

# Object-Oriented Constraint Satisfaction \*

keywords: constraint satisfaction  
ID: 330

## Abstract

Constraint satisfaction problems are limited in that their domain values are treated as atomic entities. Object-Oriented Constraint Satisfaction Problems (OOCSPs) synthesize the object-oriented paradigm with declarative constraints to provide a rich language for describing domains. This paper introduces the concept of an OOCSP, describes algorithms for solving certain classes of OOCSPs, and gives an undecidability result that shows not all OOCSPs can be solved.

## Introduction

In its most general form, a constraint satisfaction problem (CSP) is defined as a finite set of variables with associated domains and a set of constraints that impose certain relationships among the variables. A solution to a CSP is a variable assignment that satisfies all the constraints.

The domain values of a CSP are treated as atomic entities; consequently, solving a CSP with infinite domains requires a solver with hard-coded knowledge of those domains. Representing such domains as sets of structured objects often allows for finite encodings, thus obviating the need for that hard-coded knowledge. Object-Oriented Constraint Satisfaction Problems<sup>1</sup> synthesize the object-oriented paradigm with declarative constraints to provide a rich language for describing domains as sets of structured objects. Any infinite domain that can be encoded in an OOCSP can be used without hard-coding knowledge of that domain into an OOCSP solver.

For example, one can use an OOCSP in the context of utility computing to construct configurations of computing resources. A computer is built out of a processor, some amount of memory, and a hard drive. A server is built out of a computer, an operating system, a suite of software, and

\*We would like to thank Akhil Sahai and Sharad Singhal for providing the inspiration for this work and for their contributions in many discussions. Lyle Ramshaw, Nathaniel Love, and Charles Petrie also played pivotal roles in the development of OOCSPs. We would also like to thank Hewlett-Packard for its support of the work reported herein.

Copyright © 2005, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

<sup>1</sup>The term Object-Oriented Constraint Satisfaction was originally used by Paltrinieri (1994) His version was no more expressive than a CSP, but in this paper OOCSPs are strictly more expressive.

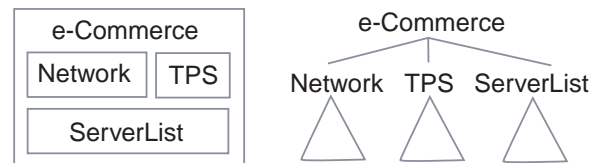


Figure 1: An e-Commerce object and its tree representation.

an IP address. An e-Commerce site is built out of a network connection, an estimated number of transactions-per-second(TPS), and some number of servers, each of which has a unique IP address. The set of servers is dependent on the value for TPS. As the load increases so does the number of servers. Thus the set of e-Commerce sites is unbounded, but its inherent structure admits a finite encoding.<sup>2</sup> Solving a CSP where one of its domains is the set of e-Commerce sites requires the solver to have information about e-Commerce sites built-in. As new software is released or hardware becomes antiquated, the e-Commerce domain changes, as must the code for the CSP solver. But because this domain can be encoded natively as an OOCSP, no specialized support is required for the OOCSP solver.

We have chosen to formalize this synthesis of an object hierarchy and declarative constraints as a context free grammar (CFG) with constraints on its production rules. Fig. 1 shows how an e-Commerce site object can be represented as a CFG parse tree. These CFGs with constraints are commonly called augmented grammars in the field of Natural Language Processing. This paper describes algorithms for solving certain classes of OOCSPs and gives an undecidability result that shows not all OOCSPs can be solved.

The first section introduces the notion of a Tree Grammar, which serves as the input to an OOCSP. The second section formally defines the semantics of Tree Grammars and the object-oriented constraint satisfaction problem. The following section gives algorithms for solving two classes of OOCSPs, and the fourth section explains an undecidability result. A discussion follows.

<sup>2</sup>Technically the set of IP addresses is finite, but for the sake of pedagogy we will treat IP addresses as though they were simply unique identifiers and therefore infinite.

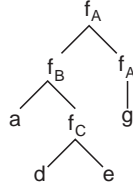


Figure 2: A simple tree

## Tree Grammars

As illustrated in the last section, we have chosen to formalize objects of OOCSPs as CFG parse trees. It turns out that assigning trees to types in an OOCSP is analogous to constructing solutions to a CSP. Thus trees and their types are the central focus of this paper.

