Logic + Control revisited: an abstract interpreter for Gödel programs

Egon Börger

Dip. di Informatica
C.so Italia 40, I-56100 PISA
boerger@di.unipi.it

Elvinia Riccobene

Dip. di Matematica
V.le Andrea Doria 6, I-95125 CATANIA
riccobene@mathct.cineca.it

Abstract

[1]We develop a simple interpreter for programs of the new logic programming language Gödel. The definition provides a clean interface between logical and control components for execution of Gödel programs. The construction is given in abstract terms which cover the general logic programming paradigm and allow for concurrency.

The formalization directly reflects the intuitive procedural understanding of programs, but is formulated at the level of abstract search spaces and proceeds in a modular fashion. This combination of procedural and abstract features, made possible by use of Gurevich's notion of *evolving algebras*, provides a tool for mathematical—machine and proof system independent—description and analysis of design decisions for logic programming languages; it also lays the ground for provably correct stepwise refinements, through a hierarchy of specifications at lower levels, down to implementations.

1 Introduction

J.W.Lloyd and P.Hill (1992) have proposed the new general–purpose logic programming language *Gödel*, with particular emphasis on improving the declarative semantics compared with Prolog. *Gödel* has a type system which is based on many-sorted logic with parametric polymorphism, a module system and infinite precision integers, rationals and floating-point numbers; it can solve constraints over finite domains of integers as well as linear rational constraints; it supports processing of finite sets as well as meta–logical facilities (created in order to provide support for meta–programs for

[1]in: G.Levi (Ed.), *Advances in Logic Programming Theory*, Oxford University Press, 1994

analysis, transformation, compilation, verification and debugging of programs).

We do not analyse any of these features but concentrate our attention in this paper on the special role which is played by three forms of non determinism in the logic programming paradigm (see below) that characterize the top level definition of the procedural semantics of Gödel programs. From the implementation point of view, non determinism reflects the intended flexibility of Gödel's computational rule and the desire "to give implementors the option" of not relying upon (some generalization of) SLDNF-resolution but "of using other theorem proving techniques to implement the language, e.g. ones which avoid floundering or are more complete" (Lloyd 1992).

We develop here, by stepwise refinement, a mathematically precise but simple procedural formalisation of the language which describes the full (control flow) behaviour of Gödel programs on the basis of *abstract*—machine and resolution independent—*search spaces*. In particular our specification provides a rigorous basis for an equivalence proof between declarative and procedural semantics of pure Gödel programs. We exemplify the model for (an SLDNF like) resolution as basic computation mechanism; we do it in a modular way and exhibit explicitly the interface where one can

- adapt the semantics to possible future changes in the design of the language,
- refine it in terms of particular proof systems or lower level specifications which are driven by consideration of execution efficiency and similar implementation issues.

Our search spaces are reminiscent of *Prolog Tree Algebras*, which appear in hybrid stack oriented form in (Börger 1990b) and have been defined in (Börger and Rosenzweig 1991a) and used in (Börger and Rosenzweig 1993a) as basis for a formal model of Prolog. The abstract search spaces have been obtained from Prolog tree algebras by a further abstraction step, introduced in order to directly reflect the desired flexible character of Gödel's computational rule.

This rule is indeed non deterministic

- in choosing where the next deduction step takes place—thus abstracting in particular from Prolog's depth–first strategy,
- in selecting, out of a conjunction of goals, the literal to be computed—thus abstracting from Prolog's left–to–right strategy ,
- in selecting a clause to reduce the current call—thus abstracting from Prolog's sequential strategy and from any scheme for indexing, switching, last call optimization etc.

Gurevich's concept of *external* functions (Gurevich 1991) in *evolving algebras* (Gurevich 1988) gave us the technical instrument to express in an ex-

plicit and transparent way the role of these three (pairwise orthogonal) non deterministic control components for the semantics of Gödel (and in general logic) programs. The effect of control by delaying, pruning, conditionals, negation, etc. on program execution is expressed by abstract conditions, which can be viewed as directives for the implementation of efficient but semantically correct search strategies. In this way a highly abstract mathematical framework for logic programming systems is laid down in which one can conduct a comparative study of the different answers which have been given, by well known logic programming languages and their implementations, to the *Gretchenfrage* of logic *programming*—namely to explicitly and exactly determine the real relation between "logic" and "control", between declarative (high-level) specification and procedural ("low"-level) implementation of (not only pure) logic programs.

On this basis we give a simple description of the high-level procedural semantics of Gödel programs, including, besides a full treatment of user-defined predicates, as characteristic built-in control flow features the *pruning* operator (which relativises Prolog's *cut* operator and generalises the *commit* operator of the concurrent logic programming languages), *negation* and *conditionals* together with their *delay* properties. Other system features in Gödel (for constraints, number manipulation etc.) can be dealt with in this model by adapting the methodology developed in (Börger 1990b, 1992; Börger and Rosenzweig 1993a; Börger and Schmitt 1991) for the formalization of their (sometimes quite different) Prolog homonyms; see also (Beierle and Börger 1992) for a treatment of polymorphic types for an extension of Prolog. We indicate the effect of depth-first search on this abstract specification and briefly discuss how to obtain from it a concurrent model of Gödel.

2 Evolving Algebras

We expect from the reader basic knowledge of logic programming (Apt 1990; Lloyd 1987) or Gödel (Hill and Lloyd 1992) and rudimentary knowledge of the language of first order logic.

Our model comes in the form of *evolving algebras*. This concept has been introduced by Gurevich (1988) and has since then been applied successfully for the specification of languages covering all the major programming paradigms; see (Gurevich and Morris 1988; Börger 1990a,b,1992; Gurevich and Moss 1990; Börger and Schmitt 1991; Gurevich and Huggins 1993; Börger and Rosenzweig 1993a; Beierle and Börger 1992; Börger and Riccobene 1993; Börger and Riccobene 1992; Börger *et al.* 1993; Blakley 1992; Gottlob *et al.* 1991). Although evolving algebras have a rigorous mathematical foundation as transition systems over first order structures (see Gurevich 1991; Glavan and Rosenzweig 1993), we make this paper independent by listing in this section some definitions which will enable the

reader to grasp our rules as 'pseudocode over abstract data'.

For a general and extensive discussion of the evolving algebra methodology for specification of (the semantics of) programming languages and systems, in particular in comparison to other well known approaches in the literature (denotational, algebraic, axiomatic, ...), we refer the reader to the introduction to (Boerger and Rosenzweig 1993a).

The abstract data come as elements of (not further analysed) sets (domains, *universes*). The operations allowed on universes will be represented by partial *functions*. The setup is allowed to *evolve* in time, by executing *function updates* of form

$$f(t_1, \cdots, t_n) := t$$

whose execution is to be understood as *changing* (or defining, if there was none) the value of function f at given arguments.

