SOS C--: a System for interpreting Operational Semantics of C-- programs

Olivier Ponsini, Carine Fédèle and Emmanuel Kounalis
Laboratoire I3S - UNSA - CNRS
2000, route des Lucioles, B.P. 121
O6 903 Sophia Antipolis, FRANCE
email: {ponsini, carine, kounalis}@i3s.unice.fr

ABSTRACT
This paper describes a system for automatically transforming programs written in a simple imperative language (called C--), into a set of first-order equations. This means that a set of first-order equations used to represent a C-- program already has a precise mathematical meaning; moreover, the standard techniques for mechanizing equational reasoning can be used for verifying properties of programs. This work shows that simple imperative programs can be seen as fully formalized logical systems, within which theorems can be proved. The system itself is formulated abstractly as a set of first-order rewrite rules. Then, it is proven to be terminating and confluent using the RRL system.

KEY WORDS
Program Verification, Rewriting, Equational Semantics

1. Introduction

The need to be able to reason about computer programs in a rigorous formal way is self evident. In order to increase confidence in code production, efforts should be focused on verifying that programs meet their requirements, that is, that they are sound with respect to their specification. In a previous work [1], FÉDÈLE and KOUNALIS introduced a theoretical framework for proving automatically properties of C-- programs, a simple imperative language. The idea was to translate source code into a set of first order equations expressing the program algebraic semantics. The use of *equational logic*, has some advantages over other, more complex logics:

1. it is very simple — the logic of substituting equals for equals;
2. many problems associated with equations, that are not decidable in more complex logics, are decidable in equational logic;
3. there are efficient algorithms for deciding many of these problems.

The general outline of our framework is shown in Fig. 1. Users write down the C-- code of a program; they

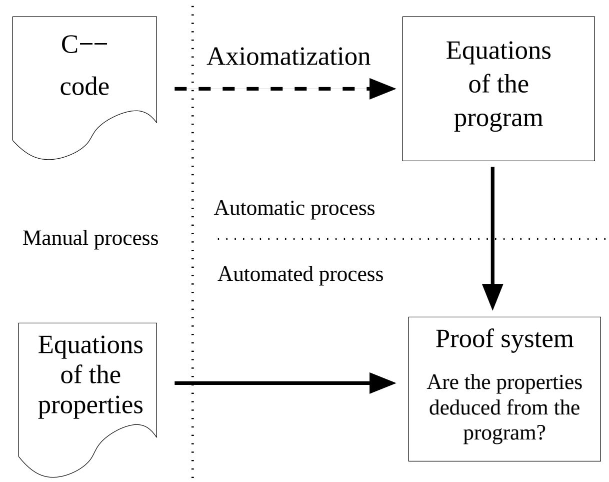

Figure 1. Proof process overview

also write the program specification as a set of properties expressed in equational logic.The source code is then transformed automatically, by the SOS C-- system, into a set of equations. The equations of the program can be seen as the axioms, and the properties to be proved as the conjectured theorems of the axioms. Therefore, the proof of these theorems from the axioms is equivalent to the proof that the program meets its specification. This last part may be done automatically using *theorem provers* able to do mathematical induction like NICE [2] or interactively using *proof checkers* like COQ [3].

This approach is conceptually different from other recent developments like COGITO [4] or SPECWARE [5], since these systems generate code from specifications. It also differs from [6] and [7] since they use *annotations* to prove the program behavior. Our approach is conceptually similar to the systems [8] and [9]. However, [9] can not treat programs with loops and [8] works in logics not easily amenable to automation.

In this paper, we extend [1] in various directions:

1. We give an abstract framework to the *program towards equation* process. In [1], there was only the basis of a rewrite system and no property had been proved on the system.

- In particular we formulate the SOS C-- system as a rewrite system. We refine the rules to enrich and be closer to the C-- semantics.
- We prove the rewrite system *completeness* (i.e. termination and confluence) using the RRL[1]. Roughly speaking, this kind of completeness means that every C-- program can be transformed into a unique equational program.
- We give a formal description of how equations are generated from environments.

2. In [1], no implementation had been done. We now assert that the axiomatization process can be automatized since we made an implementation in Java.