Trees can be succinctly represented as functional terms. For example, the tree in Fig. 2 can be represented as the functional term

$$f_A(f_B(a, f_C(d, e)), f_A(g))$$

Sets of trees that share a common pattern at the root can be represented by functional terms with variables. The set of all trees with three children where the root is  $f_A$ , the first child is  $c$ , and the second child has root  $f_B$  with two children of its own can be represented by the *tree top*

$$f_A(c, f_B(x, y), z)$$

Each tree can be assigned multiple types, and each type can include multiple trees. The tree  $f_A(f_B(a, f_C(d, e)), f_A(g))$  can be assigned the type  $r_H$  by writing the statement

$$r_H(f_A(f_B(a, f_C(d, e)), f_A(g)))$$

Likewise, the tree top  $f_A(c, f_B(x, y), z)$  can be assigned the type  $r_H$  with the expression

$$r_H(f_A(c, f_B(x, y), z))$$

Statements that assign trees to types can be constrained so that they only apply to a subset of all the trees that match the tree top. The only way to differentiate trees that match the tree top is by differentiating its variables. Type constraints, equality constraints, distinction constraints ( $\neq$ ), and constructor constraints each impose a different kind of relationship among those variables.

Type constraints require a variable to be of a specific type, as has already been discussed. Equality constraints between two variables  $x$  and  $y$  require  $x$  and  $y$  to be instantiated with two copies of the same tree, i.e. their functional representations must be syntactically identical. Distinction constraints require the trees to be syntactically different. Constructor constraints use a new tree top to require the variables, when arranged in a particular manner, to construct a tree of a particular type.

For example,

$$r_H(f_A(x, y, z)) \Leftarrow r_B(x) \wedge x = y \wedge r_C(f_C(x, a, y, x))$$

states that all trees matching  $f_A(x, y, z)$  are of type  $r_H$  as long as (1)  $x$  is of type  $r_B$ , (2)  $y$  is the same tree as  $x$ , and (3)  $f_C(x, a, y, x)$  is a tree of type  $r_C$ .

Sometimes it is convenient to set up a type hierarchy and to define new types as the intersection of old types, e.g.  $r_H(x) \Leftarrow r_G(x) \wedge r_J(x)$ . These two kinds of rules are sufficient to formally define the input to an OOCSP, a Tree Grammar, which is a variation on the definition for a context free grammar.

We first review the formal definition for a CFG to illustrate how it compares to a Tree Grammar. Recall that a context free grammar is a four-tuple  $\langle N, T, P, S \rangle$ .

$N$  : finite set of symbols called the nonterminals

$T$  : finite set of symbols, disjoint from  $N$ , called the terminals

$S$  : an element of  $N$ , the start symbol

$P$  : finite set of rules of the form

$$A \rightarrow \gamma_1 \dots \gamma_m, \text{ where } \gamma_i \in N \cup T \text{ and } A \in N$$

A Tree Grammar includes nonterminals (nodes)  $N$ , terminals (leaves)  $L$ , the start symbol  $S$ , and a finite set of production rules  $P$ , but will additionally include a set of types  $T$ , and a set of variables  $V$ .

**Definition 1 (Tree Grammars)** A Tree Grammar is a six-tuple  $\langle N, L, T, V, S, P \rangle$ .

$N$  : finite set of symbols called the nodes, e.g.  $f_A$

$L$  : nonempty, finite set of symbols, called the leaves, e.g.  $a$

$T$  : nonempty, finite set of symbols, called the types, e.g.  $r_A$

$V$  : finite set of symbols, called the variables, e.g.  $x, y, z$

$S$  : an element of  $T$ , the start type

The sets  $N, L, T$ , and  $V$  are disjoint.

$P$  : finite set of rules of the following two forms.

1.  $r_A(p) \Leftarrow C_1 \wedge \dots \wedge C_k$

$$r_A \in T$$

$$p \in \text{Terms}[N, L, V]$$

$$C_i \in \begin{cases} t = u, \text{ where } t, u \in V \cup L \\ t \neq u, \text{ where } t, u \in V \cup L \\ r_B(t), \text{ where } r_B \in T \text{ and } t \in \text{Terms}[N, L, V] \end{cases}$$

2.  $r_A(x) \Leftarrow r_{B_1}(x) \wedge \dots \wedge r_{B_n}(x)$

$$r_A, r_{B_i} \in T$$

$$x \in V$$

$\text{Terms}[x, y, z]$  is the set of all functional terms (tree tops) produced from the nodes, leaves, and variables  $x, y, z$ , respectively. No arities are placed on the function constants, but every term is finite.