The 0-ary functions are like variable quantities of ancient mathematics or *variable* of programming; this is the reason why we abstain from calling them *constants*. Functions which do not appear as outer function f on the left hand side of function updates have been called *external* functions in (Gurevich 1991). They are not changed by the rule system, but they might nevertheless be (supposed to be) subject to change due to the activity of some external agent; in this case they are called *dynamic* external functions. In this paper we will make crucial use of such dynamic external functions.

For a natural high-level formalization of Gödel's pruning operator we allow also parameterized and guarded function updates of form

$$U(\vec{j}) \text{ or } \forall j(guard(\vec{j})) : U(\vec{j})$$

where U is a function update in which parameters $\vec{j}$ may occur, meaning that $U(\vec{j})$ is executed for all values of parameters (which satisfy $guard(\vec{j})$). (Typically j will be used to range over nodes on a segment of a path in a tree.) Note that such updates are not considered in the original definition given by (Gurevich 1988).

We shall also allow some of the universes (typically initially empty) to *grow* in time, by executing updates of form

Extend U **by** $t_1, \ldots, t_n$ **with** *updates* **EndExtend**

where *updates* may (and should) depend on t_i's, setting the values of some functions on the *newly created* elements t_i of U.

The precise way our "abstract machine" (evolving algebras) may evolve in time will be determined by a finite set of *transition rules* of form

If *condition* **then** *updates*

where *condition* is a (usually boolean) expression (guard), the truth of which triggers *simultaneous* execution of all updates listed in *updates*. Simultaneous execution helps us to avoid coding, for example to interchange

two values. Note that since functions may be partial, equality in the guards is to be interpreted in the usual sense of 'partial algebras', as implying that both arguments are defined; for more details see (Gurevich 1991; Glavan and Rosenzweig 1993).

In applications an evolving algebra usually comes together with a set of *integrity constraints*, i.e. extralogical axioms and/or rules of inference, specifying the intended domains. Indeed we are usually interested only in states which are reachble from some designated *initial states*, specified using any formal methods, such as those of algebraic specification.

The forms obviously reducible to the above basic syntax, which we shall freely use as abbreviations, are **let** and **if then else**. We shall assume that we have the standard mathematical universes of booleans, integers, lists of whatever, etc (as well as the standard operations on them) at our disposal without further mention.

For the sake of exposition the model we are going to construct specifies the behaviour of Gödel programs and goals such that the module structure is flattened and, when all commits are removed from the body of each statement, what remains is a normal program and a normal goal (which may contain the conditional construct). The additional first-order features—other connectives and quantifiers, which are allowed in Gödel statements—can easily be described as built-in predicates (using the technique presented here for negation and conditionals) or by preprocessing (which reduces to normal clause form, see (Lloyd 1987) and are skipped here for space reasons.

For simplicity we assume programs and queries to be well (static) typechecked.[2]

3 Signature of Gödel Algebras

This and the next section explain the basic data types (domains and functions) which are used in the transition rules of section 5.

A Gödel computation can be seen as systematic, weakly controlled non deterministic search of all solutions to an initially given query, to be found in a space of possible solutions. For a formalization we represent the set of computation states by a set of nodes, a subset of which will be structured as a tree to reflect the control conditions (like pruning) imposed by Gödel on angelic non determinism, an idea used already in (Börger 1990b) (in hybrid stack oriented form) and in (Börger and Rosenzweig 1991a, 1993a) as basis to define backtracking in an abstract model for Prolog. Therefore *Gödel Algebras* have as basic universe a set **Node** representing all possible *Gödel computation states*, containing a distinguished element (0-ary function) *currnode* representing the current computation state.

[2]See (Beierle and Börger 1992) for a formal specification, using the methodology of evolving algebras, of polymorphic types present at runtime for an extension of Prolog.

One aspect of Gödel's non determinism is that the computation can leave what it is doing at any part of the search space and compute at another part, maybe returning later; this abstracts from any particular strategy, like depth-first or breadth-first or others, in building and searching the solution tree, and brings Gödel computations rather close to the logician's understanding of deductions. We reflect this non determinism explicitly by treating *currnode* as dynamic external function. This means that *currnode* will never be updated by any of our rules, but is nevertheless supposed to change value due to some outside (hidden or implementation defined) control. The reader might be helped by considering *currnode* as a demon who executes a basic logical computation step. The control determines not the logic of what the demon does, but the place where he is put to work. This control reflects some restrictions to full non determinism (like that no step should be repeated, that pruning should be respected etc.), which we will formulate explicitly below. [3]

Each element n of **Node** has to carry the relevant information for a complete description—at the abstraction level of the user of the language which interests us here—of the computation state it represents, thus allowing to describe the execution of its subsequent computation in the subtree rooted at n. At the desired abstraction level this information consists of the literals which have still to be computed, the substitution computed (or the constraint system accumulated) so far and the statements which are still to be considered as candidates for alternative computations. We formalise these concepts by three functions

$$goal : \mathbf{Node} \to \mathbf{Goal}$$

(associating with each node the goal which is still to be computed)

$$sub : \mathbf{Node} \to \mathbf{Subst}$$

(representing the substitution (or the sytem of constraints) current at a state)

$$cands : \mathbf{Node} \to \mathbf{Clause}^*$$

(representing the relevant candidate clauses at given state), where $\mathbf{Clause}^*$ denotes (a subset of) the power set of the set **Clause** of (user-defined) program statements.

[3] Thus we decide to exclude, from the specification of the semantics of Gödel programs at this level of abstraction, any explication of the dynamics of *currnode*. This means in particular that we can avoid to introduce the tree structure on **Node** into the signature of Gödel algebras—say by a partial function $parent : \mathbf{Node} \to \mathbf{Node}$ (undefined on *root*), to be dynamically updated through rules in such a way that from each node (different from *root*) there is a unique *parent* path towards *root*. We can define instead at the meta-level, in terms of applications of our rules, a parent-relation which will allow us to speak of **Node** as a tree. This will be useful for the description of pruning.

Goal is the set of (normal) Gödel goals, i.e. terms constructed applying the (*and*) operator "&" to the empty goal and elements of universes **Lit**, **Conditional** and **Commit**. **Lit** denotes the universe of (positive and negative) literals; **Conditional** denotes the set of Gödel *conditionals* and comes with three functions *cond, then, else* which on arguments `IF` *Cond* `THEN` *Formula1* `ELSE` *Formula2* yield the conditional's guard, THEN part, ELSE part respectively, where *Cond, Formulai* belong to the universe **Goal**. **Commit** is the universe of terms involving the *commit* operator, i.e. terms of the form $\{g_1 \& \cdots \& g_n\}\_l$, $n \geq 0$, where l is (an integer defining) the *label* of the commit and $g_i \in \mathbf{Lit} \cup \mathbf{Conditional} \cup \mathbf{Commit}$. Note that commits are not allowed to occur inside conditionals. **Term** is the set of (generic) Gödel terms which contains **Lit**, **Conditional**, **Commit**, **Goal** as subsets.