   - JavaCC[2], a parser and scanner generator, has been used for the term generation step.
   - We developed a Java version of a *generic* rewriting algorithm. The rewrite rules are loaded separately from a file so as to elaborate the rules with ease.
   - We worked out an algorithm to generate equations from environments.

In what follows, we first introduce some basic definitions and notations. Then, section 3 gives a general outline of the SOS C-- system and illustrates it with an example; the section uses the following scheme:

1. programs are terms;
2. terms produce environments;
3. environments generate equations.

2. Definitions

2.1 Rewrite Systems

We assume familiarity with the basic notions of equational logic and rewrite systems (see [11] for instance). Let $T(F, X)$ denote the set of terms built out of function symbols taken from the finite *vocabulary* F and a denumerable set X of *variables*. If t is a term and θ is a *substitution* of terms for variables in t, then $t\theta$ is an *instance* of t. An *equation* e is an element $T(F, X) \times T(F, X)$ and is written as $t = s$. A *rewrite system* $\mathcal{R}$ is a set of oriented equations $l \to r$, called *rewrite rules*. A rule is applied to a term t by finding a subterm s of t that is an instance of the left side l (i.e. $s = l\theta$) and replacing s with the corresponding instance ($r\theta$) of the rule's right side. One computes with $\mathcal{R}$ by repeatedly applying rules to rewrite (or reduce) an input term until a *normal form* (irreducible term) is obtained. Let A be a set of equations, in the case where A can be compiled into a *complete* (i.e. *terminating* and *confluent*) rewrite system $\mathcal{R}$, we can decide $t =_A s$ by testing for identity the $\mathcal{R}$-normal forms of t and s (i.e. $\mathrm{nf}(t) \stackrel{?}{=} \mathrm{nf}(s)$, where $\mathrm{nf}(t)$ (resp. $\mathrm{nf}(s)$) denotes the normal form of t (resp. s)).

2.2 The C-- Language

For our experiments we use a very simple imperative language. The C-- syntax is similar to the C one. The main features of the language are:

- assignment;
- control flow statements: *if* ... *else*, *while* and *return*;
- two predefined types: integers (*int*), and lists of integer (*list*);
- usual arithmetic operators;
- operators on lists: *getHead* which returns the first element of a list, *getQueue* which returns a copy of a list except for the first element, *NULL* which represents an empty list, and *cons* which inserts an element at the beginning of a list.

However, several common features in imperative languages are unavailable in C--: no user's defined types; no global variables; no pointers directly accessible — of course some are used in the predefined type *list*.

3. A General Outline of the SOS C-- System

The SOS C-- system can be represented as a rewrite system $\mathcal{R}$ over a first-order language $\mathcal{L}$ built out from a set of function symbols. These symbols are translations of the constructs of the source language (C--). For instance, the *assignment* statement, written $x = y$ in C--, is translated into $\mathrm{Assign}(x, y)$, where *Assign* belongs to the vocabulary of $\mathcal{L}$.

The system takes as input a *C-- program* and returns as output a *set of equations* semantically equivalent to the program: the result of the execution of the C-- program with input I is identical to the result (i.e. theorem) of the equational deduction, started with the very same input. The transformation process, called *C-- axiomatization*, is done in three steps and is carried out without any user interaction. Figure 2 shows the steps involved in the C-- axiomatization.

To illustrate these steps, we will use the *sorting program* of listing 1. This program is a C-- version of the insertion sort. Function *ins* takes an integer e and a sorted list L and returns a new sorted list which is a copy of L containing e. *ISort* takes a list L as argument and returns a sorted copy of L by inserting at the right position (call to function *ins*) the first element of L in the already sorted queue of L.

[1] Rewrite Rule Laboratory [10]. The proof is part of the full version of this paper.

[2] Java Compiler Compiler, Metamata.

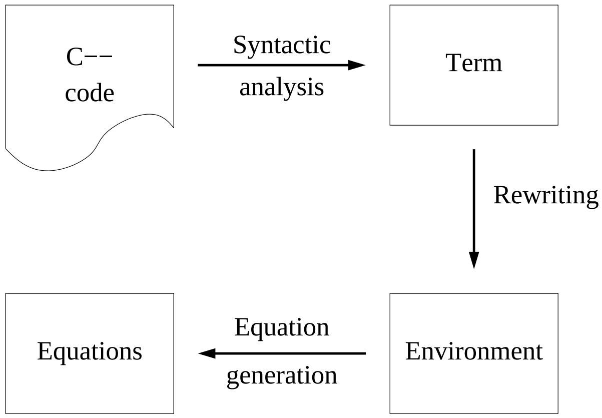

Figure 2. Axiomatization process overview

3.1 Programs are Terms

The first step consists in analyzing the functions of the source program P. The result of this syntactic analysis is a list of term T_P^f over $\mathcal{L}$; one term for each function f of P. Intuitively, a term is equivalent to a source function and suitable for rewriting.