We will refer to rules of type (1) as *composition rules* because they compose smaller objects to build larger ones. Composition rules of the form  $r_A(f_A(x, y))$  will be called *simple typing rules*. Rules of type (2) will be called *intersection rules*, and when  $n$  is one, e.g.  $r_A(x) \Leftarrow r_B(x)$ , they shall be referred to as *subtyping rules*.

A context free grammar is actually a special case of a Tree Grammar. There are no intersection rules, and each composition rule is of a special form. For example, the CFG production rule  $A \rightarrow adBCE$  defines the type  $A$  to include trees whose root is  $A$  and whose children are  $a, d$ , and trees of types  $B, C$ , and  $E$ , in that order. Using our representation, this rule would be written as:

$$r_A(f_A(a, d, x, y, z)) \Leftarrow r_B(x) \wedge r_C(y) \wedge r_E(z)$$

In CFGs, the tree top only mentions a root and its children—no child’s children are mentioned. Moreover, every child that is not a terminal is required to have a particular type.

Notice also that every tree of type  $r_A$  is represented with the functional term  $f_A$ , and every  $f_A$  is of type  $r_A$ . Because CFGs do not allow a tree to have multiple types, the distinction between types and nodes is nonexistent. Tree Grammars, on the other hand, explicitly allow a tree to have multiple types through the subtyping rules. The restrictions a CFG naturally imposes on all its type definitions are useful when solving OOCSPs and will be mentioned again when those algorithms are discussed.

## Tree Grammar Semantics and OOCSPs

Recall that the language of a CFG  $G$  is the set of strings parsed by trees rooted at the start symbol in  $G$ . The language of a Tree Grammar is the set of trees whose type is the start symbol.

**Definition 2** (*Trees( $G$ )*) Let  $G = \langle N, L, T, V, S, P \rangle$  be a Tree Grammar. The set of trees for  $G$  is defined as follows.

- Each member of  $L$  is in  $Trees(G)$ .
- If  $t_1, \dots, t_n$  are in  $Trees(G)$  and  $f_X$  is in  $N$  then  $f_X(t_1, \dots, t_n)$  is in  $Trees(G)$ .
- $Trees(G)$  is the minimum set satisfying the above.

A tree can only be of a certain type if there is a set of production rules that force it to be of that type.

**Definition 3** (*Types*) Let  $G = \langle N, L, T, V, S, P \rangle$  be a Tree Grammar. The types of trees in  $G$  are defined inductively as follows.

- Each rule  $r_A(t)$ , where  $t$  is a tree top, ensures type  $r_A$  includes all instances of  $t$ .
- If  $t$  is a tree of types  $r_{B_1}, \dots, r_{B_n}$  the rule  $r_A(x) \Leftarrow r_{B_1}(x) \wedge \dots \wedge r_{B_n}(x)$  ensures type  $r_A$  includes  $t$ .
- Suppose  $v$  is a variable assignment for the variables in the tree top expression  $e$  to trees. Consider the rule  $r_A(e) \Leftarrow C_1 \wedge \dots \wedge C_n$ . Suppose  $v$  satisfies all the  $C_i$  constraints:
  - $t = u[v]$  is satisfied iff  $t[v]$  is the same tree as  $u[v]$ .
  - $t \neq u[v]$  is satisfied iff  $t = u[v]$  is not satisfied.
  - $r_H(t)[v]$  is satisfied iff  $t[v]$  is a tree of type  $r_H$ .

Then type  $r_A$  includes the tree  $e[v]$ .

- Types are defined by the minimum set satisfying the above.

The language for a tree grammar is the set of trees of a particular type.

**Definition 4** (*Language( $G$ )*) Let  $G = \langle N, L, T, V, S, P \rangle$  be a Tree Grammar. The language of  $G$  is the set of all trees of type  $S$ .

A solution to an Object-Oriented Constraint Satisfaction Problem for a particular tree grammar  $G$  is any tree in  $Language(G)$ . This departs from the typical definition only syntactically. Any set of variables with associated domains can be construed as the components of a new object type, which when made the start symbol admits a solution exactly when the set of variables admits a solution.