Subst is a set of (not further specified) *substitutions* and comes together with three abstract functions

$$unify : \mathbf{Term} \times \mathbf{Term} \rightarrow \mathbf{Subst} \cup \{nil\}$$

associating to two terms either their unifying substitution or the answer that there is none; the substitution applying function

$$subres : \mathbf{Term} \times \mathbf{Subst} \rightarrow \mathbf{Term}$$

yielding the result of applying the given substitution to a given term; substitution concatenation

$$\circ : \mathbf{Subst} \times \mathbf{Subst} \rightarrow \mathbf{Subst}.$$

Note that the presentation in terms of substitutions does not limit the generality of our definitions. The latter apply indeed to arbitrary constraint systems by replacing unifiability of terms by solvability of constraints, unification by constraint solving and concatenation of substitutions by accumulation of constraints. (See (Börger and Schmitt 1991; Beierle and Börger 1992) for a detailed account of abstract constraint handling in the framework of evolving algebras.)

Clause comes with auxiliary functions *clhead* : **Clause** $\rightarrow$ **Lit**, *clbody* : **Clause** $\rightarrow$ **Goal** yielding *head* and *body* of Gödel statements. The current program is represented by a distinguished element *db* (database) of a universe **Program**. The candidate clauses which have to be considered in a call to define the alternatives of a computation state, are accessed using an abstract function

$$procdef : \mathbf{Lit} \times \mathbf{Program} \rightarrow \mathbf{Clause}^*$$

which yields the (normal) statements defining, in the given program, the predicate having the same functor as the given literal.

For reasons of transparency we separate two different aspects of the

action to be taken at *currnode* in executing a call: to collect the candidate statements for the potential alternative computation states and to select one of them for execution, distinguished by values *call*, *select* of a monadic function *mode* with domain **Node**[4]. To be able to speak about *termination* we will use distinguished elements *root* (with the obvious intended meaning) and *stop* with values in {0,1,-1} to indicate running of the system, stop with success and stop due to final failure respectively.

4 Delaying computations

The Gödel computation rule[5] is partly built into the system and partly under the control of the programmer through explicit `DELAY` control declarations. Let **DelayDecl** be the set of `DELAY` declarations and *delaydecl(db)* the set of delay declarations occurring in program *db*. To describe Gödel's goal selection abstractly (thus leaving it largely implementation dependent) let

$$gl\_select : \mathbf{Goal} \times \mathbf{DelayDecl} \times \mathbf{Subst} \to \mathbf{Lit} \cup \mathbf{Conditional}$$

be a (partial) function which selects from a given goal either a positive literal—which under the given substitution is non-delayed according to the given `DELAY` declarations—or a negative literal (which is ground) or a conditional (with closed guard formula)[6]. (Note that the selected literal or conditional may occur within the scope of a *commit*.) If in the current goal, wrt the current substitution and the current program, there is no such "non-delayed" literal or conditional, the computation *flounders*. *gl_select* reflects this abstractly by not being defined in this case, thus preventing applicability of any transition rule of our system because the value of *act* in the rule guards is undefined, where *act* (for *activator*) stands for

$$act \equiv gl\_select(goal(currnode), delaydecl(db), sub(currnode)).$$

The transition rules in section 5 can be read in terms of *act*, abstracting from the details of delay specification. Thus our rules apply to any goal selection mechanism of any logic programming system. A natural refinement step for Gödel programs consists in further specifying the notion of delayed positive literal, used in the above abstract specification of *gl_select*

[4] For an abstract formulation of the restrictions imposed upon angelic non determinism by pruning, negation and conditionals, it will be useful to introduce an additional mode value *abandoned*, see below.

[5] If one wants to include at this level also the constraint solving part, this could abstractly be dealt with incorporating the method of (Börger and Schmitt 1991).

[6] The present version of Gödel really does *not specify any more that negative literals must be ground or that the condition in a conditional must be closed. The implementation just has to implement correctly the declarative meaning. However, these are the likely restrictions implementations will make in the near future.* Cited from (Lloyd 1993)

(and therefore of *act*). Following (Hill and Lloyd 1992) the `DELAY` declaration *delaydecl(l,P)* for a predicate (literal) l in a program P has form `DELAY` *Atom_1* `UNTIL` *Cond_1* ... *Atom_n* `UNTIL` *Cond_n* with conditions built up from predicates NONVAR, GROUND, TRUE using conjunction and disjunction, and atoms pairwise without common instance. A set d of `DELAY` declarations delays calls as follows:

- an atom l is *delayed*, if it has a common instance with some *Atom_i* in a `DELAY` declaration in d without being an instance of *Atom_i*;
- an atom l is *delayed*, if it is an instance of an *Atom_i* in a `DELAY` declaration in d whose corresponding condition *Cond_i* is not satisfied under the instantiating substitution;

It is a routine task to formalize this definition for a refinement of *gl_select*.

5 Transition Rules

We now define the rules by which the system, starting from an algebra with $stop = 0$ - we tacitly assume this running condition to be part of each rule guard - tries to reach successful execution of the query signalled by say $stop = 1$. To abbreviate, we usually suppress the parameter *currnode* simply writing

$$\begin{aligned} mode &\equiv mode(currnode) \\ cands &\equiv cands(currnode) \\ s &\equiv sub(currnode) \\ goal &\equiv goal(currnode) \end{aligned}$$

5.1 Initialization and Rules for user-defined predicates

We do not assume Gödel Algebras to be given as static (infinite) search space; they rather evolve dynamically along the computation, starting from initial algebras determined by given program `P` and query `Q`. Hence the *initial (ization of) Gödel Algebras*: **Node** = {*root,currnode*}, *goal:=* `Q`, *mode:= call, s:=* $\emptyset$, *db:=* `P`, *stop:= 0. cands* is not (yet) defined at *currnode*.

The basic computation step, applicable to *user-defined* predicates, is split into *calling* the activator (to look for the candidate statements for alternative computations at *currnode*) and *selecting* one of them for execution. We will correspondingly have two rules: a **Call Rule** and a **Selection Rule**.

The following **Call Rule**, applicable to nodes with user defined activator in *call* mode, will store the relevant candidate statements (e.g. clauses whose head may unify with *act*, or whose constraints are consistent with the set of already accumulated constraints), copying from the procedure definition of *act* into *cands*(*currnode*). To ensure that this action of candi-

date collection takes place at each n at most once, when n gets first visited (becomes value of *currnode* in *call* mode), the mode turns to *select*.

