered in this thesis is one example".

On the contrary, we show that grammar and set-constraint-based program analyses are similar abstract interpretations with iterative fixpoint computation using either a widening or a finitary grammar/set-constraints transformer or even a finite domain for each particular program.

The understanding of grammar-based and set-constraint-based program analysis as a particular instance of abstract interpretation of a semantics has several advantages. First, the approximation process is formalized and not only explained using examples. Second, a domain of abstract properties is exhibited which is of general scope. Third, these analyses can be easily combined with other abstract-interpretation-based analyses, in particular for the analysis of numerical values. Fourth, they can be generalized to very powerful attribute-dependent and context-dependent analyses. Finally, a few misunderstandings may be removed.

Conference Record of FPCA '95 SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture, La Jolla, California, 25-28 June 1995, ACM Press, New York, U.S.A., pp. 170–181.

1. Introduction

We construct program analyses by abstract interpretation of a collecting semantics where abstract properties are coded using tuples of formal languages. A further approximation is to approximate such formal languages by different kinds of formal systems such as grammars or systems of set-constraints. The idea of using *regular tree grammars* for program analysis, is due to Jones and Muchnick [22, 23, 24] following Reynolds [29]. It was further developed by Aiken, Mishra, Murphy, Reddy and Sørensen [2, 27, 30], implemented by Aiken and Murphy [1] and reformulated by Heintze and Jaffar [16, 17, 19, 20] as *set-constraint-based program analysis*. We show that in such formal-language-based program analyses, the abstract semantics specifying the strongest abstract program property is defined as the least fixpoint of an abstract property transformer. This abstract semantics can be also presented as a system of constraints. For program analysis by resolution of such systems of set-constraints, no iteration is apparent. Therefore we are interested in understanding whether or not this least fixpoint is computed iteratively or approximated by non-iterative methods (e.g. by elimination). We show that actually this least fixpoint is computed by a chaotic iteration. Convergence can be enforced using either a widening, or a finitary abstract property transformer or else by choosing for each particular program an abstract domain which satisfies the ascending chain condition (and even is finite). This new point of view on grammar and set-constraint-based analysis allows us to easily combine these dependence-free analyses with existing non-grammar or non-set-constraint-based abstract interpretations and to generalize them to obtain very powerful infinitary context-dependent program analyses.

2. Standard Semantics

A *standard semantics* associates a behavior $S_{sd}[\![P]\!] \in B_{sd}$ to each program P of the language. This behavior is a specification of the possible program executions. Elements of the *behavior domain* B_{sd} can be higher-dimensional automata, execution traces, judgments, functions, sets of states, etc.

Example The standard semantics $\mathtt{f}_{sd}$ of the following functional program `f`:

```
f(N) =
  if (N <= 0) then
    cons(0, cons(0, cons(0, nil)))
  else let X = f(N-1) in                         (1)
    cons(a(hd(X)), cons(b(hd(tl(X))),
      cons(c(hd(tl(tl(X)))), nil)));
```

(where $\mathtt{hd(cons(}H\mathtt{,}\ T\mathtt{))} \overset{\text{def}}{=} H$ and $\mathtt{tl(cons(}H\mathtt{,}\ T\mathtt{))} \overset{\text{def}}{=} T$) can be chosen as the function computed by this program, that is simply:

$$\begin{aligned}\mathtt{f}_{sd} \;=\; & \lambda n\boldsymbol{\cdot}\,\text{if } n \le 0 \text{ then}\\ & \qquad \mathtt{cons}(0, \mathtt{cons}(0, \mathtt{cons}(0, \mathtt{nil})))\\ & \quad \text{else}\\ & \qquad \mathtt{cons}(\mathtt{a}^n(0), \mathtt{cons}(\mathtt{b}^n(0), \mathtt{cons}(\mathtt{c}^n(0), \mathtt{nil})))\end{aligned}$$

where we use the abbreviation:

$$\mathtt{a}^i(0) \;=\; \underbrace{\mathtt{a}(\cdots\mathtt{a}(}_{i \text{ times}}0)\cdots)$$

In this example, we have $B_{sd} = \mathbb{Z} \mapsto (\mathbb{L} \cup \{\bot\})$ where $\mathbb{L}$ is the set of lists and $\bot$ denotes non-termination. □

We assume that B_{sd} involves symbolic semantic values (in L) associated to indexes (in I). These symbolic semantic values are assumed to be described by sentences of a *formal language* L such that:

— the ranked *alphabet* A contains function symbols $f^n \in \Sigma$ of fixed arity $n \ge 0$;

— the language L is the set of ground terms t on A (written in prefix parenthesized notation):

$$t \;::=\; f^0 \mid f^n(t_1, \ldots, t_n);$$

(L can be understood isomorphically as the algebra T_Σ of trees on the signature Σ).

Example For the program (1), the ranked alphabet is $A = \{\mathtt{0}^0, \mathtt{nil}^0, \mathtt{a}^1, \mathtt{b}^1, \mathtt{c}^1, \mathtt{cons}^2\}$. The language L is the set of ground terms t defined by the grammar:

$$t \;::=\; \mathtt{0} \mid \mathtt{a}(t) \mid \mathtt{b}(t) \mid \mathtt{c}(t) \mid \mathtt{nil} \mid \mathtt{cons}(t_1, t_2)$$

Integers are simply (arbitrarily) ignored. □

The set I of *indexes* is assumed to be finite. The precision of the analysis depends upon the choice of this set I of indexes which can be program points, variables, expression labels, store locations, auxiliary variables (e.g. to denote intersections [27]), etc.

Example For the program (1), the set of indexes can be chosen to be $I = \{\mathcal{X}\}$. This (arbitrary) choice is made to compute a single global program invariant (associated to the index $\mathcal{X}$[1]). A more refined choice would consist of program points so as to compute local invariants (associated to these program points). □

[1]The name $\mathcal{X}$ is certainly not meaningful because it is associated with `f`. $\mathcal{X}$ has been preferred to other choices (such as $\mathcal{F}$) because it will later correspond to an unknown.

3. Collecting Semantics

A *collecting semantics* associates a strongest property $S_{co}\{\!|P|\!\} \in D_{co}$ to each program P of the language: $S_{co}\{\!|P|\!\} \overset{\text{def}}{=} \{S_{sd}[\![P]\!]\}$; Here a property is understood as the set of behaviors satisfying this property. The concrete *semantic domain* $D_{co} \overset{\text{def}}{=} \wp(B_{sd})$ is a complete boolean lattice $(D_{co}, \subseteq, \emptyset, B_{sd}, \cup, \cap, \neg)$. We are interested in answering questions of the form $S_{co} \subseteq P_{co}$ where $P_{co} \in D_{co}$ is a semantic property.

We assume that the collecting semantics S_{co}[2] can be expressed as the least fixpoint $S_{co} \overset{\text{def}}{=} \operatorname{lfp} F_{co}$ of a $\emptyset$-strict complete $\cup$-morphism *concrete property transformer* $F_{co} \in D_{co} \mapsto D_{co}$[3]. Usually F_{co} is defined by induction on the syntax of the program, an aspect that we neglect here for simplicity.

D_{co} usually formalizes program properties as the set of elements having this property. Many choices are possible. A few simple examples of collecting semantics for formal-language program analysis of functional, imperative, parallel and logic programs are considered below.

Example The collecting semantics $\mathtt{f}_{co}$ for the functional program (1) can be chosen as the set of possible functions computed by this program ignoring possible non-termination, that is simply:

$$\begin{aligned}\mathtt{f}_{co} \;=\; \{ & \lambda n\boldsymbol{\cdot}\,\text{if } n \le 0 \text{ then}\\ & \qquad \mathtt{cons}(0, \mathtt{cons}(0, \mathtt{cons}(0, \mathtt{nil})))\\ & \quad \text{else}\\ & \qquad \mathtt{cons}(\mathtt{a}^n(0), \mathtt{cons}(\mathtt{b}^n(0), \mathtt{cons}(\mathtt{c}^n(0), \mathtt{nil})))\}\end{aligned}$$

In this example, we have $D_{co} = \wp(\mathbb{Z} \mapsto \mathbb{L})$. A program property is therefore chosen to be an invariant $I \in D_{co}$. The collecting semantics of a program P is the strongest property of P, that is the singleton set $\mathtt{f}_{co} = \{\mathtt{f}_{sd}\} \in D_{co}$[4]. This allows us to formalize "P has property I" as $\mathtt{f}_{co} \subseteq I$ that is equivalently $\mathtt{f}_{sd} \in I$. □

Example The collecting semantics of the following imperative program:

```
{1}  X := cons(0, cons(0, cons(0, nil)));
{2}  while N > 0 do begin
{3}    X := cons(a(hd(X)), cons(b(hd(tl(X))),
                cons(c(hd(tl(tl(X)))), nil)));  (2)
{4}    N := N - 1;
{5}  end;
{6}
```

associates to each program point p the set $\rho_{co}[p]$ of pairs $\langle n,\, x\rangle$ of possible values of variables `N` and `X`:

$$\rho_{co}[1] \;=\; \{\langle n,\, x\rangle : n \in \mathbb{Z} \wedge x \in \mathbb{L}\}$$

[2]In what follows we write S_{co} for $S_{co}\{\!|P|\!\}$ thus leaving the involved program P implicit.

[3]The general situation in abstract interpretation is more complicated since the *approximation ordering* (for comparing program properties by implication) and the *computational ordering* (for computing fixpoints) do not coincide (see [9]). This situation does not appear for the *invariance properties* considered here.

[4]For more details on collecting semantics, see e.g. [9].

$$\begin{aligned}
\rho_{co}[2] &= \{\langle n,\ x\rangle : n \in \mathbb{Z} \wedge x = \texttt{cons}(0, \texttt{cons}(0, \texttt{cons}(0, \texttt{nil})))\} \\
\rho_{co}[3] &= \{\langle n,\ x\rangle : n > 0 \wedge x \in \{\texttt{cons}(\texttt{a}^i(0), \texttt{cons}(\texttt{b}^i(0), \texttt{cons}(\texttt{c}^i(0), \texttt{nil}))) : i \geq 0\}\} \\
\rho_{co}[4] &= \{\langle n,\ x\rangle : n > 0 \wedge x \in \{\texttt{cons}(\texttt{a}^i(0), \texttt{cons}(\texttt{b}^i(0), \texttt{cons}(\texttt{c}^i(0), \texttt{nil}))) : i > 0\}\} \\
\rho_{co}[5] &= \{\langle n,\ x\rangle : n \geq 0 \wedge x \in \{\texttt{cons}(\texttt{a}^i(0), \texttt{cons}(\texttt{b}^i(0), \texttt{cons}(\texttt{c}^i(0), \texttt{nil}))) : i > 0\}\} \\
\rho_{co}[6] &= \{\langle n,\ x\rangle : n \leq 0 \wedge x \in \{\texttt{cons}(\texttt{a}^i(0), \texttt{cons}(\texttt{b}^i(0), \texttt{cons}(\texttt{c}^i(0), \texttt{nil}))) : i \geq 0\}\}
\end{aligned}$$

In this example, we have $B_{sd} = \mathbb{P} \times \mathbb{Z} \times \mathbb{L}$ where $\mathbb{P} = \{1, \ldots, 6\}$ is the set of program points. D_{co} is $\mathbb{P} \mapsto \wp(\mathbb{Z} \times \mathbb{L})$ which is isomorphic to $\wp(\mathbb{P} \times \mathbb{Z} \times \mathbb{L})$. □