**Example 1** Suppose in the context of utility computing, we were interested in modeling computing resources for the purpose of constructing e-Commerce sites. Computers,  $C$ , are built out of an architecture,  $A$ , some amount of memory,  $M$ , and a hard drive  $H$ . We can model computers with the following production rules.

$$\begin{aligned} r_C(f_C(x, y, z)) &\Leftarrow r_A(x) \wedge r_M(y) \wedge r_H(z) \\ r_A(x86), r_A(sparc) \\ r_M(500MB), r_M(1GB) \\ r_H(40GB), r_H(80GB), r_H(160GB) \end{aligned}$$

Servers,  $S$ , are built out of a computer, an operating system,  $O$ , and an IP address,  $I$ . The WinNT operating system requires an x86 architecture, and Solaris requires SPARC.

$$\begin{aligned} r_S(f_S(f_C(x86, x, y), winnt, z)) &\Leftarrow \\ r_M(x) \wedge r_H(y) \wedge r_I(z) \\ r_S(f_S(f_C(sparc, x, y), solaris, z)) &\Leftarrow \\ r_M(x) \wedge r_H(y) \wedge r_I(z) \end{aligned}$$

For simplicity, we will use the natural numbers to represent IP addresses. The natural numbers,  $N$ , are represented as the height of a tree, i.e. encoded using the successor function.

$$\begin{aligned} r_N(0) \\ r_N(f_N(x)) &\Leftarrow r_N(x) \end{aligned}$$

Then the IP addresses can be defined as the set of natural numbers:  $r_I(x) \Leftarrow r_N(x)$ .

An e-Commerce site is built out of a network connection and a list of servers,  $L$ , each of which has a unique IP address. The empty list of servers is represented by  $nil$ , and the object  $f_L(x, y)$  is a list of servers if  $x$  is a server and  $y$  is a list of servers.

$$\begin{aligned} r_L(nil) \\ r_L(f_L(x, y)) &\Leftarrow r_S(x) \wedge r_L(y) \end{aligned}$$

Requiring that the list of servers all have unique IP addresses can be accomplished in a similar way. Every server in a list with zero or one elements has a unique IP address. If the first server in a list has a different IP address than every other server in the list, and the rest of the list has unique IP addresses, then so does the entire list.

$$\begin{aligned} r_D(nil) \\ r_D(f_L(x, nil)) \\ r_D(f_L(f_S(x, y, z), f_L(f_S(u, v, w), t)))) &\Leftarrow \\ z \neq w \wedge r_D(f_L(f_S(x, y, z), t)) \\ \wedge r_D(f_L(f_S(u, v, w), t)) \end{aligned}$$

The definition for an e-Commerce site is then straightforward—a network connection,  $W$ , and a list of servers that is also of type  $r_D$ .

$$r_E(f_E(x, y)) \Leftarrow r_W(x) \wedge r_L(y) \wedge r_D(y) \square$$

## Solving OOCSPs

In the next two sections we will be investigating various syntactic classes of Tree Grammars. Those classes are defined by the kinds of constraints allowed in the composition rules.

All the classes we consider include basic type constraints, e.g.  $r_A(f_A(x)) \Leftarrow r_B(x)$ . We will use the abbreviation  $TG$  to represent the class of tree grammars that includes intersection rules and composition rules with only type constraints. If  $TG$  is subscripted with  $=$ , the composition rules are allowed to include equality constraints as well. Likewise distinction and constructor constraints will be indicated by subscripting  $TG$  with  $\neq$  and  $c$ , respectively.

This section gives algorithms for solving OOCSPs for inputs  $TG_=$  and  $TG_{\neq}$  (under some restrictions).

As mentioned earlier, the inspiration for Tree Grammars was the CFG formalism. The results in this section rely on a feature of CFG composition rules. Every CFG composition rule for type  $r_A$  ensures that the root node of the treetop is semantically linked to  $r_A$ . That is, every CFG composition rule is of the following form.

$$r_A(f_A(\dots)) \Leftarrow \dots$$

We will call the class of composition rules that include this semantic connection between types and nodes the *CFG composition rules*. We will only examine this subset of composition rules in this section.