If *mode = call & is_user_defined(act)*
then *cands :=procdef(act,db)*
mode:= select

The following **Selection Rule**, applicable to nodes with user defined activator and in *select* mode, attempts to execute a candidate statement, to be found in *cands(currnode)* (the set of remaining alternatives), using an abstract selection function[7]

$$alt\_select : \mathbf{Clause}^* \rightarrow \mathbf{Clause}.$$

Since this selection function too is kept abstract (considered as implementation defined), the formalization given here for Gödel applies mutatis mutandis to any logic programming system.

If a candidate statement has been selected, it is erased from the list of candidates (and thereby will not be tried again). If the selected statement does not apply successfully to *act*—typically, in the case of resolution, because the renamed head of the selected clause does not unify with the activator—, nothing else is done. Otherwise the selected statement is applied: a new node in *call* mode is created [8], its goal is defined—in the case of resolution by executing the resolution step, replacing the activator by clause body and applying the unifying substitution—and its substitution is updated (in the resolution case by the unifier between the clause head and the calling literal). Note that *currnode* remains in select mode, so that at this node further candidate children may be created.

If *mode = select & is_user_defined(act)*
then let $alt \equiv alt\_select(cands)$
$cands := cands \setminus \{alt\}$
If *apply(alt,act)* $\neq$ *nil*
then Extend Node by t **with**

[7] For Prolog it is well known that due to sequential execution of program clauses, different (even consecutive) occurrences of the same clause in a program may have different effect during a computation. For this reason a faithful description of Prolog has to distinguish between occurrences of clauses and the clauses themselves, see (Börger 1990a; Börger and Rosenzweig 1993a). The introduction of an abstract clause selection function allows to hide this distinction, at this level of abstraction, and to reduce the question of different occurrences of a same clause to whether *alt_select* and thereby *procdef* have as range a set or a multi–set.

[8] Each time we introduce a new node, if not otherwise stated we will consider the present value of *currnode* as its parent node, thus providing the above mentioned tree structure on **Node** which will be needed to deal with Gödel's pruning operator. To reflect the desired non determinism of Gödel in an unspoiled way, we do *not* define this tree structure by updates *parent(t):= currnode* in the rules.

mode(t):= call
define goal(t) using apply(alt,act)
update sub at t
EndExtend

The function *apply(alt,act)* and the updates *define goal(t) using* θ, *update sub at t* are deliberately kept abstract here; they depend on the basic computation mechanism. In case of resolution they can be specified furthermore by:

$$\begin{aligned} &\textit{apply(alt,act)} \equiv \textit{unify(act,Hd)} \\ &\textit{define goal(t) using } \theta \equiv \textit{goal(t):=} \ (\textit{goal}[\textit{act} \leftarrow \textit{Bdy}])\ \theta \\ &\textit{update sub at } t \equiv \textit{sub(t):=}\ s \circ \theta \end{aligned}$$

where

$$\textit{Hd} \leftarrow \textit{Bdy} \equiv \textit{rename(alt,currnode)}$$

$$\theta \equiv \textit{unify(act,Hd)}$$

$G[lit \leftarrow term]$ is the result of replacing (the occurence of) lit in goal G by $term$, and

$$\textit{rename} : \mathbf{Term} \times \mathbf{Node} \rightarrow \mathbf{Term}$$

associates with a term a new copy of it where all variables *and all commit labels* are renamed at the level determined by the given node. (Thus we handle variable renaming abstractly, avoiding details of term and variable representation.)

Obviously one has to reflect the correctness assumption for applications of Hilbert's ϵ-operator, namely that the selection function is not applied to an empty domain (here *cands*). One possibility is to let the system check this condition in form of a guard *If cands* $= [\,]$, whose satisfaction in Prolog triggers backtracking (Börger 1990b). If one wishes to keep such control features outside the system (as belonging to implementation and not to semantics), one can impose the correct use of the selection mechanism by an external condition on how *currnode* is allowed to be chosen. Such a condition can be viewed as a directive to be respected by an implementor. The following rather weak condition suffices here:

ϵ-choice condition: *currnode* never assumes as value a node in mode *select* with empty *cands*.

This condition avoids also repetitions (i.e. trying to apply statements in a part of the deduction which has already been explored.)

The **Query Success rule** stops the computation with successful halt when hitting the empty goal $\Box$: **If** *mode* $=$ *call* & *goal* $= \Box$ **then** *stop:= 1.*[9]

[9] This rule only stops the system, without giving output or looking for further so-

5.2 The pruning operator

In this section we refine the system to include Gödel's pruning operator, called *commit*, of form $\{Formula\}\lrcorner$ (with bar commit | of concurrent programming languages and one solution commit $\{\ldots\}$ as special case). The brackets $\{\ldots\}$ indicate the *scope* of the commit inside a statement, the label l the scope of the commit over the statements in a procedure definition. The computational effect of an l-commit $\{Formula\}\lrcorner$ is that once *a* solution is found for *Formula*, a) each potential alternative solution for *Formula* is skipped, and b) all potential alternatives are skipped which arise from other statements (in the underlying procedure definition) containing a commit with the same label l.[10]

To perform actions a) and b) one has to keep track of the point in the computation where label l has been introduced into the current goal: the value of *currnode* when—in a selection rule application—l enters the newly updated goal as (renaming of a) label occurring in the body of the selected statement. Since this node will be the starting point of the pruning operation realizing a) and b), to be executed on the computation tree once the formula in the scope of an l-commit has been satisfied, we call it *pruning point* and denote it by a function

$$prun\_pt : \mathbf{Integer} \rightarrow \mathbf{Node}$$

whose values are set by updates

$$prun\_pt(l) := currnode,$$

to be added to the Selection Rule for each l occurring as (renamed and therefore fresh) commit label in the body of the selected input statment.[11]

In order to satisfy stipulation a), when a commit $\{Formula\}\lrcorner$ has become empty at *currnode* (through successful computation of the formula in the scope), one has to inspect the path[12] from the introduction of the commit—namely at *prun_pt(l)*— to *currnode* and to abandon each[13] branch

---

lutions. These features can easily be provided using the substitution associated to the halting state.

[10] From the declarative viewpoint the programmer might be asked to use pruning only where semantically correct, i.e. to prevent parts of the tree which are known to compute no new solutions from being explored. Any statement which relates "declarative" to "procedual" program meaning has to take into account the possibly semantical effect of pruning.

[11] This is an adaptation of the *goal decoration by cutpoints* introduced in (Börger 1990a). For the initialization it seems reasonable to assume *prun_pt(l) = root* for each commit label l in the initial query, and to define *root* as parent of the initial value of *currnode*.

[12] It is here that the tree structure of **Node**, defined at the meta-level as indicated above, is used.