Example The standard semantics $\texttt{P}_{sd}$ of the following CML-like parallel program `P(c, n)`:

```
P(c0, 0) =
  transmit(c0, cons(0, cons(0, cons(0, nil))))
P(csn, s(n)) =
  let val cn = channel()
  in spawn[P(cn, n)];                              (3)
     let r = receive cn
     in transmit(csn,
          cons(a(hd(r)), cons(b(hd(tl(r))),
               cons(c(hd(tl(tl(r)))), nil))));
```

maps the channel `c` and natural number `n` parameters to a list of possible values transmitted on that channel `c` and the returned value (here the unit `()`). If there is no transmission, this list is empty. In case of non termination, it can be infinite. We have $B_{sd} = (\mathbb{C} \times \mathbb{N}) \mapsto (\mathbb{L}^\infty \times \mathbb{L})$. $\texttt{P}_{co} = \{\texttt{P}_{sd}\}$ is:

$$\texttt{P}_{co} = \{\lambda\langle c,\ n\rangle\cdot\langle[\texttt{cons}(\texttt{a}^n(0), \texttt{cons}(\texttt{b}^n(0), \texttt{cons}(\texttt{c}^n(0), \texttt{nil})))], ()\rangle\}$$

which belongs to $D_{co} = \wp(B_{sd})$. □

Example The ground bottom-up collecting semantics B_{co} of the following logic program:

```
P(0, cons(0, cons(0, cons(0, nil)))) :- ;        (4)
P(s(N), cons(a(X), cons(b(Y), cons(c(Z), nil))))
        :- P(N, cons(X, cons(Y, cons(Z, nil))));
```

is the set of successful ground goals:

$$B_{co} = \{\texttt{P}(\texttt{s}^n(0), \texttt{cons}(\texttt{a}^n(0), \texttt{cons}(\texttt{b}^n(0), \texttt{cons}(\texttt{c}^n(0), \texttt{nil})))) : n \geq 0\}$$

In this example, B_{sd} is the set of ground atoms and $D_{co} = \wp(B_{sd})$. □

4. Formal-Language-Based Abstraction

As an intermediate conceptual step, we consider the abstraction of program properties as formal languages (not necessarily finitely presentable). This, coupled with a notion of formal-language transformers, provides the basis for grammar-based abstract semantics, and a direct connection to set-constraints.

4.1 Formal-Language-Based Abstract Semantics

Given that behaviors in B_{sd}, hence the collecting semantics S_{co}, involve semantic values described by sentences of L associated to indexes in I, we assume that the *formal language abstract domain* is $D_{fl} \stackrel{\text{def}}{=} I \mapsto \wp(L)$. For the pointwise subset ordering $\dot{\subseteq}$, this is a complete boolean lattice: $(D_{fl}, \dot{\subseteq}, \dot{\bot}_{fl}, \dot{\top}_{fl}, \dot{\cup}, \dot{\cap}, \dot{\neg})$ where $\dot{\bot}_{fl} \stackrel{\text{def}}{=} \lambda i\cdot \emptyset$ and $\dot{\top}_{fl} \stackrel{\text{def}}{=} \lambda i\cdot L$.

Let us recall that a Galois connection is a pair $\langle\alpha,\ \gamma\rangle$ of maps $\alpha \in L \mapsto P$ and $\gamma \in P \mapsto L$ between posets $(L, \leq)$ and $(P, \preceq)$ such that for all $x \in L$ and all $y \in P$, $\alpha(x) \preceq y$ if and only if $x \leq \gamma(y)$. This is denoted $(L, \leq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (P, \preceq)$. Following [6], the approximation of program semantic properties $P_{co} \in D_{co}$ by abstract properties $\alpha_{fl}(P_{co}) \in D_{fl}$ is assumed to be formalized by a Galois connection:

$$(D_{co}, \subseteq) \underset{\alpha_{fl}}{\overset{\gamma_{fl}}{\leftrightarrows}} (D_{fl}, \dot{\subseteq})$$

This formal language abstraction $\langle\alpha_{fl},\ \gamma_{fl}\rangle$ is language and application dependent. It is built compositionally using typical dependence-free abstractions for cartesian products, functions (with domain $\Delta \subseteq I$) and lists such as:

$$\begin{aligned}
&\alpha_c \in \wp(L \times L) \mapsto (\wp(L) \times \wp(L)) \\
&\alpha_c(\Pi) = \langle\{x : \langle x,\ y\rangle \in \Pi\},\ \{y : \langle x,\ y\rangle \in \Pi\}\rangle \\
&\alpha_f \in \wp(\Delta \mapsto L) \mapsto (\Delta \mapsto \wp(L)) \\
&\alpha_f(\Phi) = \lambda\mathcal{X}\cdot\{\varphi(\mathcal{X}) : \varphi \in \Phi\} \\
&\alpha_l \in \wp(L^\infty) \mapsto \wp(L) \\
&\alpha_l(\Lambda) = \{\lambda_i : \lambda \in \Lambda \wedge i \in \operatorname{dom}\Lambda\}
\end{aligned}$$

The abstraction $\langle\alpha_{fl},\ \gamma_{fl}\rangle$ is lifted to higher-order [9] with $\vec{\alpha}_{fl} \in (D_{co} \mapsto D_{co}) \mapsto (D_{fl} \mapsto D_{fl})$ defined by $\vec{\alpha}_{fl}(F) \stackrel{\text{def}}{=} \alpha_{fl} \circ F \circ \gamma_{fl}$ and $\vec{\gamma}_{fl} \in (D_{fl} \mapsto D_{fl}) \mapsto (D_{co} \mapsto D_{co})$ defined by $\vec{\gamma}_{fl}(F^\sharp) \stackrel{\text{def}}{=} \gamma_{fl} \circ F^\sharp \circ \alpha_{fl}$ such that:

$$((D_{co} \mapsto D_{co}), \dot{\subseteq}) \underset{\vec{\alpha}_{fl}}{\overset{\vec{\gamma}_{fl}}{\leftrightarrows}} ((D_{fl} \mapsto D_{fl}), \dot{\subseteq})$$

The resulting *formal language semantics* is $S_{fl} \stackrel{\text{def}}{=} \operatorname{lfp} F_{fl}$ where the formal language transformer $F_{fl} \in D_{fl} \mapsto D_{fl}$ is $F_{fl} \stackrel{\text{def}}{=} \vec{\alpha}_{fl}(F_{co})$.

For a given programming language, the abstraction α_{fl} must be defined by induction on the syntax of programs. We consider simple examples (1), (2), (3) and (4).

Example For the functional program (1), we can approximate the collecting semantics $\texttt{f}_{co}$ of the function `f` by $\texttt{F}_{co}$ which maps a set N of values n for the argument `N` to the set of corresponding results $\{\texttt{f}(n) : n \in N\}$:

$$\begin{aligned}
\texttt{F}_{co} = \lambda N\cdot &\{\texttt{cons}(\texttt{a}^n(0), \texttt{cons}(\texttt{b}^n(0), \texttt{cons}(\texttt{c}^n(0), \texttt{nil}))) : n \in N \wedge n > 0\} \\
&\cup \{\texttt{cons}(0, \texttt{cons}(0, \texttt{cons}(0, \texttt{nil}))) : n \in N \wedge n \leq 0\}
\end{aligned}$$

Here, the abstraction $\mathtt{F}_{co} \stackrel{\text{def}}{=} \alpha^t_{fl}(\mathtt{f}_{co})$ approximates a set of functions in $\wp(\mathbb{Z} \mapsto \mathbb{L})$ by a set transformer in $\wp(\mathbb{Z}) \mapsto \wp(\mathbb{L})$. Formally:

$$\begin{aligned}\alpha^t_{fl} &\in \wp(\mathbb{Z} \mapsto \mathbb{L}) \mapsto (\wp(\mathbb{Z}) \mapsto \wp(\mathbb{L}))\\ \alpha^t_{fl}(F) &\stackrel{\text{def}}{=} \lambda X \cdot \{f(x) : f \in F \wedge x \in X\}\\ \mathtt{F}_{co} &\stackrel{\text{def}}{=} \alpha^t_{fl}(\mathtt{f}_{co})\end{aligned}$$

Furthermore, the set transformer $\mathtt{F}_{co}$ can be approximated by its codomain:

$$\begin{aligned}\alpha^r_{fl} &\in (\wp(\mathbb{Z}) \mapsto \wp(\mathbb{L})) \mapsto \wp(\mathbb{L})\\ \alpha^r_{fl}(F) &\stackrel{\text{def}}{=} \bigcup\{F(X) : X \in \wp(\mathbb{Z})\}\\ \mathtt{f}_{fl} &\stackrel{\text{def}}{=} \alpha^r_{fl}(\mathtt{F}_{co})\end{aligned}$$

For the program (1), we have:

$$\alpha_{fl}(\mathtt{f}_{co})(\mathcal{X}) = \alpha^r_{fl}(\alpha^t_{fl}(\mathtt{f}_{co})) \qquad (5)$$

The collecting semantics $\mathtt{f}_{co}$ of the function `f` is approximated by the set of the possible results when applied to all possible arguments `N`. This set is associated to the index $\mathcal{X}$ corresponding to function `f` in (1). This is an approximation of the collecting semantics in that it is no longer possible to know the specific result corresponding to a given argument. □

Example For the imperative program (2), we define:

$$\begin{aligned}\alpha_{fl}(\rho_{co}) &= \dot{\bigcup_{i=1,\ldots,6}} \alpha_{fl}(\rho_{co}[i]) \qquad (6)\\ \alpha_{fl}(\{\langle n_j,\, x_j\rangle : j \in \Delta\})(\mathcal{X}) &= \{x_j : j \in \Delta\}\end{aligned}$$

thus ignoring program points and numerical values. □

Example For the parallel program (3), we can define:

$$\begin{aligned}\alpha_{fl}(\mathtt{P}_{co})(\mathcal{X}) = \{v : \exists P \in \mathtt{P}_{co},\, n \in \mathbb{N},\, \sigma \in \mathbb{L}^\star, \qquad (7)\\ \varsigma \in \mathbb{L}^\infty : P(\mathtt{c}, n) = \langle \sigma \cdot [v] \cdot \varsigma,\, r\rangle\}\end{aligned}$$

so that the collecting semantics $\mathtt{P}_{co}$ is approximated by the set $\mathcal{X}$ of values v which can be transmitted on channel `c` for some possible value n of the argument `n` and some possible behavior P of the program `P(c, n)`. □

Example For the logic program (4), we can define:

$$\alpha_{fl}(B_{co})(\mathcal{X}) = \{x : \mathtt{P}(n, x) \in B_{co}\} \qquad (8)$$

This abstraction consists in recording only the set of possible values of the second argument of predicate `P`, which is associated to index $\mathcal{X}$. □

4.2 Specification of formal-language transformers

In order to specify the formal language semantics $S_{fl} = \text{lfp}\, F_{fl}$ at a greater level of details without considering a particular abstraction for a particular programming language we design a meta-language $\mathcal{L}_{fl}$ which can be used to specify formal language transformers F_{fl}.

Non-ground terms T in $\mathcal{L}_{fl}$ denote sets of ground terms built to some pattern:

$$T ::= x \mid f^0 \mid f^n(T_1, \ldots, T_n)$$

(any term-variable $x \in v$ is free in the term T). Meta-expressions e in $\mathcal{L}_{fl}$ denote set-of-ground-terms transformers (thus generalizing the mathematical notation $f(\mathcal{X}) \stackrel{\text{def}}{=} \{f(x) : x \in \mathcal{X}\}$):

$$\begin{aligned}e ::= &\ \mathcal{X} \mid \{T' : T_1 \in e_1, \ldots, T_n \in e_n\}\\ &\mid e_1 \cup e_2 \mid \neg e \qquad (9)\end{aligned}$$

(set-variables $\mathcal{X} \in V$ include indexes $I \subseteq V$). In $\{T' : T_1 \in e_1, \ldots, T_n \in e_n\}$, term variables in $T', T_1, \ldots, T_n$ are local while the set-variables in $e_1, \ldots, e_n$ are free.