Each C-- *function* is seen as a list of statements and an initial environment gathered in a *GE*[3] term. The initial environment is made up of the function formal parameters combine in a *Pair* term with *EP*[4] terms. An *EP* term denotes an effective parameter. Thus, formal parameters behave like local variables to which are assigned the effective parameters.

Variable declarations are considered as assignments and are therefore added to the list of statements of the function.

A *sequence* of statements is simulated by a list of terms.

Return statements engender a pair linking the name of the function and its return expression.

Expressions appearing as right value in an assignment or as value in a function call are left as is.

The other constructs of the C-- language are simply matched with equivalent terms of the rewrite system: the *if* statement, if(c) $s1$ else $s2$, is translated into If$(c, s1, s2)$; the *while* statement, $while(c)$ s, is translated into While$(while_number, c, s)$.

For instance, the term which corresponds to the *ISort* function of the *sorting program* is:

GE({ Assign(*ret*, *NULL*),
If($L = NULL$,
{Assign(*ret*, *NULL*)},
{Assign(*ret*, *ins*(*getHead*(L),
ISort(*getQueue*(L))))}),
Return(*ISort*(L), *ret*) }, { Pair(L, EP(L)) }) .

[3]*GE* stands for Generate Environment.

[4]*EP*(x) stands for Effective Parameter x. It is the value given to the function parameter x when the function is called.

Listing 1. Sorting program.

```
list ins(int e, list L) {
list ret=NULL;
if (L==NULL) ret=cons(e, NULL);
  else if (e<=getHead(L)) ret=cons(e, L);
         else { ret=ins(e, getQueue(L));
                ret=cons(getHead(L), ret); }
return ret; }

list ISort( list L) {
list ret=NULL;
if (L==NULL) ret=NULL;
  else ret=ins(getHead(L), ISort(getQueue(L)));
return ret; }
```

A *GE* term contains two elements. The first one represents the sequence of instructions of the source function:

- the declaration of *ret* gives the first *Assign* term;
- then comes the *If* term with the condition and its two lists of statements, one for each alternative;
- and finally the *Return* term.

The second one represents the function's initial environment. It is a list of *Pair* terms. A *Pair* associates a variable and a value. Thus, the initial environment is the value of the function's variables after the call but before any instruction is evaluated.

Likewise, the term which corresponds to the *ins* function is:

GE({Assign(*ret*, *NULL*) ,
If($L = NULL$,
{Assign(*ret*, *cons*(e, *NULL*))},
{If($e \leq getHead(L)$,
{Assign(*ret*, *cons*(e, L))},
{Assign(*ret*, *ins*(e, *getQueue*(L))),
Assign(*ret*, *cons*(*getHead*(L), *ret*))}) }
Return(*ins*(e, L), *ret*) },
{Pair(L, EP(L)) , Pair(e, EP(e)) }) .

3.2 Terms produce Environments

At the second step, the system transforms a term T_P^f into an environment $E_{T_P^f}$. Intuitively, this environment contains all information about the variables of function f and their corresponding expressions (the evaluation of which yields the value of f in an execution of P).

3.2.1 Description

To obtain this environment, each term T_P^f is normalized according to the rewrite rules[5] of $\mathcal{R}$. The rules are divided

[5]The complete set of rules can be found in the full version of this paper.

into two classes. The first class contains the rules that represent the C-- language operational semantics. The second class contains rules for updating environments — mainly syntax manipulating rules for list manipulation.

Function statements are executed in an order which depends on the control flow statements and their associated conditions. These different possible orderings constitute the execution paths of a function. The environment produced by rewriting represents the distinct execution paths of a function, along with their associated conditions and the final expression of variables and function. The state of the variables is represented by a list of *Pair* terms. A *Branch* term associates a condition to a variables state. The paths are enclosed in *Choice* terms.