[13] We describe eager pruning.

which has been or still may be created for an alternative solution of *Formula*. Such branches are created for a potential alternative solution of an activator which occurred during the (successful) computation of *Formula*. Consider the potential children at each node n on the path which has its activator in the scope of a commit with label l. These children which either will be or have been created and are not on the path must never be selected. Thus we abandon those children that exist and prevent the remaining clauses in *cands(n)* from creating new children. Using the wording of (Hill and Lloyd 1992) we access these nodes and candidate clauses, which may have to be abandoned and deleted respectively, by abstract functions

$$children\_1st\_kind : \mathbf{Node} \times \mathbf{Integer} \rightarrow \mathbf{Node}^* \cup \{nil\}$$

and

$$cands\_1st\_kind : \mathbf{Node} \times \mathbf{Integer} \rightarrow \mathbf{Clause}^*$$

(It would be easy to define these functions dynamically by updates to be added to Call and Selection Rule in case the activator is in the scope of a commit with label l.) Requirement a) can therefore be formalized by adding to the Selection Rule the update

Let *path = path(currnode,prun_pt(l))*
$\forall n \in$ *path*:
    cands(n):= cands(n) \ cands_1st_kind(n,l)
    $\forall m \in$ *children_1st_kind(n,l) \ path*: *mode(m):= abandoned*

with *path(n,n')* the set of nodes which lie on the path from n to n', and by adding the following *Pruning control condition* on how *currnode* is allowed to be chosen. This restriction is formulated in terms of a mode value *abandoned*:

Pruning control condition: *currnode* never assumes as value a node in mode *abandoned* or below such a node.

Note that if one wants to have both, control management outside the system which defines the semantics and a pruning operation inside, then the effect of pruning upon future choices of *currnode* can be formulated only as external condition (as directive for implementations).

In order to satisfy stipulation b), when a commit $\{Formula\}\_l$ has become empty at *currnode*, one has to delete at *prun_pt(l)*—where the commit has been introduced—each candidate clause which contains a commit with same label l, and to abandon each child which has been created for executing such a clause. We access those nodes and candidate clauses again by (dynamic and as such easily definable) functions

$$children\_2nd\_kind : \mathbf{Node} \times \mathbf{Integer} \rightarrow \mathbf{Node}^* \cup \{nil\}$$

$$cands\_2nd\_kind : \mathbf{Node} \times \mathbf{Integer} \rightarrow \mathbf{Node}^*$$

Then it suffices to add to the Selection Rule the following updates [14]:

cands(prun_pt(l)):= cands(prun_pt(l)) \ cands_2nd_kind(prun_pt(l),l)
∀m ∈ children_2nd_kind(prun_pt(l),l) \ path: mode(m):= abandoned

where *path* is defined as above.

Summing up the preceding discussion, we can formalise the pruning operation by refining the goal update (replacement of *act* by *Bdy* in *goal* to *goal*[*act* ← *Bdy*]'), which now includes also deletion of empty commits, and by adding to the previous **Selection Rule** the following updates:

let *commgoal* ≡ (*goal*[*act* ← *Bdy*]) *θ*
∀j ∈ label_set(Bdy) : prun_pt(j):= currnode
pruning(currnode,commgoal)

where *pruning*(*n*, *goal*) stands for the updates [15]:

∀l ∈ empty_commit_label(goal) :
Let *path = path(currnode,prun_pt(l))*
cands(prun_pt(l)):= cands(prun_pt(l)) \ cands_2nd_kind(prun_pt(l),l)
∀m ∈ children_2nd_kind(prun_pt(l),l)\ path: mode(m):= abandoned
∀n ∈ path :
cands(n):= cands(n) \ cands_1st_kind(n,l)
∀m ∈ children_1st_kind(n,l)\ path: mode(m):= abandoned

The auxiliary function *label_set*: **Clause** → **Integer*** yields the set of all labels occurring in a clause body, *empty_commit_label*: **Goal** → **Integer*** yields the set of all labels *l* of empty commits {}_*l* occurring in a given goal.

After the current goal has been updated—in the case of resolution by replacing a literal by the selected clause body—it may contain a commit with empty scope, say {}_*l*. This means that the formula *h* of a previously introduced commit {*h*}_*l* has been computed with success. (Note that by the syntax of Gödel, no clause body or initial query has empty commits.) In this case "pruning" must (and by the above pruning update will) be performed.[16]

5.3 The computation of negative literals

Computation of negative literals is delayed until they have become ground; this is assured by the abstract *gl_select* function introduced in section 4. If

[14] This is a relativization of the *cut* rule for Prolog in (Börger 1990a; Börger and Rosenzweig 1993a)

[15] Remember that commits may be nested.

[16] Note that if different commits in different parts of the tree have the same label, then they come from statements in the same procedure definition and have been introduced into the current goal by the same call. This reflects the scoping of labels over a procedure definition and the renaming of labels (together with variables) in calling.

a ground negative literal $\sim lit$ becomes activator, its computation is considered to succeed/fail if the computation of *lit* finitely fails/succeeds.

Therefore to compute $\sim lit$, a subcomputation for *lit* has to be started which includes a special action at its exit to report finite failure or success for *lit*. For soundness reasons, pruning has to be disabled inside such subcomputations, which can also be nested. In order to reflect all this in a simple way, we introduce auxiliary marks `neg_beg, neg_end(n)` which during the computation of *lit* act as delimiters of the still to be computed goal [17]. When entering the subcomputation, they are placed around *lit*. We impose that they can become visible to the function *gl_select*—and thereby value of *act*—only as pair bracketing the empty goal, thus signalizing that the computation of *lit* has succeeded.

With this proviso, the following **Negation Start Rule** formalizes the call of a negative (ground) literal $\sim lit$: control may now pass to a new node with goal `neg_beg` *lit* `neg_end`(*currnode*). Note that the subtree which has the newly created node as root models the tree associated to the (negation sub-) computation of *lit*. In order to assure that this subcomputation has been exited (by success or failure) before the computation of $\sim lit$ can be terminated, *currnode* has to go into a waiting status, realized by mode *select* with undefined *cands* list and the following *Negation control condition*:

Negation control condition: *currnode* never assumes as value a node, with negated literal as activator and in mode *select*, unless the child, created at this node for the corresponding subcomputation, has been *abandoned*.

If $mode = call \ \& \ act = \sim lit$
then Extend *Node* **by** t **with**
mode(t):= call
goal(t):= `neg_beg` *lit* `neg_end`*(currnode)*
sub(t):= s
EndExtend
mode:= select

If *currnode* comes back to a node with activator $\sim lit$ (and therefore in mode *select*), by the negation control condition its only child must have been abandoned and therefore all alternatives for *lit* have produced failure. Then $\sim lit$ is considered to have been computed with *success*. This is expressed by the following **Negation Success Rule**: the computation may continue here in *call* mode, after deleting $\sim lit$ from *goal*. (Note that the computation of a negative ground literal does not modify the substitution attached to *currnode*).

[17] This technique, well known to implementors, has been used with advantage in mathematical form already in (Börger 1990a; Börger and Rosenzweig 1993a).