Example In the meta-expression:

$$\begin{aligned}&\{\mathtt{cons}(0, \mathtt{cons}(0, \mathtt{cons}(0, \mathtt{nil})))\}\\ &\cup \{\mathtt{cons}(\mathtt{a}(x), \mathtt{cons}(\mathtt{b}(y), \mathtt{cons}(\mathtt{c}(z), \mathtt{nil}))) :\\ &\qquad \mathtt{cons}(x, \mathtt{cons}(y, \mathtt{cons}(z, \mathtt{nil}))) \in \mathcal{X}\}\end{aligned}$$

the local term-variables are x, y and z while $\mathcal{X}$ is a free set-variable. □

We use the abbreviation $\{T'\}$ for $\{T' : T_1 \in e_1, \ldots, T_n \in e_n\}$ when $n = 0$, $\top \stackrel{\text{def}}{=} \{x\}$, $\bot \stackrel{\text{def}}{=} \neg\top$ and $e_1 \cap e_2 \stackrel{\text{def}}{=} \{x : x \in e_1, x \in e_2\}$.

The term semantics $[\![T]\!]$ of non-ground term T is the map $[\![T]\!] \in (v \mapsto L) \mapsto L$ inductively defined as:

$$\begin{aligned}&[\![x]\!]\kappa \stackrel{\text{def}}{=} \kappa(x) \qquad [\![f^0]\!]\kappa \stackrel{\text{def}}{=} f^0\\ &[\![f^n(T_1, \ldots, T_n)]\!]\kappa \stackrel{\text{def}}{=} f^n([\![T_1]\!]\kappa, \ldots, [\![T_n]\!]\kappa)\end{aligned}$$

The semantics $\{\!|e|\!\}$ of a meta-expression e is the map $\{\!|e|\!\} \in \big(V \mapsto \wp(L)\big) \mapsto \wp(L)$ inductively defined as:

$$\begin{aligned}&\{\!|\mathcal{X}|\!\}\rho \stackrel{\text{def}}{=} \rho(\mathcal{X})\\ &\{\!|\{T' : T_1 \in e_1, \ldots, T_n \in e_n\}|\!\}\rho \stackrel{\text{def}}{=}\\ &\quad \{[\![T']\!]\kappa : \kappa \in v \mapsto L \wedge \textstyle\bigwedge_{i=1}^n [\![T_i]\!]\kappa \in \{\!|e_i|\!\}\rho\}\\ &\{\!|e_1 \cup e_2|\!\}\rho \stackrel{\text{def}}{=} \{\!|e_1|\!\}\rho \cup \{\!|e_2|\!\}\rho\\ &\{\!|\neg e|\!\}\rho \stackrel{\text{def}}{=} L - \{\!|e|\!\}\rho\end{aligned}$$

The use of negation within an expression e is assumed to be restricted to positive terms so that $\{\!|e|\!\}$ is $\dot{\subseteq}$-monotonic.

The formal language transformer F_{fl} can now be specified using the meta-language $\mathcal{L}_{fl}$ as the fixpoint equation: $\rho = F_{fl}(\rho)$ where $\rho \in D_{fl} = I \mapsto \wp(L)$. This can be detailed as the system of equations below:

$$\begin{cases}\rho(\mathcal{X}) = F_{fl}(\rho)(\mathcal{X})\\ \mathcal{X} \in \Delta\end{cases}$$

where $\Delta \subseteq V$ and, by convention, undefined variables are assumed to correspond to empty sets, that is $\forall\rho : F_{fl}(\rho)(\mathcal{X}) = \emptyset$ whenever $\mathcal{X} \in V - \Delta$. We will assume that this system of equations can be written using the meta-language $\mathcal{L}_{fl}$ as follows:

$$\begin{cases}\mathcal{X} = e_{\mathcal{X}}\\ \mathcal{X} \in \Delta\end{cases} \qquad (10)$$

where all free variables in $e_{\mathcal{X}}$ belong to Δ, $F_{fl}(\rho)(\mathcal{X}) = \{\!|e_{\mathcal{X}}|\!\}\rho$ and set variables are indexes so that $V = I$. If necessary, we can assume that trivial equations like

$\mathcal{X} = \mathcal{X}$ have been eliminated. The semantics of the system of equations (10) is:

$$\left\{\begin{array}{lcl}\rho(\mathcal{X}) & = & \llbracket e_{\mathcal{X}} \rrbracket \rho \\ \mathcal{X} \in \Delta & & \end{array}\right. \qquad \left\{\begin{array}{lcl}\rho(\mathcal{X}) & = & \emptyset \\ \mathcal{X} \in I - \Delta & & \end{array}\right.$$

By Tarski's fixpoint theorem, $\operatorname{lfp} F_{\text{fl}} = \dot{\bigcap}\{X : F_{\text{fl}}(X) \dot{\subseteq} X\}$ so that the fixpoint equation (10) has the same least solution as the system of constraints:

$$\left\{\begin{array}{lcl} e_{\mathcal{X}} & \subseteq & \mathcal{X} \\ \mathcal{X} \in \Delta & & \end{array}\right.$$

Moreover constraints such as $e_1 \cup e_2 \subseteq \mathcal{X}$ can be simplified to $e_1 \subseteq \mathcal{X}$ and $e_2 \subseteq \mathcal{X}$. The constraint $\{T' : T_1 \in e_1, \ldots, T_n \in e_n\} \subseteq \mathcal{X}$ is equivalent to $(T_1 \in e_1 \wedge \ldots \wedge T_n \in e_n) \Rightarrow T' \in \mathcal{X}$ or $(\{T_1\} \subseteq e_1 \wedge \ldots \wedge \{T_n\} \subseteq e_n) \Rightarrow \{T'\} \subseteq \mathcal{X}$. Using such simple set theoretic algebraic identities, program analysis methods formulated as a fixpoint resolution problem (10) can be reformulated as the problem of *solving a system of constraints* (see e.g. [28]).

Example Considering the formal language abstractions $\langle \alpha_{\text{fl}}, \gamma_{\text{fl}} \rangle$ respectively defined by (5), (6), (7) and (8) for programs (1), (2), (3) and (4), we get, in all cases, the same fixpoint equation:

$$\begin{array}{lcl}\mathcal{X} & = & \{\texttt{cons}(0, \texttt{cons}(0, \texttt{cons}(0, \texttt{nil})))\} \\ & & \cup\, \{\texttt{cons}(\texttt{a}(x), \texttt{cons}(\texttt{b}(y), \texttt{cons}(\texttt{c}(z), \texttt{nil}))) : \\ & & \qquad \texttt{cons}(x, \texttt{cons}(y, \texttt{cons}(z, \texttt{nil}))) \in \mathcal{X}\}\end{array} \tag{11}$$

This can also be presented in system of constraints form:

- $\{\texttt{cons}(0, \texttt{cons}(0, \texttt{cons}(0, \texttt{nil})))\} \subseteq \mathcal{X}$
- $\{\texttt{cons}(x, \texttt{cons}(y, \texttt{cons}(z, \texttt{nil})))\} \subseteq \mathcal{X} \Rightarrow$ $\{\texttt{cons}(\texttt{a}(x), \texttt{cons}(\texttt{b}(y), \texttt{cons}(\texttt{c}(z), \texttt{nil})))\} \subseteq \mathcal{X}$

In all cases, the least solution is:

$$\mathcal{X} = \{\texttt{cons}(\texttt{a}^n(0), \texttt{cons}(\texttt{b}^n(0), \texttt{cons}(\texttt{c}^n(0), \texttt{nil}))) : n \geq 0\}$$

This least solution encodes $\{a^n b^n c^n : n \geq 0\}$, a language which is <u>not</u> context-free. □

5. Grammar-Based Abstraction

In general, elements of D_{fl} are not computer-representable. A further abstraction consists in considering subsets of L that are representable by computer-implementable grammars[5]. A context-free grammar $G \in D_{\text{gr}}$ is a triple $\langle T, N, P \rangle$ where $T \subseteq A$ is the set of terminals, $N \subseteq V$ is the set of non-terminals $\mathcal{X}$ and productions in P are of the form $\mathcal{X} \Rightarrow \sigma$ where $\sigma \in (A \cup V)^\star$. The *language $\mathcal{L}_G(\mathcal{X})$ generated* by the grammar for non-terminal $\mathcal{X}$ is $\mathcal{L}_G(\mathcal{X}) \stackrel{\text{def}}{=} \{w \in A^\star : \mathcal{X} \stackrel{\star}{\Rightarrow}_G w\}$.

Grammar-based analysis and its equivalent set-constraint-based presentation are restricted to *regular tree grammars* [23, 20] (describing finite trees or terms of a Herbrand universe). In this case the productions can be normalized in Greibach normal form: $\mathcal{X} \Rightarrow f^0$ or $\mathcal{X} \Rightarrow f^n(\mathcal{X}_1, \ldots, \mathcal{X}_n)$ where f^0, $f^n \in \Sigma$ are function symbols and $\mathcal{X}_1, \ldots, \mathcal{X}_n$ are non-terminals. The grammar abstract domain is defined as:

$$D_{\text{gr}} \stackrel{\text{def}}{=} \{G : G \text{ is a regular tree grammar}\}$$

The concretization function $\gamma_{\text{gr}} \in D_{\text{gr}} \mapsto D_{\text{fl}}$ is the language generated by the grammar for each non-terminal $\gamma_{\text{gr}} \stackrel{\text{def}}{=} \lambda G \cdot \lambda \mathcal{X} \cdot \mathcal{L}_G(\mathcal{X})$. Grammars are ordered by inclusion of the generated languages $G_1 \dot{\subseteq} G_2 \stackrel{\text{def}}{=} \forall \mathcal{X} \in V : \mathcal{L}_{G_1}(\mathcal{X}) \subseteq \mathcal{L}_{G_2}(\mathcal{X})$. Grammar equivalence $G_1 \dot{\equiv} G_2$ is defined as $G_1 \dot{\subseteq} G_2 \wedge G_2 \dot{\subseteq} G_1$. Equivalent grammars can be identified by reasoning on the quotient poset $(D_{\text{gr}}/\dot{\equiv}, \dot{\subseteq})$, later simply written $(D_{\text{gr}}, \dot{\subseteq})$.

Some questions about infinite sets of values defined by a regular tree grammar G can be answered algorithmically [1] such as $\mathcal{L}_G(\mathcal{X}) = \emptyset$, $t \in \mathcal{L}_G(\mathcal{X})$ and $\exists t : \mathcal{L}_G(\mathcal{X}) = \{t\}$ where $\mathcal{X}$ is a non-terminal and t a term. In particular abstract questions of the form $\mathcal{L}_{G_1}(\mathcal{X}) \subseteq \mathcal{L}_{G_2}(\mathcal{X})$ are decidable.