For instance, the environment of the *ISort* function of listing 1 is:

Choice(
  [Branch($L = NULL$,
    { Pair(L, L), Pair(*ret*, *NULL*),
      Pair(*ISort*(L), *NULL*) })],
  [Branch($L \neq NULL$,
    { Pair(L, L), Pair(*ret*, *ins*(*getHead*(L),
        ISort(*getQueue*(L)))),
      Pair(*ISort*(L), *ins*(*getHead*(L),
        ISort(*getQueue*(L)))) })]) .

Function *ISort* comprises one *if* statement, so we find in the environment a *Choice* term composed of two *Branch* terms. These latter terms partition the statements of the function between those which are executed when the condition $L = NULL$ is true and those which are executed when this same condition is false. In each *Branch* term there is a list of *Pair* terms which represents the state of the variables at the end of an "abstract" execution of the *ISort* function. For instance, in the case where $L = NULL$, Pair(L, L) means that L is not modified by the function; Pair(*ret*, *NULL*) means that the value of variable *ret* is *NULL*; the *Pair* term containing the function name, Pair(*ISort*(L), *NULL*) means that the function return value is *NULL*.

Likewise, the environment of the *ins* function is:

Choice(
  [Branch($L = NULL$,
    { Pair(L, L), Pair(e, e), Pair(*ret*, *cons*(e, *NULL*)),
      Pair(*ins*(e, L), *cons*(e, *NULL*)) })],
  [Choice(
    [Branch($L \neq NULL$ and $e \leq getHead(L)$,
      { Pair(L, L), Pair(e, e), Pair(*ret*, *cons*(e, L)),
        Pair(*ins*(e, L), *cons*(e, L)) })],
    [Branch($L \neq NULL$ and $e > getHead(L)$,
      { Pair(L, L), Pair(e, e),
        Pair(*ret*, *cons*(*getHead*(L), *ins*(e, *getQueue*(L)))),
        Pair(*ins*(e, L), *cons*(*getHead*(L),
          ins(e, *getQueue*(L)))) })])]) .

3.2.2 *While* Statements

We now turn our attention to the iterative construct *while*. This section gives an insight into how we handle *while* statements, and we then formalize it in sections 3.2.3 and 3.3.

The semantics of *while* statements is quite specific. Indeed, each loop is considered as a family of separate recursive functions with their own parameters and body. The idea is that a loop is a function which calls itself recursively with the value of the variables modified accordingly to the statements of the loop body. Such a loop function is defined for each variable modified in a loop body. Then, in the function containing the loop, the value of a variable modified in the loop body is the result of a call to the specific loop function. Consequently, when a loop is encountered, the environment is modified as follows:

- A new LT[6] term containing all the information needed to generate the loop functions is created. The information is the loop number, the exit condition, the statements of the loop body and the list of all the variables. This Loop Term will be used at the third step to generate a family of equations (see section 3.3).
- Each variable modified in the loop body is assigned a call to the corresponding loop function. This function takes as argument the current state of the variables.

The following example shows how *while* statements are handled. Let us suppose a C-- function declares three variables — x, y, z — two of which are modified in a loop body, like in listing 2. During rewriting of the term corresponding to function f, the following Loop Term is created:

Listing 2. A loop.

```
int f () {
int x,y,z;

x=1; y=2; z=3;

while(y>0) {
  x=x+z;
  y=y−1;
  }
 . . .
}
```

$$\text{LT}(1, y > 0, \text{GE}(statements,\ initial\ environment),\\ (x, y, z))\ .$$

The *statements* are the two assignments modifying x and y. The *initial environment* is the list of pairs $(x, \text{EP}(x))$, $(y, \text{EP}(y))$ and $(z, \text{EP}(z))$. The term GE means that a new

[6] Loop Term.

environment will be evaluated for the loop body. At the third step of the process, this will lead to the definition of two functions, LOOP^1_x and LOOP^1_y, one for each variable modified in the loop.

