ICOT Technical Memorandom: TM-1184

TM-1184

Proceedings of EGCS '92 WORKSHOP on Automated Deduction

by
R. Hasegawa & M. Stickel (SRI)

July, 1992

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191 ~5
Telex ICOT J32964

Institute for New Generation Computer Technology

INTERNATIONAL CONFERENCE ON
FIFTH GENERATION COMPUTER SYSTEMS 1992

WORKSHOP ON
AUTOMATED DEDUCTION
LOGIC PROGRAMMING AND PARALLEL COMPUTING APPROACHES

June 6, 1992 Tokyo, Japan

PROCEEDINGS

Institute for New Generation Computer Technology

FGCS '92 WORKSHOP
- W3:Automated Deduction -

9:30 - 17:00, 6 (Saturday) June, 1992

Co-Chair:
Ryuzo HASEGAWA(ICOT)
Mark E. STICKEL(SRI)

at:
Shiba Park Hotel
1-5-10, Shiba-koen, Minato-ku, Tokyo
(TEL:Tokyo-3433-4141)

*** TIME TABLE ***

09:30 - 10:15	"First-order Shannon Graph" Joachim POSEGGA University of Karlsruhe
10:15 - 11:00	"A Heterogeneous Parallel Deduction System" Geoff SUTCLIFFE Edith Cowan University
11:00 - 11:30	(Coffee Break)
11:30 - 12:15	"A Functional Languaje for Parallel Automatic Deduction" Carlos ARAYA Costa Rican Institute of Technology
12:15 - 13:30	(Lunch)
13:30 - 14:15	"The Improvement of A Parallel Theorem Prover Based on The Model Generation Method" Tetsuji KUBOYAMA Kyushu University
14:15 - 15:00	"Improving Backward Execution in Non- Deterministic Concurrent Logic Language" Salvador ABREU Universidade Nova de Informa'tica
15:00 - 15:30	(Coffee Break)
15:30 - 16:15	"Improving Performance Evaluation of Parallel Inference System" Christian SUTTNER Technical University of Munich
16:15 - 17:00	Discussion
18:00 -	Joint Party at a Chinese Restaurant Kopri(TEL:Tokyo-3434-7375)

First-order Shannon Graphs

Joachim Posegga
Universität Karlsruhe
Institut für Logik, Komplexität und Deduktionssysteme
Am Fasanengarten 5, 7500 Karlsruhe, FRG
Internet: posegga@ira.uka.de

April 8, 1992

Extended Abstract[1]

1 Introduction

Experience in hardware verification has shown that Shannon graphs (introduced by Shannon in [Shannon, 1938], also called *Binary Decision Diagrams*) are very suitable as an underlying datastructure for proving logical properties of formulæ [Brace *et al.*, 1990, Bryant, 1986]. Up to now, this method did hardly influence research in automated theorem proving, probably because it has not been extended to full first-order logic, so far. We will outline how this can be done and argue that the concept of Shannon graphs is a useful framework for implementing a first-order deduction system.

The idea underlying the proof procedure is to transform a formula into a Shannon graph, and compile this graph into a program which shows the formulæ's inconsistency when it is executed. The input formula may have arbitrary logical connectives, so no initial normal form is required. We will sketch how to perform the compilation for Prolog as a target language, although any other general-purpose language can be used. The generated program models the search through the graph which can be understood as a case analysis over the truth values of the atoms of the input formulæ. The search process tries to show that no model for the formulæ can exist by exploiting properties of the graph that are logically equivalent to the fact that there is no model. The generated clauses have no logical relation to the formula that is to be proven (in the sense that they are not a logically equivalent variant of the formula).

An experimental Prolog-implementation of a propositional prover based on these principles can prove pigeon-hole seven (ie "*seven pigeons do not fit into six holes*") in less than one minute on a Sun 4. This can be considered as good performance, so the extension of the principle to first-order logic, which is currently being implemented, seems promising.

2 Basic Definitions

Let $\mathcal{L}$ be the language of (propositional/first-order) calculus defined in the usual way, and $\mathcal{L}_{At}$ the atomic formulæ of $\mathcal{L}$. Assume further, that the language $\mathcal{L}$ does not contain the

[1] An extended version of this paper is available from the author.

atomic truth values "**1**" (*true*) and "**0**" (*false*). "*sh*" is a new logical connective of arity three, which can be regarded as an abbreviation: $sh(A,B,C) \equiv ((\neg A \wedge B) \vee (A \wedge C))$. Thus, $sh(A,B,C)$ corresponds to an expression of the form "**unless** A **then** B **else** C". The set of all Shannon Graphs is denoted by $\mathcal{SH}$ and defined over atomic formulæ and the atomic truth values as the smallest set such that

(1) $\mathbf{1}, \mathbf{0} \in \mathcal{SH}$

(2) if $\mathcal{G}_0, \mathcal{G}_1 \in \mathcal{SH}$ and $\phi \in \mathcal{L}_{At}$ then $sh(\phi, \mathcal{G}_0, \mathcal{G}_1)$ is in $\mathcal{SH}$.

Since the "*sh*"-operator, together with the atomic truth values, forms a logical basis, every (Skolemized) formula can be expressed as a Shannon graph. The transformation of a formula into this form is easily done. An atomic formula A is logically equivalent to $sh(A, \mathbf{0}, \mathbf{1})$; if we already have two graphs sh_A and sh_B representing the formulæ A and B, we construct $sh_{A \wedge B}$ by replacing all **1**-leaves of sh_A with sh_B. $sh_{A \vee B}$ is constructed analogously by replacing the **0**-leaves of sh_A. We can regard the resulting terms as trees, or as graphs, depending on how we represent the replacements of leaves. If a formula F is recursively converted by these rules, we will refer to the resulting Shannon graph as the *initial graph* for F. The time and space complexity of the above transformation *conr* is proportional to the size of the input formula if a graph is built.

In the sequel, we will use letters of the calligraphic alphabet $\mathcal{A}, \mathcal{B}, \mathcal{C}, \ldots$ to denote Shannon graphs. The formulæ of $\mathcal{SH}$ can be visualized as binary trees whose nodes are labeled with the atomic formulæ occurring in the first arguments of the *sh*-terms, and whose leaves are labeled with "**1**" or "**0**". The graphs shown in Figure 1 are examples. All leaves are represented as circles, and all other nodes as squares. The graphs have actually only one **1**-leaf and one **0**-leaf, but it is easier to draw them with multiple instances of the leaves. Edges labeled with "−" (the *negative edges*) and "+" (the *positive edges*) lead to the nodes representing the second and the third argument of the corresponding *sh*-terms. In the sequel, we will switch between referring to a "*sh*"-term and to its corresponding graph.

For convenience, three projections to access the arguments of a *sh*-expression are defined:

$$[sh(F,\mathcal{G}_1,\mathcal{G}_2)]_{\mathbf{at}} = F, \quad [sh(F,\mathcal{G}_1,\mathcal{G}_2)]_- = \mathcal{G}_1, \quad \text{and} = [sh(F,\mathcal{G}_1,\mathcal{G}_2)]_+ \quad \mathcal{G}_2.$$

Semantically, a *sh*-term can be regarded as a case-analysis over the truth values of atoms occurring in a formula. Assume there is a sequence of nodes and edges leading to a leaf of a graph; if the atoms at the nodes can be interpreted with the truth value that their outgoing edges suggest, then the whole formula will have the truth value the leaf is labeled with. This is the basic idea behind paths:

A *path* P in a Shannon graph $\mathcal{G}$ is defined as a sequence of subformulæ of $\mathcal{G}$ and denoted by an expression of the form $[v_1\mathcal{G}_0, \ldots, v_n\mathcal{G}_n]$, such that:

1. $\forall v_i \in \{v_1, \ldots, v_n\}: \; v_i \in \{-,+\}, \mathcal{G}_i \in \mathcal{SH}$
2. $\forall v_j \in \{v_1, \ldots, v_{n-1}\}: [\mathcal{G}_j]_{v_j} = \mathcal{G}_{j+1}$

A path is said to *start* at formula $\mathcal{G}_0$ (or: node $[\mathcal{G}_0]_{\mathbf{at}}$), and said to end at formula $[\mathcal{G}_n]_{v_n}$ (or: node $[[\mathcal{G}_n]_{v_n}]_{\mathbf{at}}$. Furthermore, a path P is said to be *inconsistent* if there is a substitution σ, such that for some $v_i\mathcal{G}_i, v_j\mathcal{G}_j \in P$: $[\mathcal{G}_i]_{\mathbf{at}}\sigma = [\mathcal{G}_j]_{\mathbf{at}}\sigma$, and $v_i \neq v_j$. As an

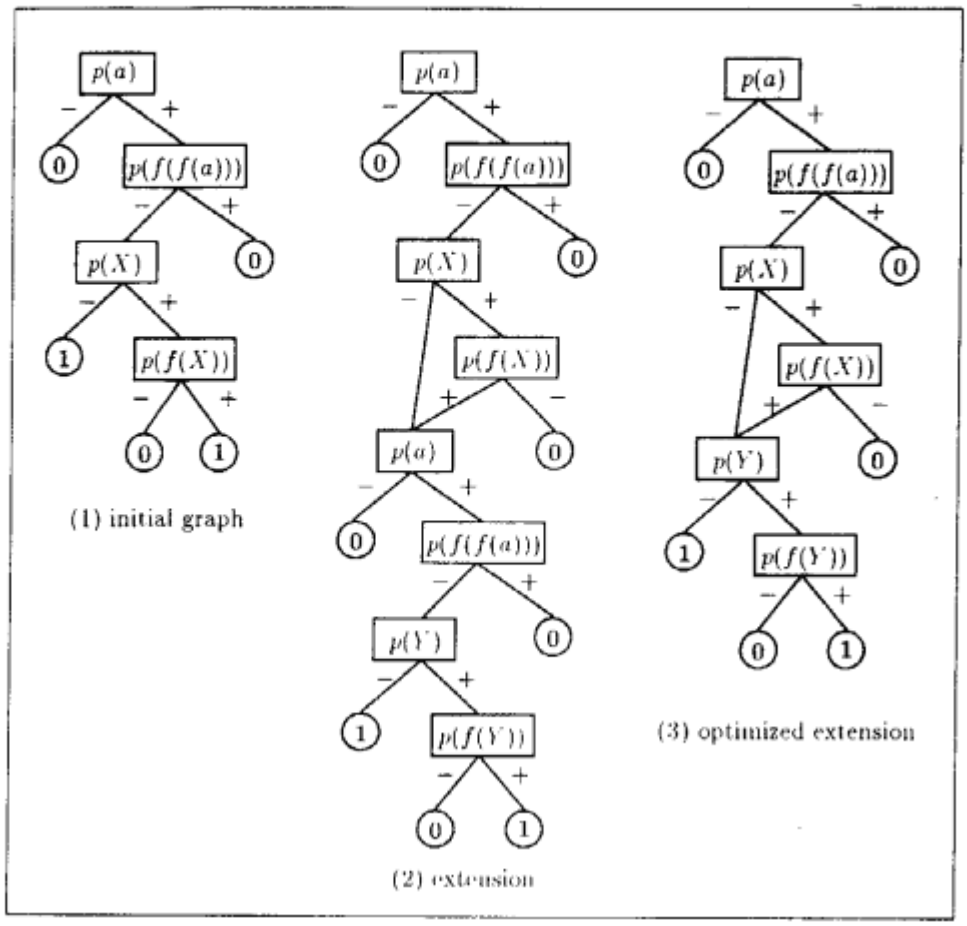

Figure 1: Proving inconsistency of $p(a) \wedge \neg p(f(f(a))) \wedge \forall(x)\ (p(x) \text{—} p(f(x)))$

example, consider the path $[+p(a), -p(f(f(a))), -p(X)]$ of $\mathcal{G}_0$, which is inconsistent under the substitution $[a/X]$.

