Project Description

1 Introduction

We are proposing a 5 year study of the use of enriched declarative programming languages in system specification and development. This is a collaborative group proposal involving 11 researchers (1 PI, 1 co-PI and 9 subcontracted collaborators). Our research is aimed at exploiting the potential of recently designed extensions to logic programming languages to aid in specification and development of integrated software systems. This potential is well beyond the capabilities of traditional logic programming languages.

Logic programming extensions (to stronger logics, constraints, monadic enrichments for data and state database query formalisms, etc.) that preserve the declarative transparency of the Horn Clause core now make it possible for declarative programming to come much closer to its original aim of serving as a powerful, clear and efficient tool both for *specification* and *development* of software systems. This is especially true for systems integrating different components: databses, rules, and other modules of code. Languages resulting from these extensions, here called DPL's (Declarative Programming Languages) permit a principled approach to system design by providing a powerful specification formalism, strong enough to capture high-level concepts and large-scale modular program structure without the need of messy coding of this structure into a weak fragment of logic. They allow for space and time saving considerations at the specification level, via the use of declarative treatment of state and control (e.g. with linear logic). Most importantly: they provide dual assistance to the system developer as a *specification* language for further refinement into code, and as an *executable prototype language* for rapid prototype development and testing.

Because of its declarative character, that is to say, because it is a mathematically precise object, DPL code can be analyzed and tested in a logical manner (using techniques for assertions, diagnosis, logical debugging and a rich operational semantics, discussed above, many of which were invented by the PI's in earlier work), providing further assistance to the system developer who wishes to test a first formalization of a client's working specifications. We are thus able to provide some tools to help deal with an as yet poorly understood and highly problematic stage in system design: passing from working specifications to formal ones, and checking these specifications. This part of system engineering is the hardest, and we are not promising a panacea for this very thorny problem, rather some clear, modular tools, that will be available, as a project deliverable, in a user-interface prototype.

We briefly mention some of extensions we will study. We will use the expressive power of

- constraints, for integrating symbolic and numerical environments, as well as for limited amounts of control,
- higher-order logic to handle meta-programming and program transformation,
- *linear logic* extensions for modelling state and control,
- monadic and categorical extensions to intruduce constraint domains, data types, control and side-effects in a modular, declarative fashion,

- relational extensions and narrowing todevelop clean approaches to compilation, program refinement, and to incorporate database features.

- new database query languages that incorporate monadic structure, collection types and polymorphism

Our work will rest on a thorough mathematical and semantic analysis of the way these extensions work in concert and how programs written in enriched DPL's perform. We will define rich specification formalisms that integrate extensions in a clean manner, and these will need to be modelled. We will use several semantic approaches to accomplish this, such as abstract interpretation and operational semantics that permit detection of different observables, declarative debugging and *compositional* modelling of code. This theoretical foundation is an essential basis for the development of truly declarative (logically clear and modular) tools for system design. It will thus predominate during the first phase (approximately half) of our research.

In the second phase, our work will center on principles for the development of specifications and refining code for integrated declarative program systems as well as program analyzers and debuggers. We will develop a user-interface prototype environment for program development, transformation and diagnosis.

The work is broad in scope: inevitably work with some extensions will prove more tractable than others, some of the theoretical obstacles more formidable than others. It is not clear that all of the extensions should be used simultaneously, which is the sort of thing our work may very well expose. Some will prove more appropriate for different jobs. We will attempt to define and model the abilities of each as part of a toolkit for the system developer.

We will seek to isolate those components that seem particularly practical and easy for the system developer to use. Our research is largely a study of principles, models and tools, but it will be guided, throughout the life of the project, by the system engineering aims of our work, and will take special care to map out clear guidelines for *development* and *use* of these specification formalisms by the system developer.

2 The Principal Investigators

A fundamental aim of this experimental partnership is to encourage fusion and cooperation between theory and practice and areas not traditionally connected in existing software and theory communities, such as software development, program sythesis and transformation and declarative modelling, language theory and compilation. The way ideas from these areas interact will be discussed below.