The composition rules in  $TG_=$  tree grammars are thus of the following form.

$$r_A(f_A(t_1, \dots, t_n)) \Leftarrow r_{B_1}(e_1) \wedge \dots \wedge r_{B_n}(e_m) \\ \wedge u_1 = v_1 \wedge \dots \wedge u_k = v_k$$

Each  $e_i$  is either a variable or a leaf, each  $t_i$  is a tree top, and each  $u_i, v_i$  is either a variable or a leaf.

The equality constraints can be removed by repeating the following cycle until no equality constraints remain. Replace variables by what they are equal to. Then remove tautologous equality constraints and remove rules containing unsatisfiable equality constraints. For example, if at some point the constraint  $a = b$  appears where  $a$  and  $b$  are leaves, remove the rule containing the constraint since it will never be satisfied. If the constraint  $a = a$  appears, remove just the constraint since it is always satisfied. At the end of this process the composition rules remaining have no equality constraints.

Consider any of the type constraints  $r_{B_i}(e_i)$ . If  $e_i$  is a leaf, we can easily check whether  $r_{B_i}$  is satisfied by first computing the types for all leaves in a bottom-up fashion (Ullman 1989). To save a little work later, we will compute all the types for all leaves and tree tops in simple typing rules and then add these results as simple typing rules to the set of production rules. For type constraints on leaves, if the constraint is satisfied, remove it; otherwise, remove the entire rule.

After removing the type constraints on leaves, the composition rules are of the form where two or more of the  $x_i$  may actually be the same variable. Sets of type constraints on the same variable implicitly produce an intersection. We can remove these intersection constraints within composition rules and handle them at the same time we handle the other intersection rules. For any set of type constraints  $r_{B_1}(x), \dots, r_{B_n}(x)$  on the variable  $x$ , introduce a new type  $r_B$  and replace  $r_{B_1}(x), \dots, r_{B_n}(x)$  with  $r_B(x)$

in the composition rule. Then include the intersection rule  $r_B(x) \Leftarrow r_{B_1}(x) \wedge \dots \wedge r_{B_n}(x)$ .

For example, the composition rule

$$r_A(f_A(x, y)) \Leftarrow r_B(x) \wedge r_C(y) \wedge r_D(y) \wedge r_E(y)$$

can be replaced by

$$r_A(f_A(x, y)) \Leftarrow r_B(x) \wedge r_F(y) \\ r_F(y) \Leftarrow r_C(y) \wedge r_D(y) \wedge r_E(y)$$

To remove the intersection rules, we will rely on the fact that the composition rules are CFG composition rules. Every tree of type  $r_A$  produced by such a composition rule is of the form  $f_A(\dots)$ . With this in mind, it is easy to see that the intersection of two distinct types is empty (except for tree tops) unless there is a subtyping rule relating the two types.

Continuing the example, the intersection rule

$$r_F(x) \Leftarrow r_C(x) \wedge r_D(x) \wedge r_E(x)$$

can only add complex trees to the  $r_F$  type if there are subtyping rules that give  $r_C, r_D,$  and  $r_E$  common subtypes. The following rules ensure they have the common subtype  $r_H$ .

$$r_C(x) \Leftarrow r_H(x) \\ r_D(x) \Leftarrow r_H(x) \\ r_E(x) \Leftarrow r_I(x) \\ r_I(x) \Leftarrow r_H(x)$$

Suppose that for the intersection rule  $r_A(x) \Leftarrow r_{B_1}(x) \wedge \dots \wedge r_{B_n}(x)$ , the types  $r_{B_1}, \dots, r_{B_n}$  share the subtypes  $r_{C_1}, \dots, r_{C_k}$ . Then the intersection rule can be replaced by a set of subtyping rules:

$$r_A(x) \Leftarrow r_{C_1}(x) \\ \vdots \\ r_A(x) \Leftarrow r_{C_k}(x)$$

Note that because we have already computed all the types for all the tree tops and leaves and stored that information as simple typing rules, no types are lost by removing this intersection rule. Thus the intersection rules can be replaced by a set of subtyping rules.

Those subtyping rules can be removed by turning our CFG composition rules into non-CFG composition rules. First compute the transitive closure on the subtyping rules, e.g. the rules  $r_A(x) \Leftarrow r_B(x)$  and  $r_B(x) \Leftarrow r_C(x)$  yield the rule  $r_A(x) \Leftarrow r_C(x)$ .