The basic idea to show unsatisfiability of a formula is the following: build an initial graph $\mathcal{G}_0$ for a Skolemized first-order formula ϕ, choose a substitution σ such that all paths to 1-leaves become inconsistent. If there is no such substitution, extend $\mathcal{G}_0$ by substituting all of its 1-leaves with $\mathcal{G}_0'$, where $\mathcal{G}_0'$ is an instance of $\mathcal{G}_0$ with renamed variables. Then again try to find a substitution, such that all paths to 1-leaves of the new graph are inconsistent. The process continues recursively until eventually for some extension an appropriate substitution exists. This will be the case if the initial formula is inconsistent. Note that one extension of the graph corresponds to a conjunction of the original graph and a copy of it with renamed variables; what happens is basically the construction of the conjunction in the following consequence of the compactness theorem:

Proposition 2.1 *$Cl_\forall F(\bar{X})$ is unsatisfiable iff there exists some $k \in \mathbb{N}$ and a substitution σ, such that the conjunction $(F(\bar{X}_0) \wedge \ldots \wedge F(\bar{X}_k))\sigma$ is unsatisfiable.*
(σ maps all variables to ground terms, ie. to terms from the Herbrand-universe $\mathcal{U}_F$ of F. Each $\bar{X}_i$ is a new variable vector $X_{i,1}, \ldots, X_{i,n}$ distinct from all $\bar{X}_j$ with $j < i$.) ∎

Consider Figure 1: $p(a) \wedge \neg p(f(f(a))) \wedge \forall(x)\ (p(x) \text{—} p(f(x)))$ is to be proven incon-

sistent. (1) is the initial graph, but there is no substitution such that both the paths to 1-leaves are inconsistent. We then replace each 1-leaf by a copy of $\mathcal{G}_0$ with "fresh" variables, giving graph (2). Now, all paths to 1-leaves become inconsistent under the substitution $\{a/X, f(a)/Y\}$. Graph (3) is an optimization of (2) that exploits the fact that each path leading to the extension necessarily includes $[+p(a), -p(f(f(a)))]$.

For implementing the above method, we propose to compile the initial graph into a set of Horn clauses. Each node is translated into a Prolog clause having as arguments the path constructed so far, the current variable binding, and a designator of the current extension. A clause succeeds if at each 1-leaf reachable from the corresponding node the path can be made inconsistent. As we want to avoid asserting new clauses, an extension is best done by calling the clause for the top node again. To get a more concrete idea, we will briefly discuss a (simplified) clause for the node $p(X)$ of graph (1) in Figure 1:

```
node_3(Binding,Path,Level):-                                    (1)
     levelbinding(Binding,Level,1,X),                           (2)
     (closed_path([-p(X)|Path])                                 (3)
       ; node_1(Binding,[-p(X)|Path],succ(Level))),             (4)
     (closed_path([-p(X)|Path])                                 (5)
       ; node_4(Binding,[+p(X)|Path],Level)).                   (6)
```

In line 2, all free variables occurring in the corresponding atom are bound. To avoid going into technical details, we just assume that `levelbinding` binds `X` accordingly, if a designator of the level the process is currently in is passed (`Level`). The predicate `inconsistent` (line 3 and 5) tries to make `Path` together with `-p(X)` inconsistent since we reach a 1-leaf at the negative edge; if this fails, line 4 extends the graph by calling the clause for the top node again, which will succeed if all paths to 1-leaves in one of the next extensions can be made inconsistent. Thus, extensions can easily be modeled without asserting new clauses, because different levels can bind variables differently. The last goal to solve is to show that all paths to 1-leaves reachable from the positive edge of this node are inconsistent, as well (line 6).

Note that the compilation process can also be done in advance for a sets of axioms. Such precompiled theories can then be loaded into a deduction system and used for inference. Reducing the Shannon graph for a logical theory before compiling it into clauses can even prune the search space for a proof carried out later. Reduction is a well-known operation on Shannon graphs, which often decreases the size of a graph considerably. Reduction can be expensive, but this seems affordable in the context of precompiling theories, since it moves some of the effort for proving theorems to an earlier stage that is usually not time-critical.

References

[Brace *et al.*, 1990] K. S. Brace, R. L. Rudell, & R. E. Bryant. Efficient implementation of a BDD package. *Proc. 27th ACM/IEEE Design Automation Conf.*, pp. 40 – 45, 1990.

[Bryant, 1986] R. Y. Bryant. Graph–based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C 35:677 – 691, 1986.

[Shannon, 1938] C. E. Shannon. A symbolic analysis of relay and switching circuits. *AIEE Transactions*, 67:713 – 723, 1938.

A Heterogeneous Parallel Deduction System

Geoff Sutcliffe
Dep't of Computer Science, Edith Cowan University &
Dep't of Computer Science, The University of Western Australia
Perth, Western Australia geoff@cs.uwa.edu.au

Abstract
This paper describes the architecture, implementation and performance, of a heterogeneous parallel deduction system (HPDS). The HPDS uses multiple deduction components, each of which attempts to find a refutation of the same input set, but using different deduction formats. The components cooperate by distributing clauses they generate to other components. The HPDS has been implemented in Prolog-D-Linda. Prolog-D-Linda provides appropriate data transfer and synchronisation facilities for implementing parallel deduction systems. The performance of the HPDS has been investigated.

Parallel Deduction Systems

A parallel deduction system is one in which multiple deduction components run in parallel on separate processors. This is distinct from those deduction systems which run multiple deduction components alternately, such as the unit preference system [Wos, Carlson & Robinson G.A., 1964], and those which are only conceptually parallel systems. Parallel deduction systems can be categorised along three axes.

Homogeneous or Heterogeneous? In a homogeneous system the multiple deduction components all use the same deduction format. Homogeneous systems do not change the search space of the underlying deduction format. Rather they take advantage of added computing power to distribute the work of searching that space. An example of such a system is ROO [Lusk, Slaney & McCune, 1991]. In a heterogeneous system the multiple deduction components use different deduction formats. The advantage of heterogeneity is that each deduction component has a different search space. Further, it is necessary for only one of the components to use a complete deduction format. This feature permits incomplete formats, which are fast on average, to be used as part of complete parallel deduction system. An example of such a system is GLD||UR [Sutcliffe, 1991]. Between homogeneous and heterogeneous are the pseudo heterogeneous systems. The components of these systems all use the same deduction format, but each component has different set-up parameters. Each component will thus have a similar search space, but will search the space in a different manner. An example of such a system is Ertel's Random Competition system [Ertel, 1991].

Common or Separate input sets? To implement a common input set for multiple deduction components, some kind of shared memory architecture is necessary. On the other hand, if each component maintains its own copy of the input set then more common computer architectures can be used. ROO is an example of a system that uses a common input set, while the Random

Competition system is one that uses separate input sets. An approximation of a common input set can be implemented by using separate input sets which are updated in parallel. The approximation can be implemented provided there is some form of inter-component communication mechanism. This approach is taken in GLD||UR.

Homogeneous systems require a common input set, for otherwise the components will duplicate each other's search. In a heterogeneous system it is ok to have separate input sets, as the components do different things naturally. However, there is an advantage to having a common input set in a heterogeneous system. As each component has a different search space, the clauses created in each component may not be created in others. By making all clauses available to all components, 'cross fertilisation' is achieved. Although hard to quantify, this effect has the potential to significantly affect a parallel system's performance. Further, if the parallel update approach is used, the distribution of clauses can be controlled so that different components' sets are updated differently.

Synchronous or Asynchronous? An asynchronous system has the advantage that components do not have to wait for each other, and greater use is made of the available computing power. On the other hand, there may be advantage to be had in making the multiple components aware of and supportive of each others activities.

This paper introduces a heterogeneous parallel deduction system (HPDS), employing a chain format linear deduction component, a UR-deduction component, and a hyper-resolution component. The HPDS is the logical successor of GLD||UR. The components of the HPDS each maintain their own copy of the input set. Each input set is updated with clauses created locally and with clauses created in the other components. The components run asynchronously.

The Parallel Implementation Environment

A prerequisite to the implementation of a parallel deduction system is to decide upon and, if necessary, develop an appropriate parallel programming environment. The HPDS has been implemented using Prolog-D-Linda [Sutcliffe & Pinakis 1991]. Linda is a programming framework of language-independent operators which may be injected into existing programming languages, resulting in new parallel programming languages. Linda permits cooperation between parallel processes by controlling access to a shared data structure called a *tuple space*. Manipulation of a tuple space is only possible using Linda operators (`out`, `in` and `rd`). Parallel execution is provided by an operator (`eval`) which starts new processes. Generically, Prolog-Linda is the extension of Prolog that supports a tuple space and the Linda operators. Prolog-D-Linda is our implementation of Prolog-Linda. Prolog-D-Linda is built on top of SICStus Prolog. It runs on a network of SUN SPARC workstations running SUN OS 4.0.3, connected by Ethernet.

In Prolog-D-Linda the tuple space is distributed (hence "Prolog-Distributed-Linda"). The tuple space and associated operations are implemented in *server* processes. The distribution of tuples across the servers is determined by a user supplied Prolog program. Linda operations in *client*

processes are translated into requests which are passed to the appropriate server. One server is designated the *eval-server*, and is responsible for starting client processes requested in `eval` operations. The servers are started by a *controller* process. After starting the servers, the controller works as the standard input and output device for all the servers, and for clients that are started via an `eval` request. This feature enables servers and clients, that are not associated with a terminal, to have user interaction.

The HPDS Architecture

The deduction components of the HPDS are Guided Linear Deduction (GLD) [Sutcliffe, 1992] (GLD is a chain format linear deduction system), a UR-deduction [Overbeek, McCharen & Wos, 1976] component, and a hyper-resolution [Robinson J.A., 1965] component. Each component maintains its own input set, asynchronously updating it with clauses created in the other components at a time convenient to itself. All of the components have been implemented in Prolog. Common features of the components are :

- All use the chain format for clauses. They all use the same code for their deduction operations, processing of clauses created, and manipulation of their input sets. This makes it easy to distribute the clauses that each creates.
- Each component creates clauses that can be used by the others. All the components apply back and forward subsumption to all clauses that they create. The subsumption checks are applied before the clauses are distributed to the other components. Information indicating which clauses are subsumed accompanies each clause that is distributed.
- Each component uses a consecutively bounded search as its overall search strategy. Although the value bounded is different in each component, the same implementation is used in each component. If a new clause is added to the input set in a search iteration, the the bound value is increased only minimally at the end of the iteration. Further, another iteration is always executed if a new clause has been added to the input set in the iteration. In the HPDS these features are affected not only by clauses created locally, but also by clauses received from the other components.