Besides regular tree grammars, many other different isomorphic formalisms can be used to describe regular tree languages such as μ-expressions, systems of fixpoint equations, systems of constraints, etc. By Ginsburg & Rice, Schützenberger's theorem, the languages generated by a grammar $G = \langle T, N, P \rangle$ for each non-terminal $\mathcal{X} \in N$ is the $\subseteq$-least solution to the system of fixpoint equations[6]:

$$\left\{\begin{array}{l} \mathcal{L}_G(\mathcal{X}) = \{f^0 \mid \mathcal{X} \Rightarrow f^0 \in P\} \cup \\ \qquad \{f^n(t_1, \ldots, t_n) \mid \mathcal{X} \Rightarrow f^n(\mathcal{X}_1, \ldots, \mathcal{X}_n) \in P \\ \qquad\qquad \wedge \forall i = 1, \ldots, n : t_i \in \mathcal{L}_G(\mathcal{X}_i)\} \\ \mathcal{X} \in N \end{array}\right.$$

which representing the language $\mathcal{L}_G(\mathcal{X})$ by the set-variable $\mathcal{X}$ itself leads to:

$$\left\{\begin{array}{l} \mathcal{X} = \bigcup\{\{f^0\} \mid \mathcal{X} \Rightarrow f^0 \in P\} \cup \\ \qquad \bigcup\{\{f^n(t_1, \ldots, t_n) : \forall i = 1, \ldots, n : t_i \in \mathcal{X}_i\} \\ \qquad\qquad \mid \mathcal{X} \Rightarrow f^n(\mathcal{X}_1, \ldots, \mathcal{X}_n) \in P\} \\ \mathcal{X} \in N \end{array}\right.$$

so that, by defining $\mathsf{f}^0 \stackrel{\text{def}}{=} \{f^0\}$ and $\mathsf{f}^n(L_1, \ldots, L_n) \stackrel{\text{def}}{=} \{f^n(t_1, \ldots, t_n) \mid \forall i = 1, \ldots, n : t_i \in L_i\}$, we obtain the system of fixpoint equations:

$$\left\{\begin{array}{l} \mathcal{X} = \bigcup\{\mathsf{f}^0 \mid \mathcal{X} \Rightarrow f^0 \in P\} \cup \\ \qquad \bigcup\{\mathsf{f}^n(\mathcal{X}_1, \ldots, \mathcal{X}_n) \mid \\ \qquad\qquad \mathcal{X} \Rightarrow f^n(\mathcal{X}_1, \ldots, \mathcal{X}_n) \in P\} \\ \mathcal{X} \in N \end{array}\right. \tag{12}$$

[5]The use of a finite grammar abstract domain seems in contradiction with the claim "No use is made of abstract domains (such as those commonly employed in abstract-interpretation styles of program analysis [5]). We remark that the results of the analysis are typically infinite sets of values and that we make no *a priori* requirement that these sets be finitely presentable" at the end of section 3 of [19]. We nevertheless imagine no way of circumventing the a priori requirement that finite computer representations are to be found in the implementation of set-based program analysis [16].

[6]Observe that we have two different fixpoints, one for the grammar transformer F_{gr} defining a grammar G and one for the grammar G defining the language generated by G.

By Tarski's fixpoint theorem, (12) has the same least solution as the system of inequations:

$$\begin{cases} \mathcal{X} \supseteq \bigcup\{\mathsf{f}^0 \mid \mathcal{X} \Rightarrow f^0 \in P\} \cup \\ \qquad \bigcup\{\mathsf{f}^n(\mathcal{X}_1, \ldots, \mathcal{X}_n) \mid \\ \qquad\qquad \mathcal{X} \Rightarrow f^n(\mathcal{X}_1, \ldots, \mathcal{X}_n) \in P\} \\ \mathcal{X} \in N \end{cases}$$

or equivalently as the collection of set-constraints:

$$\begin{cases} \mathsf{f}^0 \subseteq \mathcal{X}, \quad \mathsf{f}^n(\mathcal{X}_1, \ldots, \mathcal{X}_n) \subseteq \mathcal{X} \\ \mathcal{X} \Rightarrow f^0 \in P, \quad \mathcal{X} \Rightarrow f^n(\mathcal{X}_1, \ldots, \mathcal{X}_n) \in P \end{cases}$$

Reciprocally, given a collection of set-constraints:

$$\begin{cases} \mathsf{f}_i^0 \subseteq \mathcal{X}, \quad \mathsf{f}_j^n(\mathcal{X}_1, \ldots, \mathcal{X}_n) \subseteq \mathcal{X} \\ i, j \in \Xi \end{cases}$$

the grammar:

$$\begin{cases} \mathcal{X} \Rightarrow f_i^0, \quad \mathcal{X} \Rightarrow f_j^n(\mathcal{X}_1, \ldots, \mathcal{X}_n) \\ i, j \in \Xi \end{cases}$$

generates the same least solution. This isomorphism seems to have been first suspected in [20] where [23, 29] is mentioned as "earlier related work". It is more apparent in [16, 17] where regular tree grammars are used as an "explicit representation" of set-constraints. Set constraints transformation algorithms equivalent to those on regular tree grammars [1, 2] are also given by [3].

6. Grammar Abstract Semantics

We now study how to approximate the language transformer F_{fl} so as to define a computable abstract semantics of programs using grammars (or equivalently, sets of constraints). The restriction to regular tree grammars does not directly solve the program analysis problem since $(D_{\text{gr}}, \dot{\subseteq})$, which is not a complete partial order, contains infinite strictly increasing chains of grammars without limits (least upper bound).

Example The grammar transformer F_{gr} for program (1) is the grammar abstraction of the formal language transformer F_{fl} defined by equation (11). For example it can be defined as[7]:

$$\begin{aligned} &F_{\text{gr}}(\langle T,\ N,\ P\rangle) = \langle T \cup \{\mathtt{0}, \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{cons}, \mathtt{nil}\},\ N \cup \{\mathcal{X}\}, \\ &\{\mathcal{X} \Rightarrow \mathtt{cons}(\mathtt{0}, \mathtt{cons}(\mathtt{0}, \mathtt{cons}(\mathtt{0}, \mathtt{nil})))\} \cup \\ &\quad \{\mathcal{X} \Rightarrow \mathtt{cons}(\mathtt{a}(x), \mathtt{cons}(\mathtt{b}(y), \mathtt{cons}(\mathtt{c}(z), \mathtt{nil}))) : \\ &\qquad \mathcal{X} \Rightarrow \mathtt{cons}(x, \mathtt{cons}(y, \mathtt{cons}(z, \mathtt{nil}))) \in P\}\rangle \end{aligned}$$

The iterates ${F_{\text{gr}}}^n(\perp_{\text{gr}})$, where $\perp_{\text{gr}} = \langle \emptyset,\ \emptyset,\ \emptyset\rangle$, correspond to the strictly increasing chain of grammars G^n, $n \geq 0$ with productions:

$$\begin{cases} \mathcal{X} \Rightarrow \mathtt{cons}\big(\mathtt{a}^i(\mathtt{0}), \mathtt{cons}\big(\mathtt{b}^i(\mathtt{0}), \\ \qquad\qquad \mathtt{cons}\big(\mathtt{c}^i(\mathtt{0}), \mathtt{nil}\big)\big)\big) \\ i = 0, \ldots, n \end{cases}$$

which is both infinite and without limit. □

[7]Another grammar abstraction leading to a different grammar transformer F_{gr} will be proposed in forthcoming Sec. 6.2.2.

The situation has been considered e.g. in [7] and is similar to [11] where there exist infinite chains of infinite sets (polyhedra) with finite representations (set of inequalities or system of generators) without limits (e.g. polyhedra inscribed in a circle). Three standard ways to cope with the problem consist of using either:

— a widening,
— or a finitary abstract property transformer,
— or else a (program-dependent) abstract domain satisfying the ascending chain condition (or even finite).

6.1 Widening

A widening $\nabla \in D_{\text{gr}} \times D_{\text{gr}} \mapsto D_{\text{gr}}$ is such that for all R, $S \in D_{\text{gr}}$ we have $\gamma_{\text{gr}}(R) \subseteq \gamma_{\text{gr}}(R \nabla S)$ and $\gamma_{\text{gr}}(S) \dot{\subseteq} \gamma_{\text{gr}}(R \nabla S)$ and for all increasing chains $S^k, k \geq 0$, the chain $R^0 = S^0, \ldots, R^{k+1} = R^k \nabla S^k, \ldots$ is not strictly increasing for $\dot{\subseteq}$. The widening ∇ can be used to upper-approximate possibly non-existent least upper-bounds and to enforce convergence [5]. Now, the iterates $R^0 = \perp_{\text{gr}}$ and $R^{n+1} = R^n \nabla F_{\text{gr}}(R^n)$ ultimately stabilize, so that we can define:

$$S_{\text{gr}} \stackrel{\text{def}}{=} R^n,\ n \text{ is (the least}^{8}) \ k \text{ such that } F_{\text{gr}}(R^k) \dot{\subseteq} R^k \qquad (13)$$

in which case, we conclude that the abstract semantics is sound $S_{\text{co}} \subseteq \gamma_{\text{fl}} \circ \gamma_{\text{gr}}(S_{\text{gr}})$.

We have an abstract interpretation with iteration such that:

(a) *The abstract domain D_{gr} has infinitely many abstract values G representing infinite concrete sets $\gamma_{\text{gr}}(G)(\mathcal{X})$;*
(b) *All abstract values in domain D_{gr} have a finite computer representation;*
(c) *The abstract domain $(D_{\text{gr}}, \dot{\subseteq})$ has infinite strictly increasing chains for $\dot{\subseteq}$ without limits in D_{gr} (which, consequently is not a complete partial order);*
(d) *F_{gr} satisfies the soundness condition $F_{\text{fl}} \circ \gamma_{\text{gr}} \dot{\subseteq} \gamma_{\text{gr}} \circ F_{\text{gr}}$ and is computable;*
(e) *The increasing chain ${F_{\text{gr}}}^n(\perp_{\text{gr}}), n \geq 0$ of iterates is not converging in a finite number of steps to an approximation S_{gr} of $S_{\text{fl}} = \mathit{lfp}\, F_{\text{fl}}$ (so that a widening is necessary).*

Widenings for regular tree grammars have been defined implicitly in [1, 30][9]. A similar widening is obtained by enforcing the iterates to deterministic regular tree grammars (i.e. without two different productions of the form $\mathcal{X} \Rightarrow f^n(T_1, \ldots, T_n)$ and $\mathcal{X} \Rightarrow f^n(T_1', \ldots, T_n')$). Such a widening $G_1 \nabla G_2$ can be defined as the repeated application to $G_1 \cup G_2$ defined as $\langle T_1 \cup T_2,\ N_1$

[8]Alternatively, by removing the requirement that n be the least rank of a postfixpoint, we can account for approximate (but more efficient) inclusion tests $\trianglelefteq$, where $R \trianglelefteq S$ implies $R \subseteq S$ but not reciprocally, as e.g. in [1]

[9]Although no widening is mentioned explicitly, it is used in section 5.4 and lemma 5.3 of [1] and section 6 of [30].

$\cup N_2, P_1 \cup P_2\rangle$ of the transformation which consists in replacing all $m > 1$ productions[10] of the form:

$$\begin{cases} \mathcal{X} \Rightarrow f^n(T_1^i, \ldots, T_n^i) \\ i = 1, \ldots, m \end{cases}$$

by:

$$\begin{cases} \mathcal{X} \Rightarrow f^n(Z_1, \ldots, Z_n) \\ \begin{cases} Z_k \Rightarrow T_k^i \\ i = 1, \ldots, m \end{cases} \\ k = 1, \ldots, n \end{cases}$$

where:

1. $\{Z_k : k = 1, \ldots, n\}$ is a set of non-terminals not used in any production of G_1 or G_2;
2. If some T_k^i is a non-terminal then all its occurrences in the productions are replaced by Z_k;
3. Useless productions are eliminated (e.g. $\mathcal{X} \Rightarrow \mathcal{X}$ or productions $\mathcal{X} \Rightarrow T$ whose lefthand side $\mathcal{X}$ never appears in a production necessary to define an index corresponding to a program object).

With this widening relational information is lost. Many variants can be considered. A simple example consists in performing the grammar transformation only if m is strictly greater than some given threshold $l \geq 1$ (which can be determined statically or may vary dynamically so as to decrease in order to control the time/space consumption of the iterative computation (13)).