If *mode* = *select* & *act* = ∼ *lit*
then *mode:=* *call*
goal:= delete *act* from *goal*

with the obvious meaning of deletion.

The following **Negation Failure Rule** formalizes the case when *lit* succeeds: this provokes failure of ∼*lit* (easily formalizable, due to pruning control condition, by abandoning the node where ∼*lit* had to be computed). It happens when *act* becomes `neg_beg neg_end`(*n*) in *call* mode, for some node *n*.

If *mode* = *call* & *act* = `neg_beg neg_end`*(n)*
then *mode(n):=* *abandoned*

To disable pruning inside a negated call (Hill and Lloyd 1992, p.77), it suffices to refine the *pruning* update by putting it under the (easily formalizable) guard:

If *act* is not in the scope of neg-brackets `neg_beg neg_end(n)` **Then.**

5.4 The computation of conditionals

In (Hill and Lloyd 1992) two forms of conditionals `IF` *Condition* `THEN` *Formula1* `ELSE` *Formula2* are distinguished, depending on whether *Condition* and *Formula1* share (local) variables. The version with shared variables— so called *Some*-version with guard of form *Some* [*x*] *Cond* for a sequence *x* of possibly shared variables— has to be treated separately from what one might call the normal version, because it allows backtracking from *Formula1* to *Cond*. By the *gl_select* function definition, *activator* takes a conditional as value only when its guard *Condition* (in the *Some*-version the expression *Some* [*x*] *Cond*) has no free variables.

Let **X-Conditional** be the set of **X**-conditionals for **X**=**S**(ome), **N**(ormal). For both cases we apply the marking technique introduced already for negation subcomputations, using delimiters `X_cond_beg,X_cond_end(n)` with corresponding stipulations for the visibility to *gl_select*. The computation of the guard is triggered by the following **Guard Evaluation Rule**, which (in the very same way as the negation Start Rule) creates a node, decorated in mode call for the computation of the guard, and puts *currnode* into waiting status (realized by assigning mode *select* and undefined *cands* list, given the *Conditional-Guard control condition* which extends the negation control condition to X-conditionals).

Conditional-Guard control condition: *currnode* never assumes as value a node, activated by an X-conditional and in mode *select*, unless the child, created at this node for the computation of the guard, has been *abandoned*.

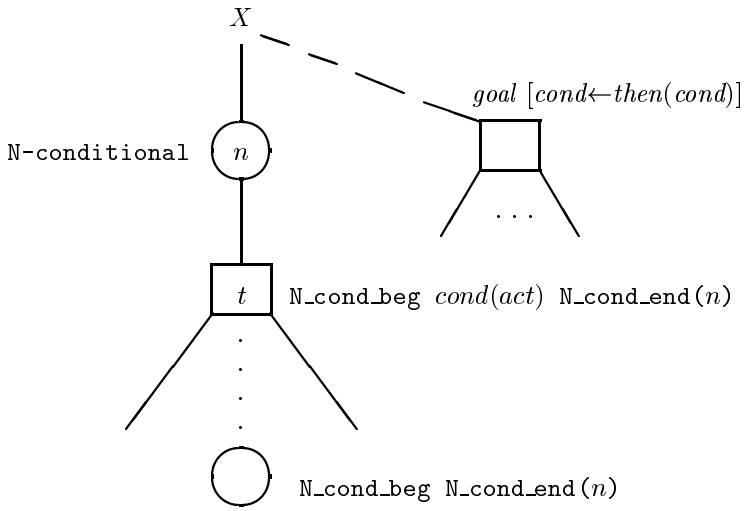

FIG. 1. The computation of an N-conditional's THEN-part

If *mode* = *call* & *act* ∈ **X-Conditional**
then Extend *Node* **by** *t* **with**
mode(t):= call
goal(t):= X_cond_beg *cond(act)* X_cond_end*(currnode)*
sub(t):= s
EndExtend
mode:= select

[18]The Conditional-Guard control condition prevents the demon to come back to the node with X-conditional activator *before* the guard computation started here has been terminated (by failure, see the ELSE rules below, or success, see the THEN rules below).

Due to the stipulations made for *gl_select* wrt the conditional markers, *act* = X_cond_beg X_cond_end(n) for some *n* signals that the guard of a conditional has been computed successfully. This triggers the following X_THEN **Rules** to create a new node *t*, ready in mode *call* for the computation of the conditional's THEN-part.

For normal conditionals (X=N) it has to be assured that the system looks only for *one* solution of the guard. By pruning control condition, it suffices to abandon the node *n* where the corresponding conditional was called and to create a new node for the computation of the THEN-part (see

[18]Thanks to Rosario Salamone for production of the two pictures.

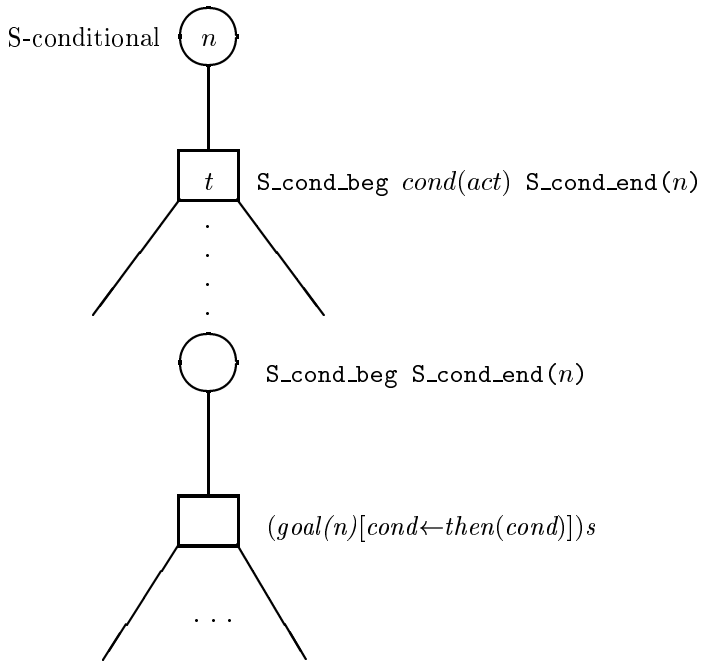

FIG. 2. The computation of an S-conditional's `THEN`-part

FIG.1)[19]. The goal and the substitution for the new node are defined without applying the substitution which has been computed during the guard evaluation, reflecting the fact that in a normal conditional the guard (which was ground when called) does not share variables with the `THEN`-part. (A similar remark applies to the goal update for the `THEN`-part in the `ELSE` rule below.)

The corresponding `S-THEN` rule has similar structure but different updates. Due to the possible presence of shared variables in S-conditionals, the substitution computed by the guard is passed to the computation of the `THEN`-part and applied to the result of the replacement in defining the new goal[20]. Since from the computation of an S-conditional's `THEN`-part, the system may (backtrack in an) attempt to resatisfy also the guard, the *mode* of *currnode* is set to *select*[21].