The GLD Component

GLD is based on Shostak's Graph Construction (GC) procedure [Shostak, 1976], but also incorporates features from other chain format linear deduction systems. In particular, GLD has a combined lemma/C-literal mechanism, which improves on the original separate mechanisms in Loveland's Model Elimination (ME) procedure [Loveland, 1969] and the GC procedure. The lemmas created by this mechanism are the clauses that are distributed to the hyper-resolution and UR components. GLD uses an extended unit preference strategy in all extension operations, and thus productively uses the unit clauses which it receives from the UR component. GLD does not accept clauses created in the hyper-resolution component, as it was found that the large number of clauses created by hyper-resolution caused an unacceptable increase in the size of the GLD search space.

The UR Component

The UR component's strength lies in its ability to solve many Horn problems very quickly. The UR component is a useful adjunct to the complete GLD and hyper-resolution components. The clauses created in the two complete components are used as nuclei in the UR component. This leads to the UR component reporting refutations for problems that it cannot solve independently.

The Hyper-resolution Component

The hyper-resolution component of the HPDS implements positive hyper-resolution. Non-factored hyper-resolvants are distributed to the other components. Like the GLD component, the hyper-resolution component benefits from using the unit clauses produced by the UR component. There is some possibility of an overlap between the clauses created in this component and those created in the GLD and UR components. However, GLD is a back chaining system, while hyper-resolution is forward chaining. Thus the searches of these two components are in the opposite direction. Thus each should benefit from the work that the other does at the other end of the general search space. The overlap with the UR component is dealt with by the subsumption checks.

The Combination

The completeness of the HPDS is assured by the completeness of GLD and hyper-resolution. The difference between these components running independently and running in the HPDS, is the addition of new input clauses created in the other components. Such clauses are subject to the same subsumption checks as those created locally. By having the consecutively bounded search run another iteration whenever any new clause is added to the input set, both the GLD component and the hyper-resolution component remain complete within the HPDS.

The HPDS Implementation

Each deduction component of the HPDS runs as a client process in Prolog-D-Linda. The deduction components do not communicate with each other directly, but rather via a deduction controller (also a Prolog-D-Linda client). The deduction controller is responsible for starting the deduction components, and for controlling the distribution of clauses created in the deduction components. Communication between the deduction components and the controller is via the Prolog-D-Linda tuple space. When a deduction component wishes to pass any data to the controller, it `outs` a tuple of the form `controller(`<source>`,`<type>`,`<data>`)`. The <source> is the name of the deduction component which produced the tuple. The <type> field indicates the type of the <data>. There are four possible <type> values : `clause`, `stop`, `io` and `statistics`. The controller retrieves these tuples from the tuple space using the `in` operation. Similarly, the controller passes data to the deduction components by placing tuples of the form <destination>`(controller,`<type>`,`<data>`)` into the tuple space.

When the deduction controller is started, it collects information about each of the deduction components from the user. This includes the name of component's source code file, the Prolog query required to start the component, and a flag indicating whether or not the component is complete. Each deduction component is then started using the `eval` operation. The task of the controller is then to continuously retrieve its message tuples from the tuple space and to deal with the data according to the value of the <type> field.

The deduction components, as well as executing their deductions appropriately, must check the tuple space at regular intervals for messages from the controller. Again, these messages are dealt with according to their <type>. The GLD component checks for messages after each centre clause is deduced. The UR component checks for messages each time a new nucleus is chosen. The hyper-resolution component checks for messages before each of its deduction operations. This frequent checking in the hyper-resolution component is necessary as in some problems a single nucleus may be in use for a long time. If the checking is reduced to only when a new nucleus is chosen, then the hyper-resolution component does not take sufficient advantage of the clauses created in the other components.

Within each deduction component, each clause that is created is passed to a clause control module in the component. The clause control module implements the subsumption checks, removes subsumed input clauses from the local input set, adds the new clause to the local input set, and transmits both the new clause and subsumed clauses' identifiers to the controller. The <type> of a new clause message is `clause`, and the <data> field contains the new clause and the subsumed clauses' identifiers. When the controller receives a message of this type it forwards the new clause and the subsumed clauses' identifiers to other deduction components, also in a `clause` message. The <source> field enables the controller to selectively forward the <data>, e.g. clauses received from the hyper-resolution component are not forwarded to the GLD component. Upon receipt of `clause` messages, the deduction components remove the subsumed clauses from their input sets, and add in the new clause.

The `stop` message type is used when a deduction component either finds a refutation, or completes the search of its search space. If a deduction component finds a refutation it sends a `stop` message to the controller, with a <data> value of `success`. The successful component then proceeds to terminate. The controller forwards the `stop` message to the other deduction components. Upon receipt of these messages the deduction components proceed to terminate. If a deduction component completes the search of its search space without finding a refutation, it sends a `stop` message to the controller with a <data> value of `failure`. If the failing component has a complete deduction format then it proceeds to terminate, and the controller forwards the `stop` message to the other deduction components. If the failing component is incomplete, then the `stop` message is ignored by the controller as a refutation could still be found. The failing component does not terminate is this situation, as it could receive a clause from another deduction component hence opening up another piece of search space. Therefore incomplete deduction components, that have completed their search, wait for further messages from the controller.

The `io` message type allows the deduction components to output data at the controller's terminal. This is useful for tracking the progress of the HPDS. Whenever the controller receives an `io` message, the <data> is printed to its standard output.

When a deduction component proceeds to terminate, it determines the number of deduction operations it has performed and the amount of CPU time it has used. These figures are sent to the controller in a `statistics` message. After the controller has sent a `stop` message to each deduction component, it collects the `statistics` messages and outputs appropriate statistics. The final task of the controller after this is to remove any messages left in the tuple space.

Testing

To evaluate a parallel deduction system, it is important "to state UNAMBIGUOUSLY ... what one wants to measure" [Suttner, 1992]. For the HPDS, the testing has measured :

- The amount of *processing* the HPDS performs to find refutations. Processing is measured in terms of CPU Units, where one CPU Unit is the amount of processing done on one processor in one unit of time.
- The amount of *time* the HPDS uses to find refutations. Time is measured in seconds.

The processing and time usages of the HPDS have been compared with those of five other deduction systems. They are :

1. The GLD component running independently.
2. The UR component running independently.
3. The hyper-resolution component running independently.
4. A naive version of the HPDS, in which all the components are run time sliced on a single processor. The only inter-component communication in the naive system is to stop all components as soon as any one finds a refutation. The processing and time usages of the naive system are calculated from from those of the independent components.
5. An 'average system', whose processing and time usages are the averages of those of the independent components.

The HPDS and the independent components have been tested on 132 problems. Due to the large number of problems, a time limit of 500 seconds was imposed on all tests. In 49 of the problems none of these systems found a refutation within the time limit. The results for the remaining 83 problems have been analysed. For each of the five systems listed above, the ratios of its and the HPDS's processing and time usages have been determined for each problem. The ratio of the processing usages is called the *productivity*. The ratio of the times taken is called the *speed-up*. Two speed-up ratios are considered. The first ratio compares the HPDS with the other system running on a processor whose processing capability combines the three processors used by the HPDS. The second ratio compares the HPDS with the other system running on one of the processors used by the HPDS. The first ratio measures speed-up due to strategy shift, while the second measures speed-up due to parallelism and strategy shift. In all cases a value

greater than 1 indicates that the HPDS is performing better than the other system. The results are summarised in the following table.

Measure	Compared With				
	GLD	UR	Hyper	Naive	Av.
HPDS & ~2nd	9	23	25	3	3
HPDS & 2nd	63	49	47	69	69
~HPDS & 2nd	9	7	1	11	11
Av. Prod'y	1.60	0.96	7.83	1.13	2.05
# Prod'y > 1	15	4	12	12	12
Av. SU(1)	0.98	0.48	4.80	0.64	1.24
# SU(1) > 1	10	3	5	8	9
Av. SU(2)	2.95	1.49	14.4	1.93	3.71
# SU(2) > 1	26	8	15	39	14

Legend

HPDS & ~2nd : The number of problems for which the HPDS found a refutation and the other system did not.
HPDS & 2nd : The number of problems for which both the HPDS and the other system found a refutation.
~HPDS & 2nd : The number of problems for which the other system found a refutation and the HPDS did not.
Av. Prod'y : The average productivity, measured over the problems for which both the HPDS and the other system found a refutation.
Prod'y > 1 : The number of problems for which the productivity is greater than 1.
Av. SU(1) : The average speed-up, measured over the problems for which both systems found a refutation, with the other system running on a combined processor.
SU(1) > 1 : The number of problems for which the speed-up(1) is greater than 1.
Av. SU(2) : The average speed-up, measured over the problems for which both systems found a refutation, with the other system running on a single processor.
SU(2) > 1 : The number of problems for which the speed-up(2) is greater than 1.

Performance Comparison Table

The HPDS finds refutations for 72 of the 83 problems analysed. The GLD component finds the refutation in 40 of the 72 problems, the UR component in 28 problems, and the hyper-resolution component finds only 4 refutations. The unit clauses produced by the UR component are believed to be a major contributing factor to the strong performance of the GLD component in the HPDS. Conversely, the clauses passed to the UR component in the HPDS enable the UR component to find the refutation in six problems which it cannot solve independently. This is clear evidence of fertilisation from the GLD and hyper-resolution components. The hyper-resolution component appears to contribute the least to the HPDS. The role that the hyper-resolution component may be playing is to improve the performance of the UR component by supplying a large number of nuclei. If this is the case, and it is also true that the UR component's unit clauses are benefiting the GLD component, then the hyper-resolution component plays an integral role in the HPDS. A version of the HPDS, in which only the GLD and UR components are used, is being tested to determine the extent to which this is true.

The figures above show that the HPDS is more productive than the other systems. The less than 1 average productivity when compared to UR deduction, is offset by the 23 problems for which

UR deduction does not find a refutation. In terms of speed-up, the HPDS is faster than all the other systems when they are run on a single processor (speed-up(2)), as should be expected. If the comparison 'plays fair', and provides the same amount of processing power to both the HPDS and the other systems (speed-up(1)), then the HPDS is less dominant. In particular, the naive system performs better. A speed-up(1) value greater than 1 is roughly equivalent to super-linear speed-up, as it is commonly defined. Thus the HPDS displays super-linear speed-ups against each of the other systems, for some problems. Again, all the speed-up figures should be viewed in conjunction with the number of problems for which the other system does not find a refutation.

Conclusion

The HPDS is one of a large class of parallel deduction systems whose architecture is identified by the relative independence of the deduction components that run in parallel. The current version of HPDS is an early development in this class of systems, and there is clearly scope for further investigation. Many questions concerning appropriate combinations of components have yet to be addressed. The Prolog-D-Linda environment makes it possible to quickly and easily build and evaluate combinations of components. Experiments with different combinations of components are, to a large extent, unhampered by the difficulty of determining the compatibility of the individual deduction formats. This is in contrast to the difficulties experienced when combining multiple refinements of resolution into a single deduction system.

Besides experiments with different deduction component combinations, there is also scope to increase the level of cooperation between the component of the HPDS. In the current configuration the deduction components are in cooperative competition. However, the cooperation consists of simply distributing clauses without knowledge of whether the clauses will benefit the recipients. In an improved HPDS, the deduction components would cooperate reactively and proactively with each other. A deduction component with particular skills could be requested to try create a particular clause, by another component without those skills. The deduction components would also be more aware of their own needs, and would carefully filter clauses created in other components. The current HPDS has only one instance of this type of filtering, where the GLD component does not accept clauses from the hyper-resolution component. The performance of the HPDS was significantly worse before this filter was added. Closer cooperation would also produce more stable results. The performance of the current HPDS is affected by very small changes in the timing of clause distribution, brought on by varying processor loads.