Example When applied to (11), we start with a grammar with no production so that after one iteration we get a grammar with one production:

$$\mathcal{X} \Rightarrow \texttt{cons}(\texttt{0}, \texttt{cons}(\texttt{0}, \texttt{cons}(\texttt{0}, \texttt{nil})))$$

One more iteration leads to:

$$\begin{array}{l} \mathcal{X} \Rightarrow \texttt{cons}(\texttt{0}, \texttt{cons}(\texttt{0}, \texttt{cons}(\texttt{0}, \texttt{nil}))) \\ \mathcal{X} \Rightarrow \texttt{cons}(\texttt{a}(\texttt{0}), \texttt{cons}(\texttt{b}(\texttt{0}), \texttt{cons}(\texttt{c}(\texttt{0}), \texttt{nil}))) \end{array}$$

The widening simplifies this into:

$$\begin{array}{ll} \mathcal{X} \Rightarrow \texttt{cons}(A, B) & \\ A \Rightarrow \texttt{0} & B \Rightarrow \texttt{cons}(\texttt{0}, \texttt{cons}(\texttt{0}, \texttt{nil})) \\ A \Rightarrow \texttt{a}(\texttt{0}) & B \Rightarrow \texttt{cons}(\texttt{b}(\texttt{0}), \texttt{cons}(\texttt{c}(\texttt{0}), \texttt{nil})) \end{array}$$

then into:

$$\begin{array}{ll} \mathcal{X} \Rightarrow \texttt{cons}(A, B) & C \Rightarrow \texttt{0} \\ A \Rightarrow \texttt{0} & C \Rightarrow \texttt{b}(\texttt{0}) \\ A \Rightarrow \texttt{a}(\texttt{0}) & D \Rightarrow \texttt{cons}(\texttt{0}, \texttt{nil}) \\ B \Rightarrow \texttt{cons}(C, D) & D \Rightarrow \texttt{cons}(\texttt{c}(\texttt{0}), \texttt{nil}) \end{array}$$

and then into:

$$\begin{array}{ll} \mathcal{X} \Rightarrow \texttt{cons}(A, B) & C \Rightarrow \texttt{b}(\texttt{0}) \\ A \Rightarrow \texttt{0} & D \Rightarrow \texttt{cons}(E, F) \\ A \Rightarrow \texttt{a}(\texttt{0}) & E \Rightarrow \texttt{0} \\ B \Rightarrow \texttt{cons}(C, D) & E \Rightarrow \texttt{c}(\texttt{0}) \\ C \Rightarrow \texttt{0} & F \Rightarrow \texttt{nil} \end{array}$$

Replacing, for presentation simplification, the non-terminals with a single production by the righthand side of this production, we obtain in more compact form:

$$\begin{array}{l} \mathcal{X} \Rightarrow \texttt{cons}(A, \texttt{cons}(B, \texttt{cons}(C, \texttt{nil}))) \\ A \Rightarrow \texttt{0} \mid \texttt{a}(\texttt{0}) \quad B \Rightarrow \texttt{0} \mid \texttt{b}(\texttt{0}) \quad C \Rightarrow \texttt{0} \mid \texttt{c}(\texttt{0}) \end{array}$$

The next iteration gives:

$$\begin{array}{l} \mathcal{X} \Rightarrow \texttt{cons}(\texttt{0}, \texttt{cons}(\texttt{0}, \texttt{cons}(\texttt{0}, \texttt{nil}))) \\ \mathcal{X} \Rightarrow \texttt{cons}(\texttt{a}(A), \texttt{cons}(\texttt{b}(B), \texttt{cons}(\texttt{c}(C), \texttt{nil}))) \\ \mathcal{X} \Rightarrow \texttt{cons}(A, \texttt{cons}(B, \texttt{cons}(C, \texttt{nil}))) \\ A \Rightarrow \texttt{0} \mid \texttt{a}(\texttt{0}) \quad B \Rightarrow \texttt{0} \mid \texttt{b}(\texttt{0}) \quad C \Rightarrow \texttt{0} \mid \texttt{c}(\texttt{0}) \end{array}$$

so that application of the widening simplification rules leads to:

$$\begin{array}{l} \mathcal{X} \Rightarrow \texttt{cons}(A', \texttt{cons}(B', \texttt{cons}(C', \texttt{nil}))) \\ A' \Rightarrow \texttt{0} \mid \texttt{a}(\texttt{0}) \mid \texttt{a}(A') \quad B' \Rightarrow \texttt{0} \mid \texttt{b}(\texttt{0}) \mid \texttt{b}(B') \\ C' \Rightarrow \texttt{0} \mid \texttt{c}(\texttt{0}) \mid \texttt{c}(C') \end{array}$$

and to:

$$\begin{array}{l} \mathcal{X} \Rightarrow \texttt{cons}(A'', \texttt{cons}(B'', \texttt{cons}(C'', \texttt{nil}))) \\ A'' \Rightarrow \texttt{0} \mid \texttt{a}(A'') \quad B'' \Rightarrow \texttt{0} \mid \texttt{b}(B'') \\ C'' \Rightarrow \texttt{0} \mid \texttt{c}(C'') \end{array}$$

This is the final result since the next iterate proves convergence. This widening leads to the approximation of the language $\{a^n b^n c^n : n \geq 0\}$ by $a^\star b^\star c^\star$. $\square$

6.2 Finitary abstract property transformer

Another solution to the convergence problem is to mask the explicit use of a widening when defining a *finitary grammar transformer* $F_{\text{gr}}^{\triangledown} \in D_{\text{gr}} \mapsto D_{\text{gr}}$ satisfying the soundness condition $F_{\text{fl}} \circ \gamma_{\text{gr}} \mathrel{\dot{\subseteq}} \gamma_{\text{gr}} \circ F_{\text{gr}}^{\triangledown}$, which is computable and satisfies the following *convergence condition*[11]:

$$\exists n \geq 0 : {F_{\text{gr}}^{\triangledown}}^n(\bot_{\text{gr}}) \mathrel{\dot{\equiv}} {F_{\text{gr}}^{\triangledown}}^{n+1}(\bot_{\text{gr}}); \qquad (14)$$

specifying that ${F_{\text{gr}}^{\triangledown}}^n(\bot_{\text{gr}}), n \geq 0$ ultimately reaches a fixpoint. Defining:

$$S_{\text{gr}} \stackrel{\text{def}}{=} {F_{\text{gr}}^{\triangledown}}^m(\bot_{\text{gr}}) \text{ where } m \text{ is the least } n \text{ such that } {F_{\text{gr}}^{\triangledown}}^n(\bot_{\text{gr}}) \mathrel{\dot{\equiv}} {F_{\text{gr}}^{\triangledown}}^{n+1}(\bot_{\text{gr}})$$

we conclude that the abstract semantics is computable and sound: $S_{\text{co}} \subseteq \gamma_{\text{fl}} \circ \gamma_{\text{gr}}(S_{\text{gr}})$. Moreover, if $F_{\text{gr}}^{\triangledown}$ is monotonic then $S_{\text{gr}} = \operatorname{lfp} F_{\text{gr}}^{\triangledown}$ and this least fixpoint is computable. This kind of completeness result is often argued to be preferable to the use of a widening. This is illusory since $\operatorname{lfp} F_{\text{gr}}^{\triangledown}$ is computable whereas the truly interesting $\operatorname{lfp} F_{\text{gr}}$ is not (and may even not exist at all e.g. when limits are missing). Moreover a finitary abstract property transformer can always be defined using a widening: $F_{\text{gr}}^{\triangledown} \stackrel{\text{def}}{=} \lambda X \cdot X \mathbin{\triangledown} F_{\text{gr}}(X)$.

Using a finitary abstract property transformer, we have an abstract interpretation with iterative fixpoint computation such that (a), (b), (c) hold while (d) and (e) are now:

(d') $F_{\text{gr}}^{\triangledown}$ satisfies the soundness condition $F_{\text{fl}} \circ \gamma_{\text{gr}} \mathrel{\dot{\subseteq}} \gamma_{\text{gr}} \circ F_{\text{gr}}^{\triangledown}$ and is computable;

(e') The increasing chain ${F_{\text{gr}}^{\triangledown}}^n(\bot_{\text{gr}}), n \geq 0$ of iterates is converging in a finite number of steps to the least fixpoint lfp $F_{\text{gr}}^{\triangledown}$ (so that no widening is necessary).

[10] If $m = 1$ no widening is necessary since there is still no sign that the number of productions might increase forever.

[11] If $F_{\text{gr}}^{\triangledown}$ is known to be monotone or extensive, the weaker condition $F_{\text{gr}}^{\triangledown}({F_{\text{gr}}^{\triangledown}}^n(\bot_{\text{gr}})) \mathrel{\dot{\subseteq}} {F_{\text{gr}}^{\triangledown}}^n(\bot_{\text{gr}})$ is sufficient.

6.2.1 A meta-language for finitary grammar transformers

One way of ensuring that $F^{\triangledown}_{\mathrm{gr}}$ is finitary in the sense of (14) is to define $F^{\triangledown}_{\mathrm{gr}}$ as in (10) using a restricted form of meta-expressions (9). Various forms of restrictions can be considered. One generic way of enforcing these restrictions is to require $F^{\triangledown}_{\mathrm{gr}}$ to be written in a specific meta-language $\mathcal{L}_{\mathrm{gr}}$ restricting meta-expressions to the form:

$$\begin{aligned} e \ ::= \ & \mathcal{X} \mid \bot \mid \{f^0\} \mid \{f^n(\mathcal{X}_1, \ldots, \mathcal{X}_n)\} \mid \\ & f^{n}{}^{-1}_{(i)}(\mathcal{X}) \mid e_1 \cup e_2 \end{aligned}$$

where $1 \leq i \leq n$ and $f^n \in \Sigma$. The semantics of the projection is:

$$\{\!| f^{n}{}^{-1}_{(i)}(e) |\!\}\rho \ \stackrel{\mathrm{def}}{=}\ \{t_i : f^n(t_1, \ldots, t_n) \in \{\!| e |\!\}\rho\}$$

In abstract form, the system of equations can also be written using a variant of the "extended grammar" notation of [23]:

$$\left\{ \begin{array}{l} \mathcal{X} \ \Rightarrow \ r_{\mathcal{X}} \\ \mathcal{X} \in \Delta \end{array} \right. \qquad (15)$$

where $\Delta \subseteq I$ and the righthand sides are[12]:

$$\begin{aligned} r \ ::= \ & \mathcal{X} \mid \bot \mid \mathsf{f}^0 \mid \mathsf{f}^n(\mathcal{X}_1, \ldots, \mathcal{X}_n) \mid \\ & \mathcal{X}.\mathsf{f}^{n}{}^{-1}_{(i)} \mid r_1 \mid r_2 \end{aligned} \qquad (16)$$

It is interpreted as the fixpoint equation:

$$\langle T,\ N,\ P\rangle \ = \ F^{\triangledown}_{\mathrm{gr}}(\langle T,\ N,\ P\rangle)$$

where $F^{\triangledown}_{\mathrm{gr}}$ is a finitary grammar transformer, which given a grammar $\langle T,\ N,\ P\rangle$, returns the grammar with productions:

$$\bigcup_{\mathcal{X} \in \Delta} [\![\mathcal{X} \ \Rightarrow \ r_{\mathcal{X}}]\!]^{\sharp}_P \qquad (17)$$

such that $[\![\mathcal{X} \ \Rightarrow \ r_{\mathcal{X}}]\!]^{\sharp}_P$ is inductively defined as follows. If $\mathcal{X} \neq \mathcal{Y}$ then:

$$[\![\mathcal{X} \Rightarrow \mathcal{Y}]\!]^{\sharp}_P \ = \ \{\mathcal{X} \Rightarrow T \mid \mathcal{Y} \Rightarrow T \in P\}$$

while if $\mathcal{X} = \mathcal{Y}$ then:

$$[\![\mathcal{X} \Rightarrow \mathcal{Y}]\!]^{\sharp}_P \ = \ [\![\mathcal{X} \Rightarrow \bot]\!]^{\sharp}_P \ = \ \emptyset$$

$$[\![\mathcal{X} \Rightarrow \mathsf{f}^0]\!]^{\sharp}_P \ = \ \{\mathcal{X} \Rightarrow \mathsf{f}^0\}$$

$$\begin{aligned} & [\![\mathcal{X} \Rightarrow \mathsf{f}^n(\mathcal{X}_1, \ldots, \mathcal{X}_n)]\!]^{\sharp}_P \ = \\ & \quad \{\mathcal{X} \Rightarrow \mathsf{f}^n(\mathcal{X}_1, \ldots, \mathcal{X}_n) : \textstyle\bigwedge_{j=1}^{n} \mathcal{L}_P(\mathcal{X}_j) \neq \emptyset\} \end{aligned}$$

$$\begin{aligned} [\![\mathcal{X} \Rightarrow \mathcal{Y}.\mathsf{f}^{n}{}^{-1}_{(i)}]\!]^{\sharp}_P \ = \ & \{\mathcal{X} \Rightarrow T_i : \\ & \mathcal{Y} \Rightarrow \mathsf{f}^n(\mathcal{X}_1, \ldots, \mathcal{X}_i, \ldots, \mathcal{X}_n) \in P \wedge \\ & \mathcal{X}_i \Rightarrow T_i \in P \wedge \forall j \neq i : \mathcal{L}_P(\mathcal{X}_j) \neq \emptyset\} \end{aligned}$$

$$[\![\mathcal{X} \Rightarrow r_1 \mid r_2]\!]^{\sharp}_P \ = \ [\![\mathcal{X} \Rightarrow r_1]\!]^{\sharp}_P \cup [\![\mathcal{X} \Rightarrow r_2]\!]^{\sharp}_P$$