[19] Since *currnode* is below n, we define n and the newly created node to be siblings.

[20] For simplicity of exposition we assume here the computed substitutions to be idempotent; otherwise, instead of applying the current substitution s, one should apply only the substitution computed during the just terminated guard evaluation.

[21] Correspondingly the new node is defined to be child of *currnode*. See FIG.2.

To prevent the system from stepping, by *backtracking* at node n, into the ELSE rules, we set additionally a mark *then_case(n)* to 1. The idea in (Hill and Lloyd 1992) seems to be that once the guard of a conditional is satisfied, computation first proceeds with the THEN-part and only upon failure tries to resatisfy the guard. This sequentiality is reflected in the following restriction of the choice of *currnode*:

S-Then control condition: Let n be an S-conditional node. If *currnode* assumes as value a node in the subtree generated by n, then either there is no node m below n with goal S_cond_beg S_cond_end(n) whose child is not *abandoned* or there is exactly one such node and *currnode* is in the subtree of m[22].

We use the well known |-notation to exhibit structural similarity but difference in content of the following N- and S-THEN **Rules**:

If $mode = call$ &
$act =$ **N_cond_beg N_cond_end**(n) | **S_cond_beg S_cond_end**(n)
then let $cond \equiv gl\_select(goal(n), delaydecl(db), sub(n))$
$newgoal \equiv goal(n)[cond \leftarrow then(cond)]$
Extend *Node* **by** t **with**
$mode(t) := call$
$goal(t) := newgoal \mid goal(t) := (newgoal)s$
$sub(t) := sub(n) \mid sub(t) := s$
Endextend
$mode(n) := abandoned \mid then\_case(n) := 1$
$\mid mode := select$

For S-conditionals, the THEN rule puts *currnode* into mode *select*; if *currnode* comes back to this point (in this mode), it gets abandoned:

If $mode = select$ & $act =$ S_cond_beg S_cond_end(n)
then *mode:= abandoned*

If the conditional guard evaluation has failed, this will allow *currnode* to return to the conditional node which originated the guard evaluation subtree and thereby passed to mode *select* (see conditional-guard control condition). In this case, by the following ELSE **Rule**, the conditional node will be abandoned and a new node is created [23], ready (in *call* mode and decorated by goal and substitution) for the computation of the ELSE-part of the conditional. Note that for S-conditionals this is allowed to happen only if the then-case has not been entered; if the THEN-case has been entered,

[22] With depth-first search, this apparently complicated condition becomes simple, see the next section.

[23] Due to abandoning of *currnode*, the latter and the new node are defined as siblings.

the system cannot enter the ELSE-part and therefore *backtracks* (through the remaining choices for the value of *currnode*).

If *mode* = *select* &
  act ∈ **N-Conditional** | *(act* ∈ **S-Conditional** & *then_case* ≠ 1*)*
then Extend *Node* **by** *t* **with**
    mode(t):= call
    goal(t):= goal[act ← *else(act)]*
    sub(t):= s
  EndExtend
  mode(currnode):= abandoned

If *mode* = *select* & *act* ∈ **S_Conditional** & *then_case* = *1*
then *mode:= abandoned*

(We assume that in conditionals without ELSE-part, *act* is deleted. See also the remark above to the N_THEN rule.)

6 Depth-First Search and Concurrency

In this section we outline briefly how the preceding definition of semantics of Gödel programs can be refined to include depth-first search (as a typical step towards efficient implementation) or concurrency (as the theoretically most challenging extension).

The ground for depth-first search has already been prepared by the definition of the parent-children relation used to describe the semantics of pruning. The definition of the *parent* function becomes part of the (semantics defining) evolving algebra by including the corresponding update into the rules. Namely add the update

$$parent(t) := currnode$$

to the rules for selection, negation start, conditional guard evaluation, S-THEN; the update

$$parent(t) := parent(n)$$

to the N-THEN rule, and the update

$$parent(t) := parent(currnode)$$

to the ELSE rules. The depth-first strategy is incorporated into the system by adding to all above mentioned rules the update

$$currnode := t$$

and setting initially *parent(currnode):= root*. The control conditions are then easily implemented using the backtracking update

$$\begin{array}{lll} backtrack & \equiv & \textbf{if } parent = root \\ & & \textbf{then } stop := -1 \\ & & \textbf{else } currnode := parent \\ & & \quad mode(parent) := Select \end{array}$$

The ϵ-choice condition is implemented by introducing the additional update

$$\textbf{If } cands = [\,] \textbf{ then } backtrack$$

into the `THEN`-part of the selection rule. The pruning control condition is realized by reinterpreting the update

$$mode(m) := abandoned$$

as *cands(m)*:=[ ]. Under the same interpretation of being *abandoned*, the control conditions for negation, conditional guard and `S-THEN` are then automatically satisfied. Thus it is a simple exercise to prove the following

Proposition 6.1. *Let G be the evolving algebra defined in section 1-5 and G' its refinement as indicated in this section. Then G' correctly implements G under the strategy of depth-first search.*

Now to the question of what is needed to build a parallel implementation of the Gödel language. The recent work of Glavan and Rosenzweig (1993) defines a notion of *concurrent runs* within the framework of evolving algebras, which is based upon a notion of *independence* of rules and runs. Using those notions it is easy to extend the above interpretation of the Gödel interpreter I by allowing different demons to execute concurrently the rules of I, restricted only by the independence property. The problem of a parallel implementation of Gödel is thereby reduced to the (not at all trivial) problem of implementing an interesting independence notion for our rules (and thereby runs). This corresponds to what has been experienced by us in our formalization of the two forms of parallel logic programming represented by PARLOG (Börger and Riccobene 1993) and CONCURRENT PROLOG (Börger and Riccobene 1992).

7 Conclusion and Outlook

The abstract interpreter for Gödel programs which has been developed in this paper—refining the core for user-defined predicates by introducing stepwise delay, pruning, negation, conditionals and their implementation under the depth-first strategy—constitutes a mathematical object of manageable complexity. It can therefore serve as a basis for mathematical investigations of computational effects of Gödel programs. Our abstract specification of the semantics of Gödel programs, which supports

the intuitive procedural understanding in a direct manner, can also serve as starting point for a hierarchy of specification refinements leading to actual implementation, where each refinement step comes with a mathematical correctness proof; very much along the lines of what has been achieved for PROLOG with respect to the Warren Abstract Machine (Börger and Rosenzweig 1993b) and for its typed extension PROTOS-L (Beierle *et al.* 1991) with respect to the Protos Abstract Machine (Beierle and Börger 1992). This may actually provide some methodological help for implementing Gödel.