Finally, the strong performance of the naive system, relative to the HPDS, suggests that heterogeneous competitive systems are worth investigating.

References

Ertel W. (1990), Random Competition: A Simple, but Efficient Method for Paralleizing Inference Systems, 342/27/90 A, Institut fur Informatik, Technische Universitat Munchen, Munich, Germany.

Loveland D.W. (1969a), A Simplified Format for the Model Elimination Theorem-Proving Procedure, In *Journal of the ACM* 16(3), ACM Press, New York, NY, 349-363.

Lusk E.L., Slaney J.K., and McCune W.W. (1991), ROO, In Kanal L.N., Suttner C. B. (Ed.), *Informal Proceedings of PPAI-91, International Workshop on Parallel Processing for Artificial Intelligence* (Sydney, Australia, 1991), International Joint Conferences on Artificial Intelligence, Inc., Sydney, Australia, 110-116.

Overbeek R., McCharen J., and Wos L. (1976), Complexity and Related Enhancements for Automated Theorem-Proving Programs, In *Computers and Mathematics with Applications* 2, Pergamon Press, England, 1-16.

Robinson J.A. (1965b), Automatic Deduction with Hyper-resolution, In *International Journal of Computer Mathematics* 1, Gordon and Breach, London, England, 227-234.

Shostak R.E. (1976), Refutation Graphs, In *Artificial Intelligence* 7, Elsevier Science, Amsterdam, The Netherlands, 51-64.

Sutcliffe G. (1991), A Parallel Linear and UR-Derivation System, In Kanal L.N., Suttner C. B. (Ed.), *Informal Proceedings of PPAI-91, International Workshop on Parallel Processing for Artificial Intelligence* (Sydney, Australia, 1991), International Joint Conferences on Artificial Intelligence, Inc., Sydney, Australia, 211-215.

Sutcliffe G. (1992), The Semantically Guided Linear Deduction System, In Kapur, D. (Ed.), *Proceedings of the 11th International Conference on Automated Deduction* (Saratoga Springs, NY, 1992), Springer-Verlag, New York, NY,

Sutcliffe G., and Pinakis J. (1991), Prolog-D-Linda : An Embedding of Linda in SICStus Prolog, 91/7, Department of Computer Science, The University of Western Australia, Perth, Australia.

Suttner C.B. (1992), Personal correspondence about evaluating parallel deduction systems.

Wos L., Carson D., and Robinson G.A. (1964), The Unit Preference Strategy in Theorem Proving, In *Proceedings of the AFIPS 1964 Fall Joint Computer Conference* (San Francisco, CA, 1964), Spartan Books, Baltimore, MD, 615-621.

A Functional Languaje for Parallel Automatic Deduction

Carlos Araya
Center for Research on Computation
Costa Rican Institute of Technology
caraya@huracan.cr

1. Introduction

This paper is about a language for programming arbitrary deduction methods suitable for Theorem Proving and Derivation procedures. The system pursues deduction as a rewriting strategy under equivalence preservation and it is part of an investigation that looks for answers to the following questions:

- What are the operations required for deduction?

- How can deductive knowledge be represented?

- What evaluation strategy can animate deduction in computers?

- What is the relation between deduction and computation?

The proposal is a new parallelizable functional programming language – Schemata [Araya90, Brown90e] – as the embodier of the answers to these questions. Schemata comprises most of the Lisp language Scheme [Rees86] and inherits from Symeval [Brown86b] the notion that deduction is a rewriting process. It has also been influenced by Prolog [Colmerauer82] and Snobol [Griswold85].

This paper is organized as follows. The second section enumerates the conceptual basis of the Schemata Language. The third section provides examples of the language application. Finally, the relevance of the language for parallel programming is succinctly exposed in the fourth section, the conclusions.

2. Main Characteristics

The following are the most outstanding characteristics of the Schemata Language:

a) The evaluation of an expression produces multiple solutions. Accordingly, symbols can have associated zero or more values which can be processed concurrently.

b) Schemata considers lambda abstractions as high level objects which rewrite expressions preserving equivalence in the Lambda Calculi spirit of Church [Church41]. Consequently, Schemata assumes equivalence between inputs and outputs.

d) By associating to symbols rewriting rules which represent the properties of the symbols, Schemata permits the axiomatization of arbitrary deduction systems. Therefore, Schemata is independent of any logic.

e) The association of the symbol to itself enables the reflection of the normal form of the symbol. Consequently, the values returned by Schemata are expressions of the same language and therefore there is no difference between programs and data structures.

f) The application of lambda abstractions is discriminated by pattern matching of the lambda argument expression against the input arguments. Consequently, Schemata execution is driven by pattern matching over expressions of the same language. In addition, the application of a symbol to argument tuples is considered as the application of each one of the symbol values to each one of the argument tuples using a diagonalization mechanism.

g) Two pattern matching operators are proposed for horizontal and vertical search – ellipses and schemators. Ellipses are for pattern matching what argument segments are for application. Schemators [Morse65] are for pattern matching what lambda-abstractions are for evaluation. The pattern matching of both primitives is highly parallelizable.

h) Schemata is a language with all its backtracking and pattern matching mechanism based on streams [Wadler85, Abelson85]. Hence, Schemata provides a very flexible, controllable, and parallelizable search strategy.

i) Streams provide a direct representation for both failure and multiple solutions [Wadler85]. Consequently, Schemata possesses nondeterminism features similar to the ones exposed by [Henderson80]. This characteristic reflects the language natural suitability for parallel execution.

3. Language Applications

So far, the more remarkable application of Schemata is a system for Nonmonotonic Reasoning (NRS) developed by the Brown and Araya [Brown90a]. The NRS performs deduction on the Z Modal Logic [Brown86a, Brown87, Park88] and can solve problems in seven different theories of nonmonotonicity including Autoepistemic Logic [Moore85, Konolige87], Nonconstructive Default Logic [Brown89a], Default Logic [Reiter80] and its extensions, the

Closed World Assumption [Brown89a], McCarthy's Parallel Circumscription [McCarthy80], and Levesque's BNO [Leveque84]. In [Brown90c, Brown90d], Brown and Araya show some practical applications of this system in the areas of Physics, Diagnosis, and Deontic Reasoning. Other practical applications of the NRS were done in Situation, Event Calculus and Assembling [Missiaen89], in Manufacturing [Park89], and in the Frame Planning [Brown90b].

The following examples provide more information on the language. An intuitionistic disjunction is specified in Schemata by associating to the symbol two rules:

```
(define  (i∨ x y) x)
(define  (i∨ x y) y)
```

Therefore, the application of i∨ to a pair of arguments will generate two different independent computations, one for each argument. For example,

```
=> (i∨ 1 2)
1
2
=>
```

For Nonmonotonic Reasoning, the NRS solves reflective equations of the form $K \equiv \Pi_K$ expressed in Z, where Π_K contains occurrences of K under the modal symbol [] and ≡ represents synonymity in the sense that $K \equiv \Pi_K$ is defined as $[](K \leftrightarrow \Pi_K)$. Therefore, K is the knowledge base with contents synonymous to Π_K. The derelativization from K of equations of this form is any solution Ø, where K does not appear in Ø, such that $Ø \equiv \Pi_Ø$, ie. $Ø \leftrightarrow \Pi_Ø$ is a tautology. Solutions to these equations are obtained by deriving from $K \equiv \Pi_K$ a disjunction of expressions of the form $K \equiv \Pi_{Øi}$ using deduction steps under equivalence preservation.

In Schemata, this concept is generated by case analysis in the following way:

```
(define  (≡ K (Π1 ('[] (Π2 K))))
  (i∨  (∧    (≡ K (Π1 T))        ([]  (Π2  K)))
       (∧    (≡ K (Π1 F))        (~   ([]  (Π2  K))))))
```

This example shows two Schemata pattern matching primitives: schemators or higher-order matchers indicated by Πi, and constants introduced by quotation. In addition, it is noticed that the disjunction of solutions is going to generate two different computations. If there is a total of n different modal occurrences of K in the original expression, the recursive application of the rule generates a total of 2^n different solutions. To finally resolve the equation into its

solutions, other NRS rules substitute back any solution into the modalized conditions conjuncted by case analysis and the expression is simplified.

The following example shed some light on the implementation of some common concepts on parallel computation. The first definition associates to the FAIL symbol the empty set of solutions since the definiens does not have a match.

```
(define FAIL (iv))
```

Using FAIL and the PRIME? predicate in the following example, we screen out the prime numbers from the collection of natural numbers. The first two definitions declare what the natural numbers are whereas the last definition filters for primes.

```
(define (natural n) n)
(define (natural n) (natural (+ n 1)))

(define (primes x) (if (prime? x) x FAIL))
```

```
=> (primes (natural 2))
2
3
5
7
...
```

As it can be observed, the infinite computation being performed for the natural numbers is fed into the PRIMES procedure which either returns the number or failure depending on whether each individual number is or is not prime.

4. Conclusions

A concise summary of the current research related to Schemata parallelizable characteristics has been presented. Using an implementation of intuitionistic disjunction, it has been shown how to generate the multiple alternative solutions required for nonmonotonic reasoning under the NRS system.

In addition, this systems demonstrates that functional languages can improve on parallel architectures if multiple definitions, and possibly a good way of discriminating application, are integrated into the language. Furthermore, this paper also shows that efficient parallel functional programming can be easily performed without burdening the user with issues related to when and how parallel capabilities are suitable. Finally, it can be claimed that this

programming style is more natural because it is closer to way in which we have been programming computers.

So far, Schemata has been implemented in several platforms including Sun, Macintosh, and Symbolics computers. There is a commercial version available of the system by AIR, Inc. At CRC, we are currently working on the first parallel prototype of the system which will probably run first on transputers, but we are actively analyzing other architectures.

References

Abelson85. Abelson, H. and Sussman, G. J., *Structure and Interpretation of Computer Programs* (MIT Press, Cambridge, MA, 1985.)

Araya90. Araya, C. and Brown, F.M, SCHEMATA: A Language for Deduction, *Proc. of the European Conference on Artificial Intelligence - 1990*, Stockholm, Sweden, August (1990.)

Brown86a. Brown, F. M., A Commonsense Theory of Nonmonotonic Reasoning, *Proceedings of the 8th International Conference on Automated Deduction*, Oxford, England (July 1986.) *Lecture Notes in Computer Science* **230** (Springer-Verlag, 1986.)

Brown86b. Brown, F. M., An Experimental Logic Based on the Fundamental Deduction Principle, *Artificial Intelligence* **30** (1986) 117-263.

Brown87. Brown, F. M., A Modal Logic for the Representation of Knowledge, *The Frame Problem in AI, Proceedings of the 1987 AAAI Workshop* (Morgan Kaufmann, Los Altos, CA 1987) 135-157.

Brown89a. Brown, F. M., The Modal Quantificational Logic Z, A Monotonic Theory of Nonmonotonic Reasoning, in *Proc. of the First International Workshop on Human and Machine Cognition*, Ed. P. Hayes and K. Ford, University of West Florida (1989.)

Brown90a. Brown, F.M. and Araya, C., A Deductive System for Theories of Nonmonotonic Reasoning, *Eighth Army Conference on Applied Mathematics and Computing*, Cornell University, Ithaca, New York (1990.)