Next replace each subtyping rule  $r_A(x) \Leftarrow r_B(x)$  with a new set of composition rules for  $r_A$ . For each composition rule  $r_B(f_B(\dots)) \Leftarrow \dots$ , insert the rule  $r_A(f_B(\dots)) \Leftarrow \dots$ . This transformation breaks the CFG-ness of the composition rules, but this property was only necessary for handling intersection.

To summarize, given a  $TG_=$  grammar with CFG composition rules, one can simplify it by (1) removing equality constraints, (2) removing the type constraints on leaves, (3) removing intersection within composition rules, and (4) removing the intersection rules. This results in a set of production rules all of the following form.

$$r_A(f_C(t_1, \dots, t_n)) \Leftarrow r_{B_1}(x_1) \wedge \dots \wedge r_{B_m}(x_m)$$

Constructing a tree out such rules can be accomplished using a variant of the well-known marking algorithm for determining emptiness in a CFG (Sipser 1996).

**Algorithm 1 (Marking)** Input:  $\langle N, L, T, V, S, P \rangle$ .

1. Remove equality, leaf type constraints, and intersection.
2. Number the rules and run the basic marking algorithm.
  - (a) For each rule for type  $r_A$  numbered  $k$  without any unmarked constraints in the body, use  $k$  to mark all the unmarked  $r_A$  type-constraints in the rule bodies.
  - (b) If no rule for  $S$  has been marked and progress has been made, goto step (a).
  - (c) If no rule for  $S$  has been marked, return NIL.
  - (d) Choose a marked rule for  $S$  and apply it to  $S$ . For each type constraint in the body, apply the production rule corresponding to the number that constraint is marked with. Repeat until no type constraints remain. If the resulting tree top includes no variables, return it. Otherwise, fill in every variable with any one of the leaves. Return the resulting tree.

**Example 2** As an example, consider the following set of production rules, with the start symbol is  $r_S$ .

1.  $r_S(f_S(x, y)) \Leftarrow r_A(x) \wedge r_B(x)$
2.  $r_A(a)$
3.  $r_B(f_B(x)) \Leftarrow r_C(x)$
4.  $r_C(f_C(d))$

Begin by marking rule (2) and (4) as they have no unmarked constraints. Also mark the type constraint  $r_A$  in rule (1) with a 2 and  $r_C$  in rule (3) with a 4. No  $r_S$  rule has been marked, so continue. Since every type constraint in rule (3) has been marked, mark rule (3) itself and then mark the  $r_B$  constraint in rule (1) with a 3. Again no rule for  $r_S$  has been marked, so repeat. This time, every constraint in rule (1) has been marked, which means the rule itself must be marked, after which the loop exits. This results in the following marked production rules.

- 1.<sup>M</sup>  $r_S(f_S(x, y)) \Leftarrow r_A^2(x) \wedge r_B^3(y)$
- 2.<sup>M</sup>  $r_A(a)$
- 3.<sup>M</sup>  $r_B(f_B(x)) \Leftarrow r_C^4(x)$
- 4.<sup>M</sup>  $r_C(f_C(d))$

Because a rule for  $r_S$  is marked, construct a tree by first applying rule (1).  $r_S(f_S(x, y)) \Leftarrow_1 r_A(x) \wedge r_B(x)$ . Then apply rule (2) to the  $r_A$  constraint, followed by rule (3) to the  $r_B$  constraint, and finally rule (4) to the  $r_C$  constraint that resulted from applying rule (3). This yields the tree  $f_S(f_A(a), f_B(f_C(d)))$ .  $\square$

**Theorem 1 (Correctness of Marking)** *Marking takes as input a tree grammar  $G = \langle N, L, T, V, S, P \rangle$  in  $TG_{=}$  with CFG composition rules. It outputs a tree in  $G$  of type  $S$  if one exists and NIL if one does not.*

Removing equality constraints is fairly trivial but handling distinction constraints requires a fundamental change to the algorithm. Step (2) of Marking can be replaced by

a restricted form of Ullman's bottom up evaluation (1989), where only a certain number of trees of each type are computed. Once this bound is reached, all the rules that produce a tree of that type are discarded. Eventually no new trees can be produced or a tree of the requested type has been found. For completeness, we will require every composition rule to be *fully safe*, i.e. every variable in the head must occur in the body, and every variable in the body must occur in the head.

