PARALLEL LOGIC PROGRAMMING WITH EXTENSIONS


PRAXIS XXI Programme

Luís Moniz Pereira
Project Director
Departamento de Informática
Universidade Nova de Lisboa
P–2825 Monte da Caparica, Portugal

José C. Cunha
Project Coordinator
Departamento de Informática
Universidade Nova de Lisboa
P–2825 Monte da Caparica
Portugal

Luís Damas
Project Coordinator
Laboratório de Inteligência Artificial
e Ciência dos Computadores
Universidade do Porto
R. Campo Alegre, 823
P–4100 Porto, Portugal

This paper is an edited version, by the authors, of part of the original proposal of the PROLOPPE Project which included contributions from C. Damásio, F. Menezes, J. C. Cunha, J. J. Alferes, L. M. Pereira, P. Barahona, P. D. Medeiros, S.P. Abreu, from the Universidade Nova de Lisboa, and A. Matos, A. P. Tomás, F. Silva, J. P. Leal, L. Damas, L. Lopes, M. Filgueiras, N. Moreira R. Reis and V. S. Costa from the Universidade do Porto, and Miguel Calejo and Carlos Soares from Servisoft.
MOTIVATION

A programming language is a tool and a vehicle for applications.

The Prolog language has successfully proven that LP is an adequate tool for programming in general, and for dealing with Artificial Intelligence (AI) problems in particular.

Moreover, LP has recently gained importance as a result of a worldwide decade of important theoretical studies regarding its evolution into a very expressive language for Knowledge Representation and Reasoning.

We need to promote the evolution of Prolog towards a more expressive new logic programming language, on the basis of the developments in LP semantics and in parallel implementation technology.

Proposed Extensions to Logic Programming:

- a variety of forms of non-monotonic reasoning,
- with explicit negation, and default negation,
- handling contradictions and automated methods for belief revision,
- constraint programming,
- parallelism and distribution support.
A wide variety of experimental applications:

- multi-agents.
- diagnosis of distributed systems
- non–monotonic reasoning.
- natural language processing,
- constraints, time-tabling, scheduling,
- decision support systems.
A language augmenting the expressive and computational power of Prolog requires the support of

- construtive negation, integrity constraints, and disjunction.

- an efficient implementation aiming at overcoming several limitations of current logic programming systems, regarding problem solving in non-trivial AI applications, and in distributed AI.

- joint exploitation of implicit and explicit parallelism over heterogeneous multiple processor architectures,
RESEARCH STRATEGY

Three related projects:

PADIPRO — PARALLEL DISTRIBUTED PROLOG AND APPLICATIONS

A short-term project with DEC.

1. the implementation of a parallel and distributed Prolog over the PVM (Parallel Virtual Machine) environment, as a contribution towards advanced heterogeneous supercomputing environments;

2. the provision of extended Prolog interpreters supporting advanced forms of computational reasoning, including explicit negation, running on top of 1;

3. the application of 1 and 2 to a case study in a strategical important application domain, model-based diagnosis, as a rich field for the exploitation of AI techniques and parallel/distributed strategies.
REAP — REASONING IN PARALLEL LOGIC PROGRAMMING

A project with Prof David Scott Warren, SUNY Stony Brook.

An Extended Logic Programming language which attains several desirable characteristics:

- A formalism to naturally express various forms of negation within the language.

- An parallelizable and distributable computational model, inherent to the non-procedural nature of logic programs.

- The use of tabling for certain predicates allows these to produce solutions whereas the SLDNF search procedure of classical Prolog would loop. A variation on the tabling mechanism feature can be used to include constructive negation in the language.
PROLOPPE — PARALLEL LOGIC PROGRAMMING WITH EXTENSIONS

A large medium-term national research project.

Project Researchers

CITIA/UNL  The CIÊNCIA Center for Information Technology and Artificial Intelligence of the Departamento de Informática, Universidade Nova de Lisboa, Portugal. (16 researchers)

LIACC/UP  The CIÊNCIA Laboratory for Artificial Intelligence and Computing Science, Universidade do Porto, Portugal. (14 researchers)
PROLOPPE TASKS OVERVIEW

This project represents a significant effort, at national scale, towards promoting the Logic Programming paradigm in several directions: theory, language, execution models, implementations and applications.