Now the iterates converge to the least solution:

$$\begin{aligned} & \exists n \geq 0 : F^{\triangledown}_{\mathrm{gr}}{}^{n}(\bot_{\mathrm{gr}}) \doteq F^{\triangledown}_{\mathrm{gr}}{}^{n+1}(\bot_{\mathrm{gr}}) \doteq \\ & \mathrm{lfp}\, F^{\triangledown}_{\mathrm{gr}} \stackrel{\mathrm{def}}{=} S_{\mathrm{gr}} \end{aligned} \qquad (18)$$

[12]This notation (15) and (16) sets up a confusion between grammars and grammar transformers. For example the grammar rule $\mathcal{X} \Rightarrow \sigma$ denotes the grammar transformer $\lambda G \cdot G \cup \{\mathcal{X} \Rightarrow \sigma\})$. This may be confusing since grammars and grammar transformers have then to be formalized in the same way.

The language (16) can be enriched while preserving this property (18) as in [2, 17, 27][13].

Example For program (1), we can approximate F_{gr} defined at (11) by the grammar transformer $F^{\triangledown}_{\mathrm{gr}}$ as follows:

$$\begin{array}{ll} \mathcal{X} \Rightarrow \mathtt{cons}(\mathtt{0}, \mathtt{cons}(\mathtt{0}, \mathtt{cons}(\mathtt{0}, \mathtt{nil}))) & \\ \quad \mid \ \mathtt{cons}(\mathtt{a}(\mathcal{X}_1), \mathtt{cons}(\mathtt{b}(\mathcal{X}_2), \mathtt{cons}(\mathtt{c}(\mathcal{X}_3), \mathtt{nil}))) & \\ \quad\quad \mathcal{X}_1 \Rightarrow \mathcal{X}.\mathtt{cons}^{-1}_{(1)} \qquad \mathcal{V}_2 \Rightarrow \mathcal{V}_1.\mathtt{cons}^{-1}_{(2)} & \\ \quad\quad \mathcal{V}_1 \Rightarrow \mathcal{X}.\mathtt{cons}^{-1}_{(2)} \qquad \mathcal{X}_3 \Rightarrow \mathcal{V}_2.\mathtt{cons}^{-1}_{(1)} & \\ \quad\quad \mathcal{X}_2 \Rightarrow \mathcal{V}_1.\mathtt{cons}^{-1}_{(1)} & \end{array}$$

The iterative fixpoint computation leads to the approximation of the language $\{a^n b^n c^n : n \geq 0\}$ by $a^{\star} b^{\star} c^{\star}$[14]. To see this, let us consider the simpler fixpoint equation:

$$\begin{array}{ll} \mathcal{X} \Rightarrow \mathtt{cons}(\mathcal{A}, \mathcal{N}) & \mathcal{X} \Rightarrow \mathtt{cons}(\mathcal{B}, \mathcal{N}) \\ \mathcal{A} \Rightarrow \mathtt{0} & \mathcal{B} \Rightarrow \mathtt{a}(\mathcal{C}) \\ \mathcal{N} \Rightarrow \mathtt{nil} & \mathcal{C} \Rightarrow \mathcal{X}.\mathtt{cons}^{-1}_{(1)} \end{array}$$

The iterates are the grammars $\langle T^n,\ N^n,\ P^n\rangle$ such that:

$P^0 = \emptyset$
$P^1 = \{\mathcal{A} \Rightarrow \mathtt{0}, \mathcal{N} \Rightarrow \mathtt{nil}\}$
$P^2 = \{\mathcal{X} \Rightarrow \mathtt{cons}(\mathcal{A}, \mathcal{N}), \mathcal{A} \Rightarrow \mathtt{0}, \mathcal{N} \Rightarrow \mathtt{nil}\}$
$P^3 = \{\mathcal{X} \Rightarrow \mathtt{cons}(\mathcal{A}, \mathcal{N}), \mathcal{A} \Rightarrow \mathtt{0}, \mathcal{N} \Rightarrow \mathtt{nil}, \mathcal{C} \Rightarrow \mathtt{0}\}$
$P^4 = \{\mathcal{X} \Rightarrow \mathtt{cons}(\mathcal{A}, \mathcal{N}), \mathcal{A} \Rightarrow \mathtt{0}, \mathcal{N} \Rightarrow \mathtt{nil},$
$\quad \mathcal{X} \Rightarrow \mathtt{cons}(\mathcal{B}, \mathcal{N}), \mathcal{B} \Rightarrow \mathtt{a}(\mathcal{C}), \mathcal{C} \Rightarrow \mathtt{0}\}$
For $n \geq 5$:
$P^n = \{\mathcal{X} \Rightarrow \mathtt{cons}(\mathcal{A}, \mathcal{N}), \mathcal{A} \Rightarrow \mathtt{0}, \mathcal{N} \Rightarrow \mathtt{nil},$
$\quad \mathcal{X} \Rightarrow \mathtt{cons}(\mathcal{B}, \mathcal{N}), \mathcal{B} \Rightarrow \mathtt{a}(\mathcal{C}), \mathcal{C} \Rightarrow \mathtt{0}, \mathcal{C} \Rightarrow \mathtt{a}(\mathcal{C})\}$

$\square$

6.2.2 Abstraction of an infinitary formal language transformer by a finitary grammar transformer

We show how to abstract F_{fl} written using the meta-language (9) by $F_{\mathrm{gr}} = \vec{\alpha}_{\mathrm{gr}}(F_{\mathrm{fl}})$ written using the meta-language (16) while satisfying the soundness condition $F_{\mathrm{fl}} \circ \gamma_{\mathrm{gr}} \subseteq \gamma_{\mathrm{gr}} \circ F_{\mathrm{gr}}$. If F_{fl} is defined by the system of equations (10) then $F_{\mathrm{gr}} = \vec{\alpha}_{\mathrm{gr}}(F_{\mathrm{fl}})$ is defined by the grammar with productions:

$$\alpha_{\mathrm{gr}}(\{\mathcal{X} = e_{\mathcal{X}} : \mathcal{X} \in \Delta\}) \ \stackrel{\mathrm{def}}{=}\ \bigcup_{\mathcal{X} \in \Delta} \alpha_E(\mathcal{X} = e_{\mathcal{X}})$$

The abstraction of each equation is a grammar production plus possibly auxiliary productions:

$$\begin{aligned} \alpha_E(\mathcal{X} = e) \ \stackrel{\mathrm{def}}{=}\ & \text{let } \langle e',\ R\rangle = \alpha_e(e) \text{ in} \\ & \{\mathcal{X} \Rightarrow e'\} \cup R \end{aligned}$$

$\alpha_e(e)$ is used to compute the righthand side of a production corresponding to an equation $\mathcal{X} = e$. This introduces auxiliary productions R which involve new auxiliary set-variables $\mathcal{Z}$, $\mathcal{Z}_1$, ..., $\mathcal{Z}_n$ and $\mathcal{Z}_{\top}$ with rules

[13]For the sake of brevity we do not consider intersection and restricted negation since the general approach remains exactly the same in that case.

[14]Observe that the value of variable `X` given by a context-sensitive language must be approximated by a regular language, showing that the assertion that "Set based approximations ignore all information about inter-variable dependencies, but make no other approximations" on page 133 of [17] is overstated.

$\{\mathcal{Z}_\top \Rightarrow f^n(\mathcal{Z}_\top, \ldots, \mathcal{Z}_\top) : f^n \in \Sigma\}$ representing the absence of information. $\alpha_e(e)$ is defined by induction on the syntax of e. We start with simple cases. In particular negation is ignored:

$$\alpha_e(\mathcal{X}) \stackrel{\text{def}}{=} \langle \mathcal{X},\, \emptyset\rangle \quad \alpha_e(\top) \stackrel{\text{def}}{=} \langle \mathcal{Z}_\top,\, \emptyset\rangle$$
$$\alpha_e(\bot) \stackrel{\text{def}}{=} \langle \bot,\, \emptyset\rangle \quad \alpha_e(\neg e) \stackrel{\text{def}}{=} \alpha_e(\top)$$

We approximate:

$$\alpha_e(\{T' : T_1 \in e_1, \ldots, T_n \in e_n\}) \stackrel{\text{def}}{=} \alpha_e(\bigcap_{i=1}^{n}\{T' : T_i \in e_i\})$$

by an intersection, which apart from simple cases, is also ignored:

$$\alpha_e(\bot \cap e_2) \stackrel{\text{def}}{=} \alpha_e(\bot) \quad \alpha_e(e_1 \cap \bot) \stackrel{\text{def}}{=} \alpha_e(\bot)$$
$$\alpha_e(e_1 \cap \top) \stackrel{\text{def}}{=} \alpha_e(e_1) \quad \alpha_e(\top \cap e_2) \stackrel{\text{def}}{=} \alpha_e(e_2)$$
$$\alpha_e(e_1 \cap e_2) \stackrel{\text{def}}{=} \alpha_e(e_1)$$

The definition of $\alpha_e(e_1 \cap e_2)$ seems arbitrary, but this is a common practice in program analysis e.g. when tests are ignored [19]. Grammars are well suited to handle union exactly:

$$\alpha_e(e_1 \cup e_2) \stackrel{\text{def}}{=} \alpha_e(e_1) \oplus \alpha_e(e_2)$$
$$\langle e_1,\, R_1\rangle \oplus \langle e2,\, R_2\rangle \stackrel{\text{def}}{=} \langle e_1 \mid e_2,\, R_1 \cup R_2\rangle$$

We now define $\alpha_e(\{T\})$ by induction on the syntax of term T:

$$\alpha_e(\{x\}) \stackrel{\text{def}}{=} \alpha_e(\top) \qquad \alpha_e(\{f^0\}) \stackrel{\text{def}}{=} \mathsf{f}^0$$
$$\alpha_e(\{f^n(T_1, \ldots, T_n)\}) \stackrel{\text{def}}{=} \langle \mathsf{f}^n(\mathcal{Z}_1, \ldots, \mathcal{Z}_n),\, \textstyle\bigcup_{i=1}^n \alpha_E(\mathcal{Z}_i = T_i)\rangle$$

We define $T[x := \mathcal{X}]$ to be the substitution of set-variable $\mathcal{X}$ for the term-variable x within term T, so as to obtain the righthand side of a grammar production:

$$x[x := \mathcal{X}] \stackrel{\text{def}}{=} \mathcal{X} \qquad y[x := \mathcal{X}] \stackrel{\text{def}}{=} \mathcal{Z}_\top$$
$$f^0[x := \mathcal{X}] \stackrel{\text{def}}{=} \mathsf{f}^0$$
$$f^n(T_1, \ldots, T_n)[x := \mathcal{X}] \stackrel{\text{def}}{=} \mathsf{f}^n(T_1[x := \mathcal{X}], \ldots, T_n[x := \mathcal{X}])$$