$$\begin{cases} \text{If } y > 0 \text{ then} \\ \quad \text{LOOP}^1_x(x,y,z) = \text{LOOP}^1_x(x+z, y-1, z) \\ \text{If not}(y > 0) \text{ then} \\ \quad \text{LOOP}^1_x(x,y,z) = x \end{cases}$$

$$\begin{cases} \text{If } y > 0 \text{ then} \\ \quad \text{LOOP}^1_y(x,y,z) = \text{LOOP}^1_y(x+z, y-1, z) \\ \text{If not}(y > 0) \text{ then} \\ \quad \text{LOOP}^1_y(x,y,z) = y \end{cases}$$

In addition, the pairs
$\text{Pair}(x, \text{Loop}^1_x(current\ state\ of\ variables))$
and
$\text{Pair}(y, \text{Loop}^1_y(current\ state\ of\ variables))$,
are inserted in the environment of function *f* to reflect the new state of these variables. The *current state of variables* refers to the state of the variables just before the *while* statement. This is just like replacing the loop in function *f* by the function calls: $x = \text{LOOP}^1_x(1,2,3)$ and $y = \text{LOOP}^1_y(1,2,3)$.

3.2.3 Rules

As we said formerly, rules are divided into two classes. Among the rules describing the language operational semantics, we find:

- GE rules: they are used to translate the behavior of the *sequence* instruction;
- $Comp$ rules: they are used to evaluate a new statement in the current environment;
  - Rules for the *assignment* and *return* statements. These rules add a new pair *Pair*(variable, value) to the environment or modify an existing pair.
  - Rules for the *if* statement. These rules aim at dividing the environment into two parts through a $Choice$ term. Each part is included in a $Branch$ term and contains an *if* alternative depending on whether the condition is valid or not.
  - Rules to define the *while* statement. These rules create an LT term and add a new pair to the environment for each variable modified in the loop body.
- two more $Comp$ rules to tell that the statements following an *if* statement must be executed whatever alternative is chosen;
- $Branch$ rules to group together two successive *if* statements by merging the conditions.

Among the rules updating an environment, we find:

- $Merge\_env$ and $Merge\_L\_var$ to merge two lists;
- $Insert\_pair$ and $Insert\_var$ to add a pair or a variable to a list;
- $GLOV$, $GLOMV$ and $GLOE$ to run through a list and build a new list by extracting, respectively, variables, modified variables or expressions from the initial list.

3.3 Environments generate Equations

At the last step, a set of equations is generated from each $E_{T^f_P}$. This set of equations defines the algebraic semantics of P. In the example of the *sorting program*, we obtain:

$$\boxed{\begin{array}{l} L = NULL \Rightarrow \text{ins}(e,L) = \text{cons}(e, NULL) \\ L \neq NULL \text{ and } e \leq \text{getHead}(L) \Rightarrow \\ \quad \text{ins}(e,L) = \text{cons}(e,L) \\ L \neq NULL \text{ and } e > \text{getHead}(L) \Rightarrow \\ \quad \text{ins}(e,L) = \text{cons}(\text{getHead}(L), \text{ins}(e, \text{getQueue}(L))) \\ L = NULL \Rightarrow \text{ISort}(L) = NULL \\ L \neq NULL \Rightarrow \\ \quad \text{ISort}(L) = \text{ins}(\text{getHead}(L), \text{ISort}(\text{getQueue}(L))) \end{array}}$$

Only a few elements in an environment will generate equations: these are the equation generators. The third and final step of the axiomatization process refines environments, extracts equation generators from environments and generates the corresponding equations.

The equation generators are:

- *lists of pairs*. They represent the state of the variables at the end of the computation. But, only the function return value is of interest, therefore, only the pair containing the function name will generate an equation.

  `Generator` : $\text{Pair}(\ldots) \cdot \ldots \cdot \text{Pair}(func\_name, expression)$
  `Equation` : $func\_name = expression$

- *Branch* terms. They appear because of an *if* statement and represent an alternative. They link a condition and a list of pairs. Again, only the pair with the function name is of interest. Each $Branch$ term generates one conditional equations.