Expected outcomes:

- The semantic definitions for the new PROLOPPE language;
- The first usable implementation of the language;
- A development environment;
- Exploitation of AI applications.

The tasks below will produce specifications of language extensions, models and prototypes, and the prototypes themselves, which will be demonstrated and made available during the project.
TASK1 Explicit Negation and Logic Programming with Non-Monotonic Reasoning, including contradiction removal, with constructive negation and disjunction.

TASK2 Constraint Logic Programming

TASK3 Execution Models: optimized compilation of the Prolog model, which includes the development of new techniques concerning the following aspects:

- optimized Prolog compilation
- execution models for the proposed extensions,

TASK4 Implicit and Explicit Parallelism:

- implicit parallelism of the OR and AND types over shared memory and distributed memory architectures;
- explicit parallelism suitable to the support of distributed logic programming;
- its application in Distributed AI, and its implementation over heterogenous multiprocessors.
TASK5 Development Environment will integrate the accomplished extensions in an environment with a set of user support tools, such as:

- low-level analyser with performance measuring tools, and sequential and parallel execution tracing;
- declarative debugger;
- browser;
- graphic library and specification languages for system interaction;
- visualization of distributed computations.

TASK6 Some applications will be developed to evaluate, test, promote PROLOPPE, as described. Other applications of non-monotonic reasoning are foreseen, e.g. to planning and to natural language, not carried out within the project, but evaluated by institutional colleges of team members.
Task 1 – Explicit Negation and LP & NMR

We will develop the work on both the declarative and operational semantics of logic programs with explicit negation (besides negation by default) – Extended Logic Programs (ELPs) – in the following aspects:

- definition of contradiction removal methods for ELPs with integrity;

- extensions of the language, and semantics, to deal with rules with disjunctive heads, in order represent incomplete and disjunctive information;

- generalize the derivation procedures developed before to the extended language.

- generalization of the procedures to deal with negative goals that have variables. In this respect we’ll follow two approaches:
  - delaying those goals (similar to the issue of priority of deterministic goals);
  - constructing instantiations for those goals (related to constraint LP)
Task 1 – The Language

Our basis is the language of Extended Logic Programs.

For knowledge representation, default negation does not suffice:

- Default negation is implicit, i.e. does not allow for explicitly stating falsity. This is a serious limitation in natural discourse, commonsense reasoning, legal reasoning, exception handling.
  E.g. $\neg \text{flies}(X) \leftarrow \text{penguin}(X)$.

- One often needs to be more careful in jumping into conclusions, e.g. in $\text{guilty} \leftarrow \neg \text{innocent}$.

- Default negation is not invariant under simple renaming of predicates.
  E.g. $\text{convicted}(X) \leftarrow \text{charged}(X), \neg \text{innocent}(X)$ is completely different from $\text{convicted}(X) \leftarrow \text{charged}(X), \text{guilty}(X)$.

Classical negation does not solve the problem: it satisfies the law of excluded middle $F \lor \neg F$, thus leaving no space for undetermined predicates.

A program is a set of rules of the form:

$$L_0 \leftarrow L_1, \ldots, L_m, \neg L_{m+1}, \ldots, \neg L_n \quad n \geq m \geq 0$$

where each $L_i$ is an objective literal, i.e. an atom $A$ or its explicit negation $\neg A$.  

Task 1 – Declarative Semantics

Our basis for the declarative semantics is \textit{WFSX}. It extends WFS (which is a 3-valued semantics) for ELPs:

1. Consider $\neg$ literals as new atoms.

2. Compute the WFM, forcing at every step $\neg L \Rightarrow \text{not } L$ (coherence)

3. If for some $L$, $\{L, \neg L\} \subseteq WFM(P)$ then $P$ is contradictory.

Unlike other ELP semantics \textit{WFSX} obeys all of:

\textbf{Well foundedness:} It can be characterized by a monotonic fixpoint operator (bottom-up computation).