Brown90b. Brown, F., Hundal, S. and Araya, C. Frame Planner, *Procs. Florida AI Research Symposium*, Florida (1990.)

Brown90c. Brown, F.M. and Araya, C., Applications of Equation Solving in a Cylindric Algebra, *Proceedings of the Third International Symposium on AI*, Monterrey, Mexico (1990)

Brown90d. Brown, F. and Araya, C., Cylindric Algebra Equation Solver, abstract in *Procs. 10ti International Conference on Automated Deduction*, Kaiserslautern, FR. Germany, Lecture Notes in Artificial Intelligence 449, Springer-Verlag, Germany (1990)

Brown90e. Brown, F. and Araya, C., Schemata, abstract in *Procs. 10th International Conferenc on Automated Deduction*, Kaiserslautern, FR. Germany, Lecture Notes in Artificial Intelligence 449, Springer-Verlag, Germany (1990)

Colmerauer82. Colmerauer, A., Prolog II - Manuel de Reference et Modele Theorique, Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille II (1982.)

Church41. Church, A., The Calculi of Lambda-Conversion, *Annals of Mathematical Studies*, N. 6 (Princeton University Press, 1941.)

Griswold85. Griswold, R. E., The Control of Searching and Backtracking in String Pattern Matching, *Implementations of PROLOG*, ed. J. A. Campbell (Ellis Horwood Limited 1985.)

Henderson80. Henderson, P., *Functional Programming: Application and Implementation* (Prentice-Hall, London, 1980.)

Konolige87. Konolige, K., On the Relation Between Default Theories and Autoepistemic Logic", *Proc. of IJCAI87* (Morgan Kaufmann, 1987.)

McCarthy80. McCarthy, J , Circumscription -- A Form of Nonmonotonic Reasoning, *Artificial Intelligence* **13** (1980.)

Missiaen89. Missiaen, Lode, Situation Calculus and Event Calculus in Modal Logic Z, Dept. of Computer Science Report CW85, Katholieke Universiteit Leuven, Leuven, Belgium (1989.)

Moore85. Moore, R., Semantical Considerations on Nonmonotonic Logic, *Artificial Intelligence* **25** (1985.)

Morse65. Morse, A. P., *A Theory of Sets* (Academic Press, 1965.)

Park88. Park, S., On Formalizing Commonsense Reasoning Using the Modal Situation Logic and Reflective Reasoning, PhD. Thesis, Dept. of Computer Science, University of Texas at Austin (1988.)

Park89. Park, S., Araya, C., Brown, F., and Hundal, S., Automated Nonmonotonic Reasoning and its Applications in Manufacturing, *4th Int. Conf. on CAD/CAM, Robotics and Factories of the Future*, Indian Institute of Technology, New Delhi (1989.)

Rees86. Rees, J. and Clinger, W., eds., Revised Report on the Algorithmic Language Scheme, MIT AI Memo 848a (1986.)

Reiter80. Reiter,R., "A Logic for Default Reasoning, *Artificial Intelligence* **13** (1980.)

Wadler85. Wadler, P., How to Replace Failure by a List of Successes, in *Functional Programming Languages and Computer Architecture*, ed. J. -P. Jouannaud, *Lecture Notes in Computer Science* **210** (Springer-Verlag, 1985.)

The Improvement of A Parallel Theorem Prover Based on The Model Generation Method

Tetsuji KUBOYAMA, Akira NISHIMOTO,
Rin-ichiro TANIGUCHI and Makoto AMAMIYA

Department of Information Systems, Kyushu University 6-1,
Kasuga-koen, Kasuga-shi, Fukuoka 816 JAPAN
email: kuboyama@is.kyushu-u.ac.jp

abstract

In recent years, attention has come to be paid to the study on automated theorem proving again, and many automated theorem provers are being proposed. One of them is SATCHMO using Prolog technology, which is based on the model generation method. This method has the following feature:

- This method is based on forward reasoning.
- Full unification with occurs check is not necessary.
- The proving processes are viewed as two parts, model generation and check on its satisfiability.

These feature are useful for us to incorporate various strategies and to implement provers on computers. In ICOT, the SATCHMO-based prover, MGTP, which uses KL1 technology, has been already built and its improvement is being carried out. The purpose of this paper is to point out inefficiency in proving processes of MGTP and to propose an improvement of MGTP. Redundancies reside in proving processes:

- A conjunctive matching of the antecedent literals against each element in a model candidate is redundant.
- Redundant OR-branches are generated in the proof tree. For example, the number of the redundant OR-branches, in the worst case, is equal to the permutation of the clauses for which conjunctive matching can be applied.

As a first step in removing the redundancies, we pay attention to the redundant OR-branches. Then we propose the method to avoid this redundancy by the following method. When there are more than one clause whose antecedents are same, a model candidate may branch into multiple ones in normal MGTP. Whereas in OR-branch restraint method(ORM), a model candidate is not branch in this case, namely, model extension is applied to the one model candidate. This method is effective in improving performance in case that the clauses have same antecedents. There exist, however, some cases in which redundant AND-branches are generated. However, it is possible to eliminate the clauses which cause this redundancy, by pre-processing a given set of clauses. The processing which is the connection analysis among clauses at the level of predicate letters enable the redundant literals to eliminate. We implement a MGTP interpreter with ORM(MGTP-ORM) in KL1, and compare the performance of MGTP-ORM with the one of a normal MGTP interpreter(MGTP-N). We show several experimental results for various sets of clauses with the MGTP-ORM interpreter. For example, we use the 5-queen problem and then MGTP-ORM is cut down on the number of reduction more than 90with all-solution mode. On the other hand, in the set of clauses including the literals in the consequents which have no connection with the ones in the antecedents, MGTP-ORM may be less effective than MGTP-S. However, applying the pre-process to the set of clauses, then the performance of MGTP-ORM almost results in the same as the performance of MGTP-N.

Improving Backward Execution in Non-Deterministic Concurrent Logic Languages

Salvador Abreu[*]
Luís Moniz Pereira
Departamento de Informática
Universidade Nova de Lisboa
2825 Monte de Caparica
PORTUGAL
spa,lmp@fct.unl.pt

Philippe Codognet
INRIA-Rocquencourt
BP 105, 78153 Le Chesnay
FRANCE
Philippe.Codognet@inria.fr

Abstract

We present a new mechanism for improving the execution of non-deterministic concurrent logic languages, such as the Andorra family of languages. The basic idea is, upon failure and backtracking, to re-schedule continuation goals in order to first execute the goal that has failed. In this way, if the backtrack point is not pertinent to the failure, the original failing goal will fail again and this will immediately amount to further deeper backtracking. Such a heuristic will hence save useless deduction/backtracking work. This method would be very difficult to implement in sequential stack-based logic systems such as Prolog, but fits well in the execution models of concurrent logic languages. We have implemented this scheme by modifying the Andorra Kernel Language prototype implementation developed at SICS (AKL version 0.0), and evaluation results show that this backward execution strategy improves performance for a variety of benchmarks, giving speedups up to a factor two or three.

1 Introduction

Since the first research in the early 80's, the field of concurrent logic programming has raised a number of important issues about the way of executing logic programs and how to move away from the traditional, simple but rigid, search strategy popularized by Prolog. On the one hand, the depth-first left-to-right strategy of Prolog, due to its simple stack-based execution model, has lead to tremendous advances in Prolog implementation technology which has been decisive for establishing logic programming as a mature declarative paradigm. Such results were exported to the area of automated deduction with theorem provers such as Stickel's PTTP. But on the other hand, one may sometimes prefer a more flexible control strategy, and a more adequate search strategy that reduces the overall number of inferences needed to solve the problem can make up for a slower inference speed. This is especially true for theorem proving and automated deduction applications, were the control may be as important as the inference speed in order to achieve good performances. In that direction concurrent logic languages such as KL1, Concurrent Prolog or Parlog are a step forward to escape from the rigid depth first search strategy. Moreover, some implementations of concurrent languages now achieve raw speeds close to that of traditional Prolog system.

The *Andorra model* was proposed by D.H.D. Warren [15] in order to combine Or- and And-parallelism, and it has now bred a variety of idioms and extensions developed by different research groups, among which Andorra-I [3, 2] and Andorra Kernel Language [5, 6]. The essential idea is to execute determinate goals first and concurrently, delaying thus the execution of non-determinate goals until no determinate goal can proceed any more. Hence the Andorra principle can amount to *a priori* pruning and it can thus reduce the size of the computation when compared to standard Prolog up to one order of magnitude [2].

Although much work has been done to improve the forward execution in deductive systems, as exemplified by the Andorra model described above, very few works deal with the improvement of backward

[*] This work was done while at INRIA on an extended visit.

execution. It is however interesting, upon failure during the computation process, to take into account the information of this very failure to improve the search strategy. For instance, one can consider that the goal that has currently failed is likely to fail again after backtracking, that is, if the backtrack point is not pertinent. This goal should thus be executed first after backtracking, as this could immediately amount to further deeper backtracking if the backtrack point is not pertinent. In this way, the whole recomputation of the part of the proof between the backtracking point and the failing goal is avoided w.r.t a standard (non-reordering) strategy. The point being that if this goal just failed, it is likely to be "problematic" and therefore it may fail again with the next bindings produced as a result of backtracking. In other words, *a failed goal is likely to fail again*, this can indeed be seen as yet another instance of the "first-fail" principle apply to backward execution.

Such a re-ordering of the continuation goals (goals that remain in the resolvent) upon backtracking, based only on the information of which is the current failing goal, can thus save much deduction/backtracking work. This simple idea has indeed been first proposed by Lee Naish for some years[12] as an example of heterogeneous SLD resolution. A related computation rule is presented, and it is argued that it is a form of intelligent backtracking. However, such a strategy is not well suited to the computational model of Prolog as implemented for example by stack-based architectures such as the famous Warren Abstract Machine (WAM) because a failing goal is immediately discarded, leaving no trace, and it is very hard if not impossible to relate it to its other occurrences, which will be tried anew after the last choice-point's goal has generated another solution. There is also no way to manipulate and keep part of the stacks to avoid recomputing independent parts of the proof. Also remark that Prolog's operational semantics requires a strict execution order which prevent such re-ordering. One had to wait until the development of concurrent logic languages to fully rework those ideas. Indeed at the semantic level these languages have the necessary flexibility and do not impose a strict ordering, and at the implementation level they include sufficiently versatile data-structures to make re-ordering (re-scheduling) easy, as, in one way or an other, the whole computation tree has to be explicitly represented.

We have implemented this scheme in the prototype version of the Andorra Kernel Language (AKL 0.0) developed at SICS [7]. As mentioned above, one of the main interest of the proposed scheme is its ease of implementation by simple modifications of an existing implementation of a concurrent language. One has only to integrate a re-scheduling of continuation goals upon backtracking. There is no need to modify terms or variable bindings representations as in an intelligent backtracking scheme. The overhead of our machinery w.r.t. the original AKL implementation is very small, limited to 5-10%. We have experimented with various strategies on a variety of benchmarks, and we have encouraging results: a average gain of 50% (as measured by the number of nondeterminate promotions) can be observed, with peak values up to 300%.

2 Andorra Kernel Language

The semantics of the Andorra Kernel Language (AKL) is given by a set of rewrite rules applicable to and-or trees designated as *configurations*, see [5]. We will follow in this paper the AKL terminology for and-or trees: a subtree whose root is an or-node will be called a choice-box, and a subtree whose root is an and-node will be called a and-box.