After these preliminaries, we define $\alpha_e(\{T' : T \in e\})$ by induction on the syntax of e and, if necessary on the syntax of T':

$$\alpha_e(\{T' : T \in \bot\}) \stackrel{\text{def}}{=} \alpha_e(\bot)$$
$$\alpha_e(\{T' : T \in \top\}) \stackrel{\text{def}}{=} \alpha_e(\{T'\})$$
$$\alpha_e(\{T' : x \in \mathcal{X}\}) \stackrel{\text{def}}{=} \{T'[x := \mathcal{X}]\}$$
$$\alpha_e(\{T' : f^0 \in \mathcal{X}\}) \stackrel{\text{def}}{=} \alpha_e(\{T'\})$$
$$\alpha_e(\{T' : f^n(T_1, \ldots, T_n) \in \mathcal{X}\}) \stackrel{\text{def}}{=} \alpha_e(\{T' : T_1 \in f^{n-1}_{(1)}(\mathcal{X}), \ldots, T_n \in f^{n-1}_{(n)}(\mathcal{X})\})$$

Observe that once again tests $f^0 \in \mathcal{X}$ are ignored (but could be taken into account since this question is decidable for regular tree grammars). Dependencies between components of an n-ary constructor are ignored in $\alpha_e(\{T' : f^n(T_1, \ldots, T_n) \in \mathcal{X}\})$. Let us define the notation:

$$\langle e,\, R\rangle \uplus R' \stackrel{\text{def}}{=} \langle e,\, R \cup R'\rangle$$

The projection can be reduced to the required form $\mathcal{Y} \Rightarrow \mathcal{X}.f^{n-1}_{(i)}$ as follows:

$$\alpha_e(\{T' : y \in f^{n-1}_{(i)}(\mathcal{X}), T_{i+1} \in f^{n-1}_{(i+1)}(\mathcal{X}), \ldots, T_n \in f^{n-1}_{(n)}(\mathcal{X})\})$$
$$\stackrel{\text{def}}{=} \alpha_e(\{T'[y := \mathcal{Z}] : T_{i+1} \in f^{n-1}_{(i+1)}(\mathcal{X}), \ldots, T_n \in f^{n-1}_{(n)}(\mathcal{X})\}) \uplus \{\mathcal{Z} \Rightarrow \mathcal{X}.f^{n-1}_{(i)}\}$$

$$\alpha_e(\{T' : g^0 \in f^{n-1}_{(i)}(\mathcal{X}), T_{i+1} \in f^{n-1}_{(i+1)}(\mathcal{X}), \ldots, T_n \in f^{n-1}_{(n)}(\mathcal{X})\})$$
$$\stackrel{\text{def}}{=} \alpha_e(\{T' : T_{i+1} \in f^{n-1}_{(i+1)}(\mathcal{X}), \ldots, T_n \in f^{n-1}_{(n)}(\mathcal{X})\})$$

$$\alpha_e(\{T' : g^n(T_1, \ldots, T_n) \in f^{n-1}_{(i)}(\mathcal{X}), T_{i+1} \in f^{n-1}_{(i+1)}(\mathcal{X}), \ldots, T_n \in f^{n-1}_{(n)}(\mathcal{X})\})$$
$$\stackrel{\text{def}}{=} \alpha_e(\{T' : T_{i+1} \in f^{n-1}_{(i+1)}(\mathcal{X}), \ldots, T_n \in f^{n-1}_{(n)}(\mathcal{X})\})$$

Let us finish the definition of $\alpha_e(\{T' : T \in e\})$ by induction on the syntax of e:

$$\alpha_e(\{T' : T \in \{T''\}\}) \stackrel{\text{def}}{=} \alpha_e(\{T' : T \in \mathcal{Z}\}) \uplus \alpha_E(\mathcal{Z} = \{T''\})$$
$$\alpha_e(\{T' : T \in \{T'' : T''' \in e\}\}) \stackrel{\text{def}}{=} \alpha_e(\{T' : T \in \mathcal{Z}\}) \uplus \alpha_E(\mathcal{Z} = \{T'' : T''' \in e\})$$

Again union is handled exactly while intersection and negation are approximated very roughly:

$$\alpha_e(\{T' : T \in e_1 \cup e_2\}) \stackrel{\text{def}}{=} \alpha_e(\{T' : T \in e_1\}) \oplus \alpha_e(\{T' : T \in e_2\})$$
$$\alpha_e(\{T' : T \in \bot \cap e_2\}) \stackrel{\text{def}}{=} \alpha_e(\{T' : T \in \bot\})$$
$$\alpha_e(\{T' : T \in e_1 \cap \bot\}) \stackrel{\text{def}}{=} \alpha_e(\{T' : T \in \bot\})$$
$$\alpha_e(\{T' : T \in \top \cap e_2\}) \stackrel{\text{def}}{=} \alpha_e(\{T' : T \in e_2\})$$
$$\alpha_e(\{T' : T \in e_1 \cap \top\}) \stackrel{\text{def}}{=} \alpha_e(\{T' : T \in e_1\})$$
$$\alpha_e(\{T' : T \in e_1 \cap e_2\}) \stackrel{\text{def}}{=} \alpha_e(\{T' : T \in e_1\}$$
$$\alpha_e(\{T' : T \in \neg e\}) \stackrel{\text{def}}{=} \alpha_e(\{T' : T \in \top\})$$

Observe that in set-constraint-based analysis, part of this abstraction process is performed as a simplification process while solving set-constraints.

6.2.3 Finitary Set Constraints Transformer

The finitary grammar transformer (15) can be rewritten in a sets of constraints transformer form:

$$\begin{cases} \mathcal{X} \supseteq c_{\mathcal{X}} \\ \mathcal{X} \in \Delta \end{cases} \tag{19}$$

where $\Delta \subseteq I$ and the righthand sides are:

$$c \;::=\; \mathcal{X} \mid \bot \mid \mathsf{f}^0 \mid \mathsf{f}^n(\mathcal{X}_1, \ldots, \mathcal{X}_n) \mid \mathcal{X}.\mathsf{f}^{n-1}_{(i)} \mid c_1 \cup c_2$$

For solving this system of inequations iteratively, a chaotic iteration with initial empty set of constraints may be chosen to start with the evaluation of the constraint transformers of the form $\mathcal{X} \supseteq c_1 \cup c_2$ which introduce the constraints $\mathcal{X} \supseteq c_1$, $\mathcal{X} \supseteq c_2$. This can be understood as conversion in the standard form of [17]. When this is done, these components of the system of inequations (19) are definitely stable whence no longer need to be considered. Moreover we have seen that this operation can be done when the system of inequations (19) (or (15)) is established. The same way the constraint transformers like $\mathcal{X} \supseteq \mathsf{f}^0$ and $\mathcal{X} \supseteq \mathsf{f}^n(\mathcal{X}_1, \ldots, \mathcal{X}_n)$ can be evaluated only once. Alternatively, since these constraint transformers are written in the same syntax as the constraints themselves, they can be understood as initial constraints. Then the chaotic iteration only concerns the projections $\mathcal{X} \supseteq \mathcal{Y}.\mathsf{f}^{n\text{-}1}_{(i)}$. These iterates can then be confused with those of a constraint transformer algorithm. With this explanation one can always claim that (19) is not solved iteratively [17, 16][15] . However the resolution is isomorphic with the least fixpoint chaotic iteration that we have just defined.

6.3 A Program Dependent Finite Grammar Abstract Domain

A finite grammar abstract domain $D_{\mathrm{gr}}[P] \subseteq D_{\mathrm{gr}}$ can be considered for each particular program P[16]. This can be used as an alternative proof of (18). Indeed, observe that the system of equations (15) defines a transformer $F^{\triangledown}_{\mathrm{gr}}$ over grammars such that if G is in Greibach normal form (each production $\mathcal{X} \Rightarrow r_{\mathcal{X}}$ has a standardized righthand side of the form f^0 or $\mathsf{f}^n(\mathcal{X}_1, \ldots, \mathcal{X}_n)$) then so is $F^{\triangledown}_{\mathrm{gr}}(G)$. Since I is finite, there are finitely many non-terminals $\mathcal{X}$, $\mathcal{X}_1, \ldots, \mathcal{X}_n$. For each particular program there are also finitely many possible terminals f^0, f^n, ...since they must all occur in the program (formally in the system of equations (15)). Moreover the initial normalization of the system of equations (15) into Greibach normal form introduces finitely many new non-terminals in I. Let $D_{\mathrm{gr}}[P]$ be the set of grammars in Greibach normal form that is with non-terminals in I, terminals appearing in program P and rules with standardized righthand sides of the form f^0 or $\mathsf{f}^n(\mathcal{X}_1, \ldots, \mathcal{X}_n)$. For each P, $D_{\mathrm{gr}}[P]$ is finite since there are only finitely many possible different productions. However $D_{\mathrm{gr}} = \bigcup_P D_{\mathrm{gr}}[P]$ is infinite (since the terminal vocabulary may be infinite). The convergence (18) follows immediately from the observation that for a particular program P we need not reason on $F^{\triangledown}_{\mathrm{gr}} \in D_{\mathrm{gr}} \mapsto D_{\mathrm{gr}}$. Since $F^{\triangledown}_{\mathrm{gr}}$ is defined by (15), we can, by definition of $[\![\mathcal{X} \Rightarrow r_{\mathcal{X}}]\!]^{\sharp}_{P}$ in (17), equally well reason on $F^{\triangledown}_{\mathrm{gr}} \in D_{\mathrm{gr}}[P] \mapsto D_{\mathrm{gr}}[P]$. By restricting the grammar/set-constraint-based analysis of a given program P to the finite abstract domain $D_{\mathrm{gr}}[P]$, we understand grammar/set-constraint-based program analysis as a simple finite abstract interpretation[17].

We have an abstract interpretation such that for each program P:

- *The abstract domain $D_{\mathrm{gr}}[P]$ has finitely many abstract values G representing infinite concrete sets $\gamma_{\mathrm{gr}}[G](\mathcal{X})$;*
- *All abstract values in domain $D_{\mathrm{gr}}[P]$ have a finite representation;*
- *The abstract property transformer F_{gr} maps an abstract value $G \in D_{\mathrm{gr}}[P]$ into an abstract value $F_{\mathrm{gr}}(G) \in D_{\mathrm{gr}}[P]$[18];*
- *The increasing chain ${F_{\mathrm{gr}}}^n(\bot_{\mathrm{gr}})$, $n \geq 0$ of iterates is finite whence converges to the least fixpoint lfp F_{gr}.*

The choice of a program dependent abstract domain is a common practice, e.g. when choosing to associate abstract values to variables occurring in the program.

6.4 Language independent implementation

It is essential to specify language-independent abstract interpretations in order to obtain reuseable implementations or at least to minimize the language-dependent part.

An implementation of a program analysis by grammar/set-constraint-based approximation of formal languages would have a programming-language-dependent part consisting in a compiler for translating a program into the formal language transformer F_{fl} written in the meta-language $\mathcal{L}_{\mathrm{fl}}$ (more precisely into a sentence of the meta-language the semantics of which is F_{fl}).

The programming-language-independent part of this implementation would consists of a simplifier of the formal language transformer F_{fl} (written using the meta-language (9)) into the finitary grammar transformer $F^{\triangledown}_{\mathrm{gr}}$ (written in the meta-language $\mathcal{L}_{\mathrm{gr}}$) such that $F_{\mathrm{gr}} = \vec{\alpha}_{\mathrm{gr}}(F_{\mathrm{fl}})$. This is possible since the abstraction $\vec{\alpha}_{\mathrm{gr}}$ is computable. Moreover, reusable chaotic fixpoint computation algorithms can be used to evaluate lfp F_{gr}.

Such partly reusable implementations have been designed which are mono-language and multi-analysis (see e.g. [26]). The proposition here is mono-analysis and multi-language. Practical experience is necessary as far as incorporation in real systems is concerned.