\textbf{Cumulativity:} Lemmas can be safely added (essential for tabulation).

\textbf{Relevance:} The truth of $L$ depends only on the subprogram below $L$.

\textbf{Partial Evaluation:} Appropriate folding and unfolding of objective literals preserve the semantics.
Task 1 – Operational Semantics

SLX is a derivation procedure sound and complete (for allowed programs and ground queries) wrt. WFSX. It generalizes SLDNF by:

- having two kinds of derivations: T-derivations proving truth of literals, and TU-derivations proving non-falsity (i.e. truth or undefinedness) of literal. Shifting from one kind of derivation to the other is required when proving default literals: the T-derivation of not $L$ succeeds if the TU-derivations of $L$ fail; the TU-derivation of not $L$ succeeds if the T-derivations of $L$ fail;

- extending it to explicit negation by: adding an extra method for deriving truth of not $L$ – if a T-derivation for $\neg L$ succeeds – and restricting TU-derivations of objective literals – a TU-derivation of $L$ fails if a T-derivation of $\neg L$ succeeds (this allows for overriding of explicit over default negation).

- having loop checking both within T and TU-derivations, and over derivations. Loop checking within a derivation makes literals fail (they are false). Loop checking over derivations makes literals fail in T-derivations and succeed in TU-derivations (these literals are undefined).

WFSX programs can be compiled into Prolog code via partial evaluation of the SLX meta-interpreter.
TASK2 — Constraint Logic Programming

Constraint resolution methods are investigated and used in the implementation of constraint logic programming languages:

- resolution methods for the linear Diophantines equation systems and other more general constraints concerning naturals and finite domains;

- incremental hierarchical constraint solvers over natural and finite domains.
TASK 3 — Execution Models

We will develop other execution models and search strategies in the following aspects:

- the design of an Intermediate Computer Description, oriented to the underneath architecture;

- the implementation of a compiler with procedure and intra-procedure level optimization, featuring unfolding, choice points elimination and mode or sequentiality detection;

- extensibility support to provide the proposed extensions to the logic language.

- optimization of the search process based in "Intelligent Pruning", a method that resembles Prolog's intelligent backtracking, but applied to AKL (AKL/IP) execution model;

- sequential and parallel implementation of the AKL/IP language;

- application of the AKL/IP execution model to non-monotonic reasoning, as support of an implementation of the previous extensions.
TASK 4 — Parallelism

In order to obtain a truly efficient implementation and to benefit from parallelism and distribution, we will support:

- innovative execution model joint implicit and explicit parallelism,

- distribution over heterogeneous multiple processor architectures.

- joint exploitation of implicit and explicit parallelism over heterogeneous multiple processor architectures,

- tools for program development with sequential and parallel execution, with support for performance measurement, debugging and visualization.
TASK5 - Development Environment

Objectives

Interactive graphical environment for PROLOPPE

Multi-platform

Macintosh, Windows 95, Motif

Project focus on efficient environment deployment, not programming environment research

Scientific project value lies in the PROLOPPE language, environment is just a vehicle

Mono user

Start favoring academic evangelization and small projects, to later “pull” implementation of multi-programmer facilities
Approach

Build on a proven C++ extendable application framework, already with multi-platform implementations

Prolog-driven environment

- Prolog as scripting language
- Prolog goals implementing high-level interface events
- Prolog structures representing high-level interface objects

Omnipresent Prolog interface builder
Architecture

User Interface thread + Prolog thread (later PROLOPPE)

Light processes now available in all mainstream operating systems

User Interface thread

Additional PROLOPPE specific classes (Trees, Graphs)
Low-end user event handling by XVT-Power++

Prolog/PROLOPPE machine thread

Communication built-ins
Prolog wrappers for methods of main XVT classes
Interface Builder
Environment Browser
Additional tools: declarative debugger, tracer
Programming environment project tasks

Now starting: XVT and YAP learning and testing just completed