The current (sequential) implementation of AKL [7] uses copying for the *choice splitting* operation, as decribed by the semantic rewrite rule. This operation is very different from the run-time organization of, say, Prolog systems where choice splitting, i.e. non-determinism is handled by choice-point creation and backtracking. In the AKL scheme, there are no choice-points, the concept being replaced by the multiple clauses of the nondeterminate goal being present in otherwise similar and-box contexts, contained in a common choice-box. This leads us to define the notion of copied or re-incident node as follows.
A *re-incident node* G' is another instance of a choice-box or and-continuation for a goal G that occurs in another branch, at the same depth in the and-or tree. In other words, given configuration (1),

$$\mathbf{choice}(\cdots, \mathbf{and}(S_1, \mathbf{choice}(\cdots), S_2), \cdots) \tag{1}$$

after applying the choice-splitting rule we obtain configuration (2).

$$\mathbf{choice}(\cdots, \mathbf{and}(S_1, \mathbf{choice}(\cdots), S_2), \mathbf{and}(S_1', \mathbf{choice}(\cdots), S_2'), \cdots) \tag{2}$$

S_1' and S_2' are *re-incident nodes* of respectively S_1 and S_2, as they share the same potential solutions, represented by an identical set of applicable clauses.

3 And-Or tree transformations based on failure

Our backward execution method will thus be formalized as an and-or tree transformation executed upon failure. Intuitively, the and-or tree transformations can be viewed as a heuristic to speed up the search space scanning process by providing a means by which separate or-branches of the proof tree influence one another: one branch's early failure may be used to "draw attention" to the part of the tree that caused failure, so that it will be selected earlier than it normally would in other instances present in other parts of the tree.

We experimented with different strategies for reordering the and-or tree, that can be presented under several orthogonal aspects, as described below.

1. **Under what circumstances to reorder?**

   These are conditions that may have to be fulfilled in order for the modifications to the shape of the and-or tree to take place. Different conditions are enumerated and may have to be satisfied independently of each other.

   (a) *When the failing node is an instance of the last clause for the parent goal.*
   This corresponds to reorder only in case of deep backtracking.

   (b) *When the re-incident nodes have the same grandparent choice-box*[1] *as the failing node.*
   The effect of this is to further narrow the scope of the reordering operation.

2. **What parts of the tree to reorder?**

   (a) *Reorder only the re-incident nodes.*

   (b) *Reorder all the nodes in a path from each re-incident node to the lowest common ancestor.*
   Using this criterion will make more extensive changes to the and-or tree, as it calls attention not only to a specific node's other instances, but also to the computation that gave rise to its failure.

3. **Where to reorder to?**

   If a a failure occurs and a given subset of the conditions specified in 1 are fulfilled, the current and or tree (the AKL configuration) should be rewritten. The transformation will only affect the re-incident nodes of the failing literal, which will be moved. The possibilities we have considered are:

   (a) One interesting location to move a re-incident node to, assuming it failed because of the binding of a variable X, is right after X's producer. This is the optimal place to move it to, however it would require some form of variable binding dependency analysis, which puts this outside the immediate scope of this work.

   (b) *"Bring To Front"*. This method takes all the re-incident nodes and brings them to the front of their sibling choice-boxes (or and-continuations). It can be very efficient for some rather poorly-written programs, but it easily leads to bad results because of its "randomizing" effect, as we verified experimentally. This is also the simplest method to implement efficiently.

   (c) *"Incremental"*. Instead of bringing the re incident nodes to the front it moves them to the left by one place. The intuition behind this method is that it will incrementally approximate the place where it should go to as defined by (3a), until it *passes over* it, after which calls to the goal will suspend, waiting for the producer goal to effect its binding. This method should be more "progressive" than method (3b), and will hopefully provide a better approximation of a monotone gain in performance.

4 Statistics

We have implemented the method, with variuos reordering strategies, in the AKL 0.0 prototype implementation developped at SICS [7]. This implementation is not as efficient as state-of-the-art Prolog

[1] The grandparent choice-box is the parent choice-box of the containing and-box, ie. the closest enclosing choice-box.

compilers such as Sicstus Prolog, being 3 to 4 times slower. We believe however that the results reported here can be extrapolated to a more efficient implementation as the method is quite independent of the inference engine. For this purpose, we give as measurement of a reordering strategy's performance the number of choice splits (ie. nondeterminate promotions) performed, as this is probably the most expensive operation in AKL. It corresponds to choice-point creation in normal Prolog. This numbers are thus independent of the underlying inference engine raw speed. The results for mesurements in cpu time and in overall number of inferences are similar to those in number of choice split, and are not included for this reason.

The overhead introduced by the need to maintain the structures necessary for the reordering method results in a slowdown in the order of 5 to 10% over the un-modified AKL implementation, depending on the programs.

Table 1 depicts the relative performance of the various methods we described previously, the labels on the columns stand for:

- `btf`. Bring to front. Reorder when: (1a). What: (2a). Where to: (3b).
- `btf/gp`. Bring to front, same grandparent. Reorder when: (1a) and (1b). What: (2a). Where to: (3b).
- `btf/t`. Bring to front, tree. Reorder when: (1a). What: (2b). Where to: (3b).
- `sbo`. Step by one. Reorder when: (1a). What: (2a). Where to: (3c).
- `sbo/gp`. Step by one, same grandparent. Reorder when: (1a) and (1b). What: (2a). Where to: (3c).
- `sbo/t`. Step by one, tree. Reorder when: (1a). What: (2b). Where to: (3c).

Table 1 displays the performance of these methods relative to the un-modified AKL strategy, as a ratio where numbers greater than 1 represent a speedup (ie. a lower number of choice split operations).

	Method					
Benchmark	btf	btf/gp	btf/t	sbo	sbo/gp	sbo/t
circuit	1.38	1.38	1.38	1.00	1.00	1.00
bagof-circuit	2.91	2.91	2.91	1.19	1.28	1.52
color-13-good	0.56	0.68	0.56	0.26	0.91	0.27
color-13-bad	0.74	0.97	0.74	0.34	1.71	0.46
example-g	2.75	2.75	2.75	2.75	2.75	2.75
crypt	1.00	1.00	1.00	1.00	1.00	1.00
knights-5	0.97	1.00	0.97	0.97	1.00	0.97
queens-8	2.09	1.53	2.09	1.05	1.00	1.05
ham	0.90	1.00	0.95	1.00	1.00	1.00
blocks		1.20		1.97	1.00	1.97
money	1.00	1.00	1.00	1.00	1.00	1.00
bagof-salt-mustard	1.86	1.00	2.01	1.20	1.00	1.33
zebra	2.82	2.54	2.82	1.84	1.80	1.78

Table 1: ratios of Number of Choice Splits (unmodified/modified AKL)

`bagof-circuit` generates all solutions to a digital circuit fault diagnosis problem from [11], `zebra` is a version of the "houses" problem, `crypt` is adapted from [17], `money` is the "send+more=money" crypt-arithmetic puzzle, `example-g` is the example previously discussed, `bagof-salt-mustard` is the "salt-and-mustard" puzzle, `color-13-good` and `color-13-bad` are the traditional map-coloring puzzle from [1] with both a favorable and a bad goal ordering, `ham` and `bagof-ham` consist in finding a Hamiltonian path through a graph, and `blocks` is a simple planning problem in the blocks-world.

The overall best method seems to be the `btf` family, the fastest ones being `btf` and `btf/t`, as they obtain an average 60% gain in performance on these benchmarks, with peak speedups up to a factor three. Some remarks concerning the strategies on this data:

1. The methods that reorder the entire sub-tree leading to the re-incident nodes (names ending in `/t`) perform almost identically to those that only reorder the re-incident nodes. This may be attributable to the fact that the and-or trees represented by the AKL configurations encountered during these benchmarks are not very deep.

2. The "incremental" method (`sbo`) is more stable than the "bring to front" (`btf`) one. In practice this means that the gains are not as high, however, there are not as many pathological cases.

3. The "same grandparent" restricting heuristic (`lh`, names ending in `/gp`) seems to restrict the gains a little bit, however and most interestingly it prevents some of the truly pathological behaviour of the strategy it's being used with.

4. The `blocks` benchmark with the `btf` and `btf/t` heuristics took too long to complete so the figures are not in the table.

5 Conclusion and further work

We have shown that it is feasible to apply simple search-space pruning heuristics to improve the non-deterministic behaviour in the Andorra-based concurrent logic Language. The experimental results are encouraging and the methods we have described are susceptible of being implemented efficiently and with few modifications to the original engine.

Hence we believe that *a priori* pruning, such as the one provided by the Andorra principle, should be combined with a posteriori pruning, based on failure information, to achieve the best results.

These results can be immediately applied to automated deduction applications, which are both in demand for non-determinism and for control of the search space, as proposed by nondeterministic concurrent logic languages. This is an example of the synergy between research on parallel implementations (concurrent logic languages) research on deduction systems (reordering method based on failure).

References

[1] M. Bruynooghe and L. M. Pereira. Deduction revision by intelligent backtracking. In J.A. Campbell, editor, *implementations of Prolog*. Ellis-Horwood, 1984.

[2] Vítor Santos Costa, David H. D. Warren, and Rong Yang. The Andorra-I engine: A parallel implementation of the basic andorra model. In Furukawa [4], pages 825–839.

[3] Vítor Santos Costa, David H. D. Warren, and Rong Yang. The Andorra-I preprocessor: Supporting full Prolog on the basic andorra model. In Furukawa [4], pages 443–456.

[4] Koichi Furukawa, editor. *Proceedings of the Eighth International Conference on Logic Programming*, Cambridge, Massachusetts London, England, 1991. MIT Press.

[5] Seif Haridi and Sverker Janson. Kernel Andorra Prolog and its Computation Model. In Warren and Szeredi [16], pages 31–46.

[6] Sverker Janson and Seif Haridi. Programming paradigms of the Andorra kernel language. In Saraswat and Ueda [13], pages 167–186.

[7] Sverker Janson and Johan Montelius. Design of a sequential prototype implementation of the andorra kernel language. Technical report, SICS, September 1991. (draft).

[8] Robert A. Kowalski and Kenneth A. Bowen, editors. *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, Cambridge, Massachusetts London, England, 1988. MIT Press.

[9] Jean-Louis Lassez, editor. *Proceedings of the Fourth International Conference on Logic Pro gramming*, MIT Press Series in Logic Programming, Cambridge, Massachusetts London, England, 1987. "MIT Press".

[10] Giorgio Levi and Maurizio Martelli, editors. *Proceedings of the Sixth International Conference on Logic Programming*. Cambridge, Massachusetts London, England, 1989. MIT Press.

[11] S. Morishita, M. Numao, and S. Hirose. Symbolical construction of truth value domain for logic program. In Lassez [9], pages 533–555.

[12] Lee Naish. Heterogeneous SLD resolution. *The Journal of Logic Programming*, 1(4):297–303, December 1984.

[13] Vijay Saraswat and Kazunori Ueda, editors. *Logic Programming Proceedings of the 1991 International Symposium*, London, England, 1991. Massachusetts Institute of Technology.

[14] Ehud Shapiro, editor. *Proceedings of the Third International Conference on Logic Programming*, Lecture Notes in Computer Science. Springer-Verlag, 1986.