**Algorithm 2 (Bottom-up)** Input:  $\langle N, L, T, V, S, P \rangle$

1. Remove equality, leaf type constraints, and intersection.
2. Run altered bottom-up evaluation.
  - (a)  $k = \max$  number of  $\neq$  constraints in any rule.
  - (b) Build a bin  $b_A$  for each type  $r_A$ .
  - (c) For each rule without a body  $r_A(t)$ , place  $t$  in  $b_A$ .
  - (d) If  $|b_A| \geq k$  remove the production rules for type  $r_A$ .
  - (e) Use bottom-up evaluation to produce new trees.
  - (f) If no new trees were produced, return NIL. If there is some tree in the bin for  $S$ , return it. Otherwise repeat the last two steps.
  - (g) Return NIL.

**Theorem 2 (Correctness of Bottom-up)** *Bottom-up takes as input a tree grammar  $G = \langle N, L, T, V, S, P \rangle$  in  $TG_{\neq}$  with CFG composition rules. If at step (2), the composition rules are fully safe, it outputs a tree in  $G$  of type  $S$  if one exists and NIL if one does not.*

## Undecidability

In this section, we explain how to encode Diophantine equations within  $TG_c$ . This result places an upper bound on the syntactic fragment of all tree grammars for which OOCSPs are decidable, but leaves a gap in our understanding between undecidability and the decidability results of the last section.

Let  $P(x_1, \dots, x_n)$  be an arbitrary polynomial with integral coefficients and positive, integral exponents over the variables  $x_1, \dots, x_n$ . It is well known that finding a solution to  $P(x_1, \dots, x_n) = 0$  where each  $x_i$  is a natural number is undecidable. We will show how to encode such a polynomial within  $TG_c$ ; there can be no algorithm solving all OOCSPs since if there were, this transformation could be applied, resulting in a decision procedure for these Diophantine problems.

The natural numbers are represented using the analog of the successor function,  $f_N$ , as demonstrated in a previous section. Addition ( $r_A$ ) and multiplication ( $r_M$ ) operate on these  $f_N$  trees.  $x + y = z$  implies  $(x + 1) + y = (z + 1)$ . Likewise,  $x * y = z$  implies  $(x + 1) * y = z + y$ .

$$\begin{aligned} r_A(f_A(0, y, y)) &\Leftarrow r_N(y) \\ r_A(f_A(f_N(x), y, f_N(z))) &\Leftarrow r_N(x) \wedge r_N(y) \wedge r_N(z) \\ r_A(f_A(x, y, z)) & \end{aligned}$$

$$\begin{aligned} r_M(f_M(0, y, 0)) &\Leftarrow r_N(y) \\ r_M(f_M(f_N(x), y, z)) &\Leftarrow r_N(x) \wedge r_N(y) \wedge r_N(z) \\ r_M(f_M(x, y, w)) &\wedge r_A(f_A(y, w, z)) \end{aligned}$$

Technically, we need both positive and negative numbers, with addition and multiplication operating on the integers;

that encoding is straightforward but tedious. In exponentiation ( $r_E$ ), the first argument is the exponent, the second the base, and the third the base raised to the exponent. The recursive portion of the definition is structurally identical to that of multiplication.

$$\begin{aligned} r_E(f_E(0, y, f_N(0))) \\ r_E(f_E(f_N(x), y, z)) &\Leftarrow r_N(x) \wedge r_N(y) \wedge r_N(z) \\ r_E(f_E(x, y, w)) &\wedge r_M(f_M(y, w, z)) \end{aligned}$$

Using this machinery, every polynomial can be encoded as a Tree Grammar.

**Example 3** Consider the monomial  $x^3y^2$ . We will build a new type,  $r_P$ , for trees with three children so that if  $x$  is the left child and  $y$  is the middle child, the right child is  $x^3y^2$ .

$$\begin{aligned} r_P(f_P(x, y, t)) &\Leftarrow r_N(x) \wedge r_N(y) \wedge r_N(t) \wedge \\ &r_E(f_E(f_N(f_N(f_N(0))), x, z)) \wedge \\ &r_E(f_E(f_N(f_N(0)), y, w)) \wedge \\ &r_M(f_M(z, w, t)) \end{aligned}$$

If this were the polynomial of interest, we could construct an OOCSP to solve the equation  $x^3y^2 = 0$  by introducing a new type  $r_S$ , and making it the start symbol.

$$r_S(f_S(x, y)) \Leftarrow r_N(x) \wedge r_N(y) \wedge r_P(f_P(x, y, 0)) \quad \square$$

The summation of monomials can be encoded in a similar fashion, as can setting that sum to zero. The full encoding requires nothing outside  $TG_c$ . In fact, it appears that limiting every composition rule to two constructors is sufficient for undecidability, but space restrictions do not allow us to explain further.