The 11 members of the research group are listed here with their affiliations and principal areas of research. CV's are given in the biographical sketch section.

Peter Buneman University of Pennsylvania, Databases, Database Languages.

Daniel Dougherty, co-PI Wesleyan University, Rewriting, Unification, Relational Rewriting, Automated deduction.

Peter Freyd University of Pennsylvania, Categories, Categorical Logic, Relational Computing, Categorical semantics of Functional and Declarative Programming.

Manuel Hermenegildo Technical University of Madrid, Compilation, Debugging, Automatic Parallelization, Abstract Interpretation, Higher-order languages.

Giorgio Levi University of Pisa, Declarative programming, Operational Semantics, Abstract Interpretation, Constraints, Program Diagnosis.

Jim Lipton, PI Wesleyan University, Relational and Logic Programming, Categorical syntax and semantics of declarative programming.

Dale Miller Pennsylvania State University, Lambda-Prolog, Logical Extensions, Linear Logic Programming, Specification.

Oege de Moor Oxford, Program Development from Specifications, Relational Computing, Programming Language design.

Gopalan Nadathur University of Chicago, Lambda-Prolog, Higher-order logic programming, implementation and compilation.

Michael O'Donnell University of Chicago, Logic Programming and programming with Equations, integration of Imperative and Object Oriented features.

Catuscia Palamidessi Pennsylvania State University and the University of Genoa, Higher-order and Parallel Logic programming.

3 The Declarative Programming Paradigm and Prolog

Declarative programming means exploiting mathematically correct formal specifications directly to execute or synthesize correct code. This is the correctness *guarantee* that defines the subject (sometimes called *declarative transparency* or "declarity" for short) and it comes with a price. On the one hand proposed extensions must meet stringent semantical requirements: preservation of the meaning of original features, a high degree of modularity, a clear model theory that gives independent mathematical meaning to the program text consistent with program behavior. On the other hand one is barred from indiscriminate use of time-saving computational short-cuts which might compromise the clarity of code and sacrifice the original gains. The major challenge then, is to find mathematically sound principles for adding control components, expressive power and efficiency. This makes the subject mathematically challenging and explains why technological progress in the field must be accompanied by constant progress in theoretical foundations (linear logic, type theory, categorical logic, relational formalisms, constraint systems, varieties, monads).

3.1 Early successes of Declarative systems

Since its introduction in 1972, Prolog has aroused great expectations in communities where fast development of prototypes, knowledge representation, rule-based systems, mechanized reasoning and nondeterminism have been used. Clients with long term concerns about software correctness have been especially drawn to the idea of executable specifications.

Prolog has enjoyed widespread use as a rapid prototyping language for systems with rule based components or with significant mathematical features. Examples of such applications include prototyping systems that mix physical constraints (in the form of systems of differential equations, or

finite-element specifications) and rules, language interpreters in which rules play a significant role, as well as symbolic algebra systems and AI applications (such as game-like programs). Prolog has often been the language of choice for building sophisticated front ends for difficult-to-use database or system components, as well as natural language interfaces to such programs. Finally, it has been used extensively for building on-the-fly, domain specific rule systems of the sort that are often programmed using rule-based system shells. These shells enjoyed widespread use in the 80s, but had some drawbacks. They were often short-lived because they lacked portability. Applications developed using these expensive, proprietary, systems could not be combined with other components. Prolog with its metaprogramming capabilities has been a more successful alternative to these shells: it has the advantage of an international standard, public domain availability, extensive libraries, interfaces to C, etc., that ensure longevity, portability and system openness.