[15] D. H. D. Warren. The andorra principle. Internal report, Gigalips Group, 1988.

[16] David H. D. Warren and Peter Szeredi, editors. *Proceedings of the Seventh International Conference on Logic Programming*, Cambridge, Massachusetts London, England, 1990. MIT Press.

[17] R. Yang. Solving simple substitution ciphers in Andorra-I. In Levi and Martelli [10], pages 113–128.

Improving Performance Evaluation of Parallel Inference Systems[1]

Christian B. Suttner

Forschungsgruppe Intellektik
Technische Universität München
Augustenstr. 46 Rgb, D-8000 München 2
E-mail: suttner@informatik.tu-muenchen.de

EXTENDED ABSTRACT

In Short:

The relevant aspects for performance evaluation are described and the deficiencies of commonly used performance metrics are exposed. Thereby, the problemacy of a typical application of common performance metrics for the evaluation of parallel inference systems is shown. This issue is exemplified by discussing the popular "relative speedup" metric. Finally, some other metrics are presented which allow adequate evaluations.

1 Introduction

Automated deduction is a computationally intensive field. The enormous search spaces usually encountered for inference systems and the desire to solve larger problems require the use of parallel machines to increase the available computational power as compared to sequential computers.

In the last few years, a number of parallel inference systems have been developed: For example, automated theorem provers such as PARTHEO ([SL90]), ROO ([LMS92]), and MGTP ([FH91]), logic programming systems such as MUSE ([AK90]), and special purpose reasoning systems such as HELIC-II ([NSO+92]). Since more and more parallel machines as well as techniques to execute deductive processes in parallel emerge, a meaningful and accurate evaluation of the weaknesses and strengths of a particular parallel system are of significant importance.

2 Summary

An evaluation methodology is being developed which helps the user to take into account all relevant aspects necessary to derive a meaningful evaluation. There are basically three issues which need to be regarded:

[1]This work was supported by the Deutsche Forschungsgemeinschaft within the Sonderforschungsbereich 342, Teilprojekt A5 (Parallelization of Inference Systems).

1. the purpose of evaluation ("why is evaluated?")

2. the workload ratio ("does the workload depend on the number of processors?")

3. the purpose of parallelism ("what is parallelism used for?")

It is of crucial importance to realize that for each of these issues different answers exist, and that each affects the way evaluations should be performed!

The first issue influences the conceptual selection of metrics which can be used to derive meaningful statements, while the second issue determines which particular definitions must be used. As an example, consider the commonly used relative speedup metric $S_{rel}(n)$, which is defined as the time for the parallel system to solve the problem using one processor divided by the time it requires using n processors. It is well known ([Hoc83], [SG91]) that this metric should not be used for system comparisons or for establishing the general quality of a parallelization (this regards issue one). Despite this, relative speedup is widely used, partly based on the assumption that it is an adequate metric to assess the degree of parallelism inherent to the algorithm at hand. However, for typical parallel inference systems *strategy-shift*[2] effects occur, i.e. the amount of work (measured in elementary[3] operations of the system such as logical inferences) done in order to solve the problem varies with the number of processors used (issue two). This phenomenon for example is responsible for the superlinear speedups[4] often reported for parallel inference systems (e.g. [BK87], [BCLM]). We show that if both the workload and the average inference rate per processor are dependent[5] on the number of processors, relative speedup does not even allow to clearly assess the degree of parallelism inherent to the algorithm. Since both conditions are satisfied for most parallel inference systems, relative speedup on its own is rarely adequate and needs to be combined with other metrics to become meaningful.

Finally, the third issue determines the definition of optimal performance. This also influences the selection of metrics and furthermore has important consequences on the definition of scalability, because it changes the way Amdahl's law should be interpreted ([Gus88], [Zho89]).

These issues and their interactions are currently being summarized to form a basic evaluation methodology which gives guidelines for how to approach metric

[2]Typically, a given inference system defines a particular search space for a given problem. The term strategy-shift is used to denote any change of the order in which nodes of the search space are explored. In parallel inference systems, such a change usually occurs and is dependent on the number of processors.

[3]Elementary operations are assumed to require uniform time.

[4]Superlinear speedup is a matter of dispute whenever a strategy-shift based reduction of work occurs. This is because it might be possible (and reasonable) to employ a similar strategy for a sequential system. In that case, it can be argued that the parallel system obtains superlinear speedup only because a superior search strategy is compared with an inferior one, and speedup is not solely contributable to the use of multiple processors.

[5]For example, the latter is the case if the performance of the parallel system degrades as more processors are added. Common sources for degradation are load imbalance and communication overhead.

selection and application, independent of specific metrics ([Sut92]).

Furthermore, performance metrics proposed in the literature for parallel systems analysis are inspected and their appropriateness for inference systems is investigated. For example, as a single measure for inherent parallelism, generalized speedup (equal to speed of the parallel system using n processors divided by its speed using one processor; see [SG91] also) is an improvement over relative speedup since it additionally takes into account workload changes and therefore separates strategy-shift effects from work distribution effects.

In order to improve the usefulness of evaluations, two new metrics are proposed. As an example, productivity $P(n)$ is introduced, which is defined as the time required by the parallel system to find a solution using one processor divided by the accumulated processing time using up to n processors. It improves on the standard definition of efficiency (i.e. speedup divided by the number of processors) in that it more adequately assesses processor utilization. In particular, other than traditional efficiency, it allows to rate two algorithms with equal speedup behaviour differently as long as their processor utilization differs.

Finally, the consequences of the issues discussed above are applied to derive an appropriate definition of scalability. Thus it is clarified how to use adequate performance metrics in a series of experiments in order to obtain reliable evidence regarding the performance behaviour of the parallel algorithm for increasing numbers of processors.

References

[AK90] K.A.M. Ali and R. Karlsson. The MUSE Or-parallel Prolog Model and its Performance. In *Proceedings of the 1990 North American Conference on Logic Programming*. MIT Press, 1990.

[BCLM] S. Bose, E.M. Clarke, D.E. Long, and S. Michaylov. Parthenon: A Parallel Theorem Prover for Non-Horn Clauses. *To appear in: Journal of Automated Reasoning.*

[BK87] R.M. Butler and N.T. Karonis. Exploitation of Parallelism in Prototypical Deduction Problems. In *Proceedings of the 9th International Conference on Automated Deduction (CADE)*, pages 333–343. LNAI , Springer-Verlag, 1987.

[FH91] H. Fujita and R. Hasegawa. A Model Generation Theorem Prover in KL1 Using a Ramified-Stack Algorithm. In *Proceedings of the Eight International Conference on Logic Programming*, pages 535–548. MIT Press, 1991.

[Gus88] J.L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5), 1988.

[Hoc83] R.W. Hockney. Characterizing Computers and Optimizing the FACR(l) Poisson-Solver on Parallel Unicomputers. *IEEE Transactions on Computers*, C-32(10), 1983.

[LMS92] E.L. Lusk, W. McCune, and J.K. Slaney. ROO - A Parallel Theorem Prover. In *Informal Proceedings of PPAI-91.* Technical Report SFB 342/1/92 A (TUM-I9201), Technische Universität München, 1992.

[NSO+92] K. Nitta, K. Sakane, Y. Ohtake, S. Maeda, M. Ono, and H. Ohsaki. A Legal Reasoning System on the Parallel Inference Machine. In *Informal Proceedings of PPAI-91.* Technical Report SFB 342/1/92 A (TUM-I9201), Technische Universität München, 1992.

[SG91] X.-H. Sun and J.L. Gustafson. Toward a Better Parallel Performance Metric. *Parallel Computing 17*, pages 1093 – 1109, 1991.

[SL90] J. Schumann and R. Letz. PARTHEO: A High-Performance Parallel Theorem Prover. In *Proceedings of the 10th International Conference on Automated Deduction (CADE)*, pages 40–56. LNAI 449, Springer-Verlag, 1990.

[Sut92] C.B. Suttner. On Experimental Performance Evaluation of Parallel Systems. In Preparation, 1992.

[Zho89] X. Zhou. Bridging the Gap between Amdahl's Law and Sandia Laboratory's Result. *Communications of the ACM*, 32(8):1014 - 1015, 1989.

Extended Abstract: The Proposal of a New Method of Generalisation within Automated Deduction *

S. Baker, A. Ireland and A. Smaill
Department of Artificial Intelligence
Edinburgh University
Tel: 031-650-2725
E-mail: siani@uk.ac.ed.aisb

1 Introduction

Generalisation is a proof step which allows the postulation of a new theorem as a substitute for the current goal, from which the latter follows easily. It is a powerful tool with a variety of rôles, such as enabling proofs, defining new concepts, turning proofs for a specific example into ones valid or a range of examples and producing clearer proofs. Although generalisation is an important problem in theorem-proving, it has by no means been solved. It is important and still being investigated for reasons which have to do with cut elimination and the lack of heuristics for providing cut formulae respectively. A cut elimination theorem for a system states that every proof in that system may be replaced by one which does not involve use of the cut rule[1]. Uniform proof search methods can be used for logical systems, in sequent calculus form, where the cut rule is not used. In general, cut elimination holds for arithmetical systems with the ω-rule, but not for systems with ordinary induction. Hence in the latter, there is the problem of generalisation, since arbitrary formulae can be cut in. This makes automatic theorem-proving very difficult, especially as there is no easy or fail-safe method of generating the required cut formula. An important technique for investigating derivability in formal systems of arithmetic has been to embed such systems into semi-formal systems with the ω-rule. This paper presents a new approach to the problems of generalisation by means of "guiding proofs" in the stronger system, which sometimes succeeds in producing proofs in the original system when other methods fail. This method is suitable for automation and results in the suggestion of an appropriate cut formula.

2 The Constructive Ω-Rule

In order to describe the generalisation method proposed, it is first necessary to provide a description of the 'stronger' system mentioned above. A suitable rule other than induction which might be added to Peano's

*The research reported in this paper was supported by an SERC studentship to the first author and by ESPRIT BRA 3245
[1]See [Schwichtenberg 77], for example.

axioms to form a system formalising arithmetic is the ω-rule:

$$\frac{\text{FROM :} \quad A(\underline{0}), A(\underline{1}) \ldots A(\underline{n}) \ldots}{\text{CONCLUDE :} \quad \forall x A(x)}$$

where $\underline{n}$ is a formal numeral, which for natural number n consists in the n-fold iteration of the successor function applied to zero, and A is formulated within the language of arithmetic.[2] However, this is not a good candidate for implementation since there are an infinite number of premisses. We would wish to restrict the ω-rule so that the infinite proofs considered possess some important properties of finite proofs. Hence, a more suitable option is the constructive ω-rule. The ω-rule is said to be constructive if there is a recursive function f such that for every n, $f(n)$ is a Gödel number of $P(n)$, where $P(n)$ is defined for every natural number n and is a proof of $A(\underline{n})$ [Takeuti 87]. This is equivalent to the requirement that there is a uniform, computable procedure describing $P(n)$ [Yoccoz 89], which is the basis taken for implementation (as opposed to the Gödel numbering approach). The sequent calculus enriched with the constructive ω-rule (let us call it $PA_{c\omega}$) has cut elimination, and is complete [Shoenfield 59]. Moreover, since the ω-rule implies the induction rule, $PA_{c\omega}$ + IND is a conservative extension of $PA_{c\omega}$. There are many versions of a restricted ω-rule; this one has been chosen because it is suitable for automation. Note that in particular this differs from the form of the ω-rule (involving the notion of provability) considered by Rosser [Rosser 37] and subsequently Feferman [Feferman 62].