**Theorem 3 (Undecidability of  $TG_c$ )** *The class of OOCSPs that take as input  $TG_c$  is undecidable.*

## Discussion

OOCSPs include two types of rules: composition rules and intersection rules. Those where the composition rules are limited to type and equality constraints can be solved by a Marking algorithm similar to that for determining whether a CFG is empty. Distinction ( $\neq$ ) constraints require an algorithm that actually builds a small set of trees for each type. This Bottom-up evaluation is a generalization of marking; hence, it can be used for solving OOCSPs that include type constraints, equality constraints, and distinction constraints.

OOCSPs that allow two or more constructors per composition rule is sufficiently expressive to encode Diophantine equations, thus making the class formally undecidable. Addition can be written using one constructor constraint, but multiplication requires two. One topic for future work is to determine whether restricting composition rules to one constructor results in decidability. This type of OOCSP is likely to be decidable since Presburger arithmetic, which includes addition but not multiplication, is decidable but Peano arithmetic, which includes both is not.

Non-recursive OOCSPs are no more expressive than standard CSPs because they can always be flattened. Certain kinds of CSPs with infinite domains can be represented as

OOCSPs. The encoding for the natural numbers, for instance, has been illustrated in previous sections. But OOCSPs are strictly less expressive than the class of CSPs with infinite domains. For example, a CSP over the real numbers cannot be expressed as an OOCSP.

In a CSP with infinite domains, knowledge of those domains must be hard-coded into the CSP-solver. OOCSPs can be used to encode infinite domains using a finite set of rules, which greatly reduces the need for domain-specific solvers. But while domains themselves can be more compactly represented in OOCSPs, representing CSP constraints may require an exponential number of production rules because they are expressed in disjunctive-normal form.

Object-oriented constraint satisfaction was a term first coined by Paltrinieri(1994; 1995). His version is no more expressive than a CSP with finite domains, but because our version is in general undecidable, the OOCSPs in this paper are clearly more expressive. Dynamic Constraint Satisfaction Problems (Mittal & Falkenhainer 1990) allow the values of special variables to affect the number of regular variables in a solution, but those special variables must be recognized as such by the DCSP solver. Hierarchical constraint satisfaction allows one to define a hierarchy of constraints. Those with the highest precedence must be satisfied; the others represent a ranked set of preferences (Borning, Freeman-Benson, & Wilson 1992). Other related topics include OO programming languages that provide native constructs for solving CSPs (Caseau 1994; Roy & Pacht 1997) and natural language generation.

## References

- Borning, A.; Freeman-Benson, B. N.; and Wilson, M. 1992. Constraint hierarchies. *Lisp and Symbolic Computation* 5:223–270.
- Caseau, Y. 1994. Constraint satisfaction with an object-oriented knowledge representation language. *Applied Intelligence* 4(2):157–184.
- Hinrichs, T.; Love, N.; Petrie, C.; Ramshaw, L.; Sahai, A.; and Singhal, S. 2004. Using object-oriented constraint satisfaction for automated configuration generation. *DSOM*.
- Mittal, S., and Falkenhainer, B. 1990. Dynamic constraint satisfaction problems. *AAAI 90* 25–32.
- Paltrinieri, M. 1994. Some remarks on the design of constraint satisfaction problems. *Workshop on Principles and Practice of Constraint Programming* 299–311.
- Paltrinieri, M. 1995. A visual environment for constraint programming. *IEEE Symposium on Visual Languages*.
- Roy, P., and Pacht, F. 1997. Reifying constraint satisfaction in smalltalk. *Journal of Object-Oriented Programming* 10(4):43–51.
- Sipser, M. 1996. *Introduction to the Theory of Computation*. Brooks Cole.
- Ullman, J. 1989. *Principles of Database and Knowledge-Base Systems*. Computer Science Press.