7. Combining Grammar and Non-Grammar-Based Abstractions

Grammar-based abstractions can be combined with non-grammar-based ones. For example, one can consider

[15][16] states that "The fundamental difference between set based analysis and other approaches in literature (which are based on abstract interpretation) is that set based analysis does not employ an iterative least fixpoint computation over a finitary domain."

[16]This was suggested to us by Alain Deutsch.

[17]Therefore the statement on page 198 of [20] that abstract interpretation is "limited by the essentially finite bound on the number of different states between which the approximate reasoning can discriminate" indeed applies to set-based analysis. That it does not apply to abstract interpretation was shown in [8] using widening.

[18]Here the abstract values encoding abstract properties are grammars.

an index I which is partitioned into a set of symbolic variables I_{gr} and a set of numerical variables I_{num}. An example of reduced product [6] with $\mathcal{X}, \mathcal{Y} \in I_{\mathrm{gr}}$ and $\mathcal{A}, \mathcal{B} \in I_{\mathrm{num}}$ would be:

$$\begin{cases} \mathcal{X} \Rightarrow \mathtt{cons}(\mathcal{A}, \mathcal{X}) \mid \mathtt{nil} & \mathcal{A} = 0 \bmod 2 \\ \mathcal{Y} \Rightarrow \mathtt{cons}(\mathcal{B}, \mathcal{Y}) \mid \mathtt{nil} & \mathcal{B} = 1 \bmod 2 \end{cases}$$

which seems an interesting alternative to [18] where numerical values are handled symbolically (i.e. $\mathcal{A} \Rightarrow 0 \mid \mathcal{A}+2 \mid \mathcal{A}-2$ and $\mathcal{B} \Rightarrow 1 \mid \mathcal{B}+2 \mid \mathcal{B}-2$). This can be generalized to dependence-sensitive numerical abstract domains [15] such as:

$$\begin{cases} \mathcal{X} \Rightarrow \mathtt{cons}(\mathcal{A}, \mathcal{X}) \mid \mathtt{nil} & 2\mathcal{A} + 3\mathcal{B} = 0 \bmod 2 \\ \mathcal{Y} \Rightarrow \mathtt{cons}(\mathcal{B}, \mathcal{Y}) \mid \mathtt{nil} & \end{cases}$$

One particular case of reduced product consists in considering the product of the concrete and abstract domains, using the abstract domain for defining widenings on the concrete one, which is essentially what is implicitly done in [21].

8. Context-Sensitive Grammar Abstractions

Although the abstractions considered so far are context-free with respect to indexes, nothing prevents using the usual abstract interpretation techniques to obtain relational, i.e. context sensitive or polyvariant analyses.

One such technique, *abstract lattice completion* [6, 9], consists in considering an abstract domain made of sets of formal languages/grammars/sets of constraints $D^{\star}_{\mathrm{df}} \stackrel{\mathrm{def}}{=} I \mapsto \wp(\wp(L))$. For example, a different formal language/grammar/set of constraints can be used for each function call to obtain a polyvariant analysis. This is considered as an implementation trick in section 10 of [19], maybe because, for finite sets, it can be obtained by copying parts of the program (see also e.g. [30]). Since in general the size of such sets (or the number of copies) must be limited, a widening is necessary (e.g. which naïvely consists in limiting the size of sets to two elements as in [19] (so that the idea of exact approximation of the collecting semantics ($S_{\mathrm{gr}} = \alpha(S_{\mathrm{co}})$) is no longer valid and knowledge of the program analysis algorithm is required to predict which parts of the program will be copied) or may involve more sophisticated ideas such as *dynamic partitioning*, as in [4]).

Another powerful way of refining the grammar analysis is to consider an abstract domain which consists of a grammar where a different counter is used for each production to count the number of times the production is used in the derivation of a symbolic value. By analyzing numerical relationships between these counter values, one can express context-dependent non-uniform information akin to [12, 13, 14]. For example, approximating numerical relationships by linear equalities [25], we would obtain the language $\{a^{\ell} b^{\ell} c^{\ell} : \ell \geq 0\}$ for program (1) in the form of the following grammar with counters:

$$\begin{cases} \mathcal{X} \stackrel{i}{\Longrightarrow} \mathtt{cons}(A, \mathtt{cons}(B, \mathtt{cons}(C, \mathtt{nil}))) & \\ A \stackrel{j}{\Longrightarrow} \mathtt{0} & B \stackrel{m}{\Longrightarrow} \mathtt{b}(B) \\ A \stackrel{k}{\Longrightarrow} \mathtt{a}(A) & C \stackrel{p}{\Longrightarrow} \mathtt{0} \\ B \stackrel{l}{\Longrightarrow} \mathtt{0} & C \stackrel{q}{\Longrightarrow} \mathtt{c}(C) \\ i = j = l = p = 1 \wedge k = m = q & \end{cases} \qquad (20)$$

This also provides a simple way to combine the analysis of numerical and non-numerical data. For example in program (1), the initial value $n \geq 0$ of variable `n` can be taken into account. Then using the above context-dependent grammar abstraction with approximation of numerical relationships by linear inequalities [11], we would obtain (20) with the additional constraint $k \leq n$, which is now a fairly precise invariant for the program corresponding to the language $\{a^{\ell} b^{\ell} c^{\ell} : 0 \leq \ell \leq n\}$.

Observe that the above abstract interpretation is obtained by refining F_{fl} as defined by $F_{\mathrm{fl}} \stackrel{\mathrm{def}}{=} \vec{\alpha}_{\mathrm{fl}}(F_{\mathrm{co}})$, (10), (9) and cannot be obtained from $F^{\triangledown}_{\mathrm{gr}}$ defined in (15), (16) since all dependencies have already been lost in $F^{\triangledown}_{\mathrm{gr}}$. This shows the limits of set-constraint-based program analysis: the initial dependence-free approximation is too coarse thus disallowing later refinements.

9. Conclusion

There has been a long tradition of using grammar-based and set-constraint based analysis for functional and logic programming languages. These have been traditionally seen as fundamentally different from abstract interpretation. On the contrary, we have shown that these formal language/grammar/set-constraints program analyses <u>are</u> abstract interpretations. This point of view has several advantages:

- The presentation of these analyses is independent of a particular (imperative, functional, parallel, logic, etc) style of programming language and of a particular style of fixpoint specification (fixpoint operator, system of equations, system of constraints, closure-condition, rule-based formal system or game-theoretic form) [10];
- A reusable implementation can be designed by expressing the abstract property transformers (F_{fl}, F_{gr}, $F^{\triangledown}_{\mathrm{gr}}$) in common meta-languages ($\mathcal{L}_{\mathrm{fl}}$, $\mathcal{L}_{\mathrm{gr}}$);
- Combinations with other analyses (e.g. to handle arithmetic) are possible;
- Extensions to relational analyses allowing for various forms of polyvariance are easy (e.g. by lattice completion) and no longer need to be presented as implementation tricks;
- Abstract domains can be refined (e.g. using counters) to obtain powerful context sensitive program analyzes.
- The idea of using grammar codings of infinite symbolic sets can be generalized. For example, one can

consider:

– Infinite regular trees, e.g. to handle lazy functional programming languages with infinite data structures or PROLOG III;
– Generalizations of formal (string) language theory such as (hyper)graph language theory and grammars (to describe abstract properties) and (hyper)graph rewriting techniques (to describe abstract semantic transformers) to handle aliases, sharing, etc.

Acknowledgments This work was partly supported by ESPRIT BRA 8130 LOMAPS. We thank Nevin Heintze for his seminar at ENS on Oct. 21, 1994 and for the ensuing animated discussions, Christopher Colby, Alain Deutsch and Arnaud Venet for their remarks on a preliminary version of this paper and the referees for their kind comments.

References

[1] A. Aiken & B. R. Murphy. Implementing regular tree expressions. *Proc. 5th FPCA*, LNCS 523, 427–447. Springer-Verlag, 1991.

[2] A. Aiken & B. R. Murphy. Static type inference in a dynamically typed language. In *18th ACM POPL*, 279–290, 1991.

[3] A. Aiken & E. L. Wimmers. Solving systems of set constraints (extended abstract). In *Proc. 7th IEEE LICS'92*, 329–340, 1992.

[4] F. Bourdoncle. Abstract interpretation by dynamic partitioning. *J. Func. Prog.*, 2(4), 1992.

[5] P. Cousot & R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM POPL*, 238–252, 1977.

[6] P. Cousot & R. Cousot. Systematic design of program analysis frameworks. In *6th ACM POPL*, 269–282, 1979.

[7] P. Cousot & R. Cousot. Abstract interpretation frameworks. *J. Logic and Comp.*, 2(4):511–547, 1992.

[8] P. Cousot & R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. *Proc. PLILP'92*, LNCS 631, 269–295. Springer-Verlag, 1992.

[9] P. Cousot & R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages). In *Proc. 1994 IEEE ICCL*, 95–112, 1994.

[10] P. Cousot & R. Cousot. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form. In *Proc. CAV'95*, LNCS, Springer-Verlag, to appear, 1995.

[11] P. Cousot & N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM POPL*, 84–97, 1978.

[12] A. Deutsch. An operational model of strictness properties and its abstraction. *Proc. 1991 Glasgow University Functional Programming Workshop*, Workshops in Comp., 82–99. Springer-Verlag, 1991.

[13] A. Deutsch. A storeless model of aliasing and its abstraction using finite representations of right-regular equivalence relations. In *Proc. 1992 IEEE ICCL*, 2–13, 1992.

[14] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *Proc. ACM PLDI*, 230–241, 1994.

[15] P. Granger. Static analysis of linear congruence equalities among variables of a program. *Proc. TAPSOFT'91*, Vol. 1 (CAAP'91), LNCS 493, 169–192. Springer-Verlag, 1991.

[16] N. Heintze. Practical aspects of set based analysis. In *Proc. Joint Int. Conf. and Symp. on Logic Programming*, 765–779. MIT Press, 1992.

[17] N. Heintze. *Set Based Program Analysis.* PhD, Carnegie Mellon University, Pittsburgh, Pa., Oct. 1992.

[18] N. Heintze. Set based analysis and arithmetic. In *Proc. ACM Conf. Lisp & Func. Prog.*, 306–317, 1993.

[19] N. Heintze. Set-based analysis of ML programs (extended abstract). In *Proc. ACM Conf. Lisp & Func. Prog.*, 1994.

[20] N. Heintze & J. Jaffar. A finite presentation theorem for approximating logic programs (extended abstract). In *17th ACM POPL*, 197–209, 1990.

[21] N. Heintze & J. Jaffar. An engine for logic program analysis. In *Proc. 7th IEEE LICS'92*, 318–328, 1992.

[22] N. D. Jones. Flow analysis of lazy higher-order functional programs. In S. Abramsky & C. Hankin, eds., *Abstract Interpretation of Declarative Languages*, 103–122. Ellis Horwood, 1987.

[23] N. D. Jones & S. S. Muchnick. Flow analysis and optimization of LISP-like structures. In *6th ACM POPL*, 244–256, 1979.

[24] N. D. Jones & S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *9th ACM POPL*, 66–74, 1982.

[25] M. Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

[26] B. Le Charlier & P. Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for Prolog. In *Proc. 1992 IEEE ICCL*, 137–146, 1992.

[27] P. Mishra & U. Reddy. Declaration-free type checking. In *12th ACM POPL*, 7–21, 1985.

[28] J. Palsberg. Global program analysis in constraint form. *Proc. 19th CAAP'94*, LNCS 787, 276–290. Springer-Verlag, 1994.

[29] J. Reynolds. Automatic computation of data set definitions. In *Information Processing'68*, 456–461. North Holland, 1969.

[30] M. H. Sørensen. A grammar-based data-flow analysis to stop deforestation. *Proc. 19th CAAP'94*, LNCS 787, 335–351. Springer-Verlag, 1994.