One use of the constructive ω-rule is to enable automated proof of formulae, such as $\forall x\ (x+x)+x = x+(x+x)$, which cannot be proved in the normal axiomatisation of arithmetic without recourse to the cut rule (notably, induction does not carry through). As mentioned above, in these cases the correct proof could be extremely difficult to find automatically. However, it is possible to prove this equation using the ω-rule by generating proofs of $(0+0)+0 = 0+(0+0), (1+1)+1 = 1+(1+1), \ldots$ and working out the general pattern of these proofs.

For the implementation we need to provide (for the nth case) a description for the general proof in a constructive way (in this case a recursive way), which captures the notion that each $P(n)$ is being proved in a uniform way (from parameter n). We will extend our original system for arithmetic by enriching the syntax with names for the syntax of the original theory and representations of proofs in the original theory. Let us consider the automation of the example mentioned in the preceding paragraph. We will presume that, within the particular formalisation of arithmetic chosen, one is given the axioms of addition $s(x)+y = s(x+y)$ (1) and $0+y = y$ (2). The general proof, $P(n)$, will be as follows:

Proof

$$\begin{array}{ll}
 & (s^n(0)+s^n(0))+s^n(0) = s^n(0)+(s^n(0)+s^n(0)) \\
\text{USE (1) } n \text{ TIMES ON RIGHT} & (s^n(0)s^n(0))+s^n(0) = s^n(0+(s^n(0)+s^n(0))) \\
\text{USE (1) } n \text{ TIMES ON LEFT} & s^n(0+s^n(0))+s^n(0) = s^n(0+(s^n(0)+s^n(0))) \\
\text{USE (2) TWICE} & s^{2n}(0)+s^n(0) = s^n(s^n(0)+s^n(0)) \\
\text{USE (1) } n \text{ TIMES ON LEFT} & s^n(\underbrace{s^n(0)}_{\lambda}+\underbrace{s^n(0)}_{\lambda}) = s^n(\underbrace{s^n(0)}_{\lambda}+\underbrace{s^n(0)}_{\lambda}) \\
 & \text{EQUALITY}
\end{array}$$

By $s^n(0)$ we mean the term formed by applying the successor function n times to 0. The next stages use the axioms as rewrite rules, with the aim of reducing both sides of the equation to the same formula. The recursive function for which we are looking is described by the sequence of rule applications, parametrised

[2] Note that for the Gödel formula $G(x)$ we have that, for each natural number n, $PA \vdash G(\underline{n})$ but it is not true that $PA \vdash \forall x G(x)$; thus this system is stronger than PA (where the latter represents Peano's axioms plus induction).

over n. In practice, the first few proofs will be special cases, and it is rather the correspondence between the proofs of P(99), say, and P(100), which we hope to capture. The processes of generation of a (recursive) general proof from individual proof instances, and the (metalevel) checking that this is indeed the correct proof have been automated (see [Baker 90]).

3 Generalisation

Note that there is a class of proofs, including $(x+x)+x = x+(x+x)$, which are provable in PA only using the cut rule but which are provable in $PA_{c\omega}$ [Baker 90]. We consider whether the proof in $PA_{c\omega}$ suggests a proof in PA, ie. in particular, what the **CUT FORMULA** would be in a proof in Peano Arithmetic? That is, what would the A be below?

$$\frac{A \vdash (x+x)+x = x+(x+x) \qquad \vdash A}{\vdash (x+x)+x = x+(x+x)^*}\text{CUT}$$

Ordinary induction does not work on $*$ (primarily because the second, third and sixth terms in the step case may not be broken down by the rewrite rules corresponding to (1) and (2) above, and hence fertilisation cannot occur). That is why we have to use the cut rule. We would wish A to be a more general version of $*$, so that we could prove $A \vdash *$, but on the other hand to be suitable to give an inductive proof, so that we could prove $\vdash A$ by induction. Hence we would be tackling the problem of generalisation by using an alternative (stronger) representation of arithmetic as a guide.

One answer would be to see what remains unaltered in the nth case proof, and then write out the original formula, but with the corresponding term re-named. So, for the example on page 2, we would wish to rewrite the variable corresponding to λ as y. In this case, this would give

$$A \equiv (x+y)+y = x+(y+y).$$

A could then be proved by induction on x. Note that what is meant by 'unaltered' is defined by what is unaffected by the rewrite rules. This procedure is amenable to automation (all that is required is detection of the unaltered terms), and so the cut formula may be produced automatically. Note also that it has not been proved that the cut formula would definitely work, but rather postulated that if there were a cut formula which did work, then it would be of the form where induction was not carried out on these 'unaltered' terms. This method of generalisation will allow the proof of some theorems which pose a problem for other methods, such as $x \neq 0 \rightarrow p(x)+s(s(x)) = s(x)+x$ [Baker 90].

This method should be compared with current generalisation methods. Of these, perhaps the most famous if that implemented by Boyer and Moore in their theorem-prover NQTHM [Boyer & Moore 88]. The main heuristic for generalisation is that identical terms occurring on both the left and right side of the equation are picked for rewriting as a new variable (with certain restrictions). This may be a quick method if it happens to work, but may also entail the proofs of many lemmas, which might need to be stored in advance in anticipation of such an event in order to be more efficient. The problems inherent in Boyer and Moore's approach have led Raymond Aubin to develop their work [Aubin 75]. Aubin's method is to "guess" a generalisation by generalising occurrences in the definitional argument position, and then to work through a number of individual cases to see if the guess seems to work. If it does work, he will look for a proof. If

$$
\begin{array}{rcll}
\forall x\ x + s(x) & = & s(x + x) & (1) \\
\forall x\ (x + x) + x & = & x + (x + x) & (2) \\
\forall x\ x + s(x) & = & s(x) + x & (3) \\
\forall x\ x.(x + x) & = & x.x + x.x & (4) \\
\forall x\ (2 + x) + x & = & 2 + (x + x) & (5) \\
\forall x\ \forall y\ (x + y) + x & = & x + (y + x) & (6) \\
\forall x\ \neq 0 \rightarrow p(x) + s(s(x)) & = & s(x) + x & (7) \\
\forall x\ even(x + x) & = & true & (8) \\
\forall l\ (l <> l) <> l & = & l <> (l <> l) & (9) \\
\forall l\ len(rev(l)) & = & len(l) & (10) \\
\forall l\ rotate(len(l), l) & = & l & (11) \\
\forall l\ rev2(l, nil) & = & rev(l) & (12)
\end{array}
$$

Note that induction is blocked for the above expressions, but they may all be proved by the method proposed (namely by using the constructive ω-rule) and a correct cut formula produced as appropriate.

Table 1: Some Examples of Theorems Proved

it does not, then he will "guess" a different generalisation. However, Aubin's solution does not work in all cases. In particular, if a constructor such as a successor function appears in an original goal, together with individual variables, Aubin's method may result in over-generalisation or indeed no solution at all. The proposed guiding method provides a uniform approach and does not have to check extra criteria, nor work through individual examples. There is less likelihood of overgeneralisation to a non-theorem.

A selection of the theorems proved by this method is listed in Table 1. Note that examples (1)–(7) involve generalisation apart, or else generalisation of common subexpressions. In these cases the method works fairly straightforwardly. Although the examples listed in the table are of a similar simple form, this method may also be applied to complicated examples containing nested quantifiers, etc., for the ω-rule applies to arbitrary sequents. Example (6) provides an instance of nested use of the ω-rule, which carries through directly. For example (7), the cut formula of $even(2.x)$ may be extracted from the form of the general proof, which is an improvement over other methods. However, in some cases where an ω-proof may be provided, it is not clear what the cut formula might be.

The approach also applies more generally to other data-types. Not only is it the case that certain new structural patterns may be seen in the general proof which may guide generalisation, but also that the general representation of an arbitrary object of that type (eg. $s^n(0)$ for natural numbers, $x_1 :: x_2 :: \ldots x_m :: nil$ for lists, etc.) enables the structure of that particular data-type to be exploited, in the sense that rewrite rules may be used which would not otherwise be applicable. In the natural number examples given above, the

general proof is linear in the sense that the proof of $P(s(n))$ reduces to that of $P(n)$. However, in many examples involving lists, this is not so, and a new method for providing a cut formula is needed. The result is an approach which subsumes the previous suggestion: namely, if $R(s(n))$ reduces to $R'(n)$ in the general proof, to inspect the base case $R'^{\cdots'}(0)$, and then replace common subexpressions in this with common variables. In this way, correct cut formulae are suggested for examples (8)–(11) (where <> denotes list concatenation). In particular, the generalisation of (9) is given as $\forall a\ \forall l\ len(rev(l) <> a) = len(rev(a) <> l)$, which is a better result (since it only requires one induction) than that more commonly suggested by other methods of $\forall a\ \forall l\ len(rev(l) <> a) = len(a <> l)$ (requiring two inductions). Definitions of the predicates involved may be found in [Hesketh 91].

4 Conclusions

A new method for generalisation has been proposed which is robust enough to capture in many cases what the alternative methods can do (in some cases with less work), plus it works on examples on which they fail. Implementation of this method is currently being carried out within the framework of an interactive theorem-prover with Prolog as the tactic language, in which the object-level logic is replaced by classical and constructive theories of arithmetic. This approach works for theories other than arithmetic and logics other than a sequent version of the predicate calculus, and may rather be regarded as suggesting a general framework. So long as a procedure for constructing a proof for each individual of a sort is specified, universal statements about objects of the sort could be proved. Thus it appears that the approach described in this paper may be an aid to automated deduction, and provides a mechanism for guiding proofs in more conventional systems.

References

[Aubin 75] R. Aubin. Some generalization heuristics in proofs by induction. In G. Huet and G. Kahn, editors, *Actes du Colloque Construction: Amélioration et vérification de Programmes.* Institut de recherche d'informatique et d'automatique, 1975.

[Baker 90] S. Baker. Notes on the constructive omega rule and a new method of generalisation. Discussion Paper 102, Dept. of Artificial Intelligence, Edinburgh, 1990.

[Boyer & Moore 88] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook.* Academic Press, Boston, 1988.

[Feferman 62] Solomon Feferman. Transfinite recursive progressions of axiomatic theories. *Journal of Symbolic Logic*, 27:259–316, 1962.

[Hesketh 91] J.T. Hesketh. *Using Middle-Out Reasoning to Guide Inductive Theorem Proving.* Unpublished PhD thesis, University of Edinburgh, 1991.

[Rosser 37] B. Rosser. *Godel-theorems for non-constructive logics*, volume 2. JSL, 1937.

[Schwichtenberg 77] H. Schwichtenberg. Proof theory: Some applications of cut-elimination. In Barwise, editor, *Handbook of Mathematical Logic*, pages 867–896. North-Holland, 1977.

[Shoenfield 59] J.R. Shoenfield. On a restricted ω-rule. *Bull. Acad. Sc. Polon. Sci., Ser. des sc. math., astr. et phys.*, 7:405–7, 1959.

[Takeuti 87] G. Takeuti. *Proof theory.* North-Holland, 2 edition, 1987.

[Yoccoz 89] S. Yoccoz. Constructive aspects of the omega-rule: Application to proof systems in computer science and algorithmic logic. *Lecture Notes in Computer Science*, 379:553–565, 1989.