8 Acknowledgements

The authors wish to express their warm thanks to John Lloyd and Pat Hill for careful study and fruitful discussion of many previous versions of this paper. Their criticism and illuminating comments helped us considerably in our attempt to faithfully model their basic intuitions of the Gödel programming language.

The work was started when the second author was visiting (March–June 1992) at the Computer Science Department, University of Bristol (UK); it has been partially supported by "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of CNR, under Grant n.90.00671.69.

Bibliography

1. Apt, K. (1990). Logic Programing. *Handbook of Theoretical Computer Science* (ed. J.van Leeuwen). Elsevier, *vol.B*, 493–574.
2. Beierle, C. and Börger, E. (1992). Correctness Proof for the WAM with types. *CSL'91, 5th Workshop on Computer Science Logic* (eds. E.Börger, H.Kleine Büning, G.Jaeger, M.M.Richter). Springer *LNCS 626*, pp. 15–34.
3. Beierle, C. and Meyer, G. and Semle, H. (1991). Extending the Warren Abstract Machine to Polymorphic Order-Sorted Resolution. *Logic Programming: Proceedings of the 1991 International Symposium* (eds. V.Saraswat, K.Ueda), pp. 272–288. MIT Press.
4. Blakley, B. (1992). A Smalltalk Evolving Algebra And Its Uses. *Ph. D. Thesis*, University of Michigan.
5. Börger, E. (1990a). A Logical Operational Semantics for Full Prolog. Part I: Selection Core and Control. *CSL'89, 3rd Workshop on Computer Science Logic* (eds. E. Börger, H.Kleine Büning, M.M.Richter), Springer *LNCS 440*, pp. 36–64.
6. Börger, E. (1990b). A Logical Operational Semantics for Full Prolog. Part II: Built-in Predicates for Database Manipulations. *Proc. of Mathematical Foundations of Computer Science* (ed. B.Rovan), Springer *LNCS 452*, pp. 1–14.

7. Börger, E. (1992). A Logical Operational Semantics of Full Prolog: Part 3. Built-in Predicates for Files, Terms, Arithmetic and Input–Output. *Logic from Computer Science* (ed. Y.N.Moschovakis). MSRI Publications *vol.21*. Springer Verlag, pp. 17–50.
8. Börger, E. and López-Fraguas, F.J. and Rodríguez-Artalejo, M. (1993). Towards a Mathematical Specification of a Graph-Narrowing Machine. Manuscript (submitted).
9. Börger, E. and Riccobene, E. (1993). A formal specification of PARLOG. *Semantics of Programming Languages and Model Theory* (eds. M.Droste & Y.Gurevich). Gordon and Breach (to appear).
10. Börger, E. and Riccobene, E. (1992). A Mathematical Model of Concurrent Prolog. *CSTR-92-15*, Dept. of Computer Science, University of Bristol.
11. Börger, E. and Rosenzweig, D. (1991a). Prolog Tree Algebras. A formal specification of Prolog. *Proceedings ITI'91 (13th International Conference on "Information Technology Interface)*, Zagreb, pp. 513–518.
12. Börger, E. and Rosenzweig, D. (1991b) From Prolog Algebras Towards WAM - A Mathematical Study of Implementation. *Computer Science Logic* (eds. E.Börger, H.Kleine Büning, M.M.Richter, W.Schönfeld). Springer *LNCS 533*, pp. 31–66.
13. Börger, E. and Rosenzweig, D. (1991c) WAM Algebras - A Mathematical Study of Implementation. Part II. *Logic Programming* (ed. A.Voronkov). Springer *LNCS 592*, pp. 35–54.
14. Börger, E. and Rosenzweig, D. (1993a). A Mathematical Definition of Full Prolog. *Science of Computer Programming* (to appear). Preliminary version: *TR 33/92*, Dipartimento di Informatica, Universita di Pisa, pp. 23.
15. Börger, E. and Rosenzweig, D. (1993b). The WAM—Definition and Compiler Correctness. *Logic Programming: Formal Methods and Practical Applications* (eds. C.Beierle & L.Plümer). *Studies in Computer Science and Artificial Intelligence*, North–Holland (to appear). Preliminary version: *TR 14/92*, Dipartimento di Informatica, Universita di Pisa, pp. 57.
16. Börger, E. and Schmitt, P. (1991). A formal operational semantics for languages of type Prolog III. *Computer Science Logic* (eds. E.Börger, H.Kleine Büning, M.Richter, W.Schönfeld). Springer *LNCS 533*, pp. 67–79.
17. Glavan, P. and Rosenzweig, D. (1993) Communicating Evolving Algebras. *Computer Science Logic* (eds. E. Börger, S. Martini, G.Jäger, H.Kleine Büning, M. M. Richter). Springer *LNCS 702*, pp. 182-215.
18. Gottlob. G. and Kappell, G. and Schrefl, M. (1991). Semantics of Object-Oriented Data Models - The Evolving Algebra Approach. *Next*

Generation Information System Technology (eds. J. W. Schmidt & A. A. Stogny). Springer *LNCS 504*, pp. 144–160.

19. Gurevich, Y. (1988). Logic and the challenge of Computer Science. *Trends in Theoretical Computer Science* (ed. E.Börger). Computer Science Press, Rockville MA, pp. 1–57.
20. Gurevich, Y. (1991). Evolving Algebras. A Tutorial Introduction. *EATCS Bulletin*, *43*, February 1991.
21. Gurevich, Y. and Huggins, J.K. (1993). The Semantics of the C Programming Language. *Computer Science Logic* (eds. E.+Börger, S.Martini, G.Jäger, H.Kleine Büning, M.M.Richter). Springer *LNCS 702*, pp. 274-308.
22. Gurevich, Y. and Morris, J. (1988). Algebraic operational semantics and Modula-2. *CSL'87, 1st Workshop on Computer Science Logic* (eds. E.Börger, H.Kleine Büning, M.M.Richter). Springer *LNCS 329*, pp. 81–101.
23. Gurevich, Y. and Moss, L. (1990). Algebraic Operational Semantics and Occam. *CSL'89, 3d Workshop on Computer Science Logic* (eds. E.Börger, H.Kleine Büning, M.M.Richter). Springer *LNCS 440*, pp. 176–192.
24. Hill, P.M. and Lloyd, J.W. (October 1992) The Gödel Programming Language. *CSTR-92-27*, University of Bristol.
25. Hill, P.M. and Lloyd, J.W. and Shepherdson, J.C. (1990). Properties of a pruning operator. *Journal of Logic Programming*, *vol.1 - n.1*, pp. 99–143.
26. Lloyd, J.W. (1987). *Foundations of Logic Programming* (2nd edn). Springer Verlag.
27. Lloyd, J.W. (1992). *Private communication*, e-mail of 3-Nov-1992.
28. Lloyd, J.W. (1993). *Private communication*, e-mail of 5-April-1993.