  `Generator` : $\text{Branch}(condition, \text{Pair}(\ldots) \cdot \ldots \cdot \text{Pair}(func\_name, exp))$
  `Equation` : $condition = True \Rightarrow func\_name = exp$

- *LT* terms. They generate a family of conditional equations that defines recursively the loop functions — one loop function for each modified variable in the loop body. Two equations are needed, one for the recursive call — with the variables state modified according to the loop body — and one for the exit case which gives the result of the loop function, that is the current value of the considered modified variable.

$$\texttt{Generator} \; : \; \mathrm{LT}(num, cond, \mathrm{Pair}(v_1, e_1) \cdot \ldots \cdot \mathrm{Pair}(v_n, e_n), \{v_1, \ldots, v_n\})$$

$$\texttt{Equation} \; :$$

$$\bigcup_{1 \leq i \leq m} \left\{ \begin{array}{l} cond = True \Rightarrow \\ \quad \mathrm{LOOP}^{num}_{v_{\mathrm{mod}_i}}(v_1, \ldots, v_n) = \\ \qquad \mathrm{LOOP}^{num}_{v_{\mathrm{mod}_i}}(e_1, \ldots, e_n) \; , \\ cond = False \Rightarrow \\ \quad \mathrm{LOOP}^{num}_{v_{\mathrm{mod}_i}}(v_1, \ldots, v_n) = v_{\mathrm{mod}_i} \end{array} \right\}$$

Here, $v_{\mathrm{mod}_1}, \ldots, v_{\mathrm{mod}_m}$ are the variables modified in the loop body and $v_1, \ldots, v_n$ are all the variables appearing in the C-- function. A variable v is known to have been modified when its value differs from $\mathrm{EP}(v)$ which is the value assigned to it before getting in the loop.

4. Conclusion

In this paper we have discussed a system to automatically obtain an equivalent equational formulation of a C-- program from source code. The process leading to the equations requires three steps. The central point of the discussed method is the generation of an environment by means of a rewrite system which implements the operational semantics of the C-- language. The first stage consists in building a term suitable for rewriting through the syntactic analysis of the program code. The last stage consists in translating the environment into equations. An implementation of this system has been carried out in Java. A lot of work has still to be done. Future work includes:

- Adding functionalities to the C-- language in order to come closer to real imperative languages. Indeed, despite the use of *assignment* and *while* constructs, we are still very close to a functional programming style.
- implementing interfaces towards proof systems, that is, providing the equations in the specific proof system syntax;
- experimenting on a larger scale proving properties from the equations in proof systems. This in order to identify a class of properties and programs that can be proven sound using our system.

An extended version of this paper and a software implementation of the SOS C-- system can be found at the following address: http://www.i3s.unice.fr/˜ponsini.

References

[1] C. Fédèle and E. Kounalis, Automatic proofs of properties of simple C-- modules, Proc. *14th IEEE Conf. on Automated Software Engineering*, Cocoa Beach, USA, 1999, 283–286.

[2] E. Kounalis and P. Urso, Generalization discovery for proofs by induction in conditional theories, Proc. *12th FLAIRS Conf.*, Orlando, USA, 1999, 250–256.

[3] B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, C. Mu noz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner, The Coq Proof Assistant Reference Manual – Version V6.1, Technical Report 0203, INRIA, 1997.

[4] O. Traynor, D. Hazel, P. Kearney, A. Martin, R. Nickson, and L. Wildman, The Cogito development system, Proc. *6th Conf. on Algebraic Methodology and Software Technology*, Sidney, Australia, 1997, 586–591.

[5] R. Juellig, Y. Srinivas, and J. Liu, SPECWARE: An advanced environment for the formal development of complex software systems, Proc. *5th Conf. on Algebraic Methodology and Software Technology*, Munich, Germany, 1996, 551.

[6] J-C. Filliâtre, Proof of imperative programs in type theory, Proc. *2nd Workshop on Types for Proofs and Programs*, Kloster Irsee, Germany, 1998, 78–92.

[7] S. Antoy and J. Gannon, Using Term Rewriting to Verify Software, *IEEE Transactions on Software Engineering*, 20(4), 1994, 259–274.

[8] M. Ward, Abstracting a specification from code, *Journal of Software Maintenance: Research and Practice*, 5(2), 1993, 101–122.

[9] D. Schweizer and C. Denzler, Verifying the specification–to–code correspondence for abstract data types, Proc. *6th Conf. on Dependable Computing for Critical Applications*, Garmisch-Partenkirchen, Germany, 1997.

[10] D. Kapur and H. Zhang. *RRL : Rewrite Rule Laboratory*, 1989.

[11] F. Baader and T. Nipkow, *Term Rewriting and All That* (Cambridge: Cambridge University Press, 1998).