Applications There have been extensive government and industrial applications of logic programming languages in the areas mentioned in the preceding paragraph. We discuss a few representative examples that underscore *scalability, rapid prototyping and ease of use.*

US Govt. applications

In the late 1980's, the EPA developed a Prolog based system for on-site toxic spill diagnosis in the first 24 hours (before experts could arrive on the scene), which was installed on laptops at different public safety locations near toxic sites. Another typical use of Prolog as a rapid-development tool took place at MRSA (US army Logistics, Lexington KY. In 1988, programmers used an off-the-shelf Prolog implementation to build a user interface to a twenty-year old COBOL inventory database that was shipped to US Army facilities in tar files on a monthly basis. The database was extremely rigid, and could only be accessed by a small group of programmers. Requests (by US Mail) for information about parts would generate 100 page printouts that were then checked by hand for days. To solve this problem, MRSA programmers built a simple user dialogue interface that would compile user sessions to COBOL programs returning specific information.

In the same facility a PROLOG system was built to track a production line to flag bottlenecks in procurement. Code development time was short, the system was easy for nonexperts to use.

In the 1980's, Topographic Labs at Ft. Belvoir used declarative prototypes for on-site terrain recognition models.

Industrial Applications

Since 1992 there have been regular meetings of the *Practical Application of Prolog Conferences.* Examining the program of the most recent meeting (April 1997) (available at `http://www.demon.co.uk/ar/PAP97/`) reveals that programming languages based on declarative concepts have been used extensively in industry, including applications such as interfacing users and databases to the World-Wide Web, nuclear power plant monitoring, urban traffic control, natural language translation and parsing of financial statements, and the design and theory of intelligent agents. Some of the companies using Prolog systems in their work include ATT, Boeing, Price Waterhouse, Vality Corp., Microsoft, and several Wall Street firms.

A number of Prolog applications have also been fielded in mass markets. For example, Microsoft's popular "WindowsNT" operating system includes a Prolog Processor internally. Prolog rules are used to describe in a declarative way the relationship and incompatibilities between software and hardware modules. The system uses this knowledge to correctly and efficiently configure itself for different hardware and software combinations.

3.2 Limitations of conventional declarative languages: the need for extensions

Prolog has enjoyed a commercial success in the highly specific markets it (accurately) targeted twenty years ago. However it has fallen short of some of the lofty expectations of its earliest days. One reason is that Prolog is the result of an unfortunate fusion of a weak, but logically pure declarative core, and a series of control features and "hacks" aimed at making up for its deficiencies. The original notion of "specification" in logic programming (proposed by e.g. Kowalski), Horn Clause logic, turned out to be far too restrictive for many applications, and sacrificed too much efficiency for the sake of declarity. As a result, control features (the *cut*, *assert* and *retract*) and semantically ambiguous so-called higher-order predicates (*call*, *univ*) were added in an ad hoc manner to the pure Horn Clause fragment, resulting in a hybrid language that was hardly declarative at all. To this day, Prolog is encumbered by control constructs, regarded as indispensable by programmers, that critically compromise the clarity of the code and make debugging a nightmare.

Extensions Logic programs written in the pure Horn Clause fragment are easy to understand, debug, and prove correct, provided they do not resort to intricate coding schemes to simulate control, or to capture the structure and behavior of elaborate module architectures and data types. If they do, Prolog's logical clarity is not really of much help. The logic and its conventional semantics can only vouch for the agreement of input-output values with the code, but *not for the correctness of the translation of imported features into Prolog code*. Unfortunately, this is often the kind of problem the programmer and system designer are really interested in. System designers want to think in terms of the high level concepts that actually occur in the original specification.

This is a problem directly addressed by truly declarative extensions to the Prolog core. They make it possible for declarative code to have a high-level structure that reflects large scale characteristics of the specification. At the same time, such extensions allow for greater ease in passing from working specifications to code and for the implementation of high-level diagnosis and validation environments.

A listing of the main contributions of the project researchers to the development of extensions to the Horn-clause core is discussed in the precursor research section following the project description.

4 The Case for Declarative System Design

There have been a number of steps made in the direction of axiomatic, modular specifications [8, 21] of software architectures (building on earlier pioneering work by Douglas Smith on the Kestrel Interactive Specification System [267]). Many formal languages have been introduced for software design over the past two decades [123, 261, 248, 247, 234]. Although some of these approaches have been termed *declarative* because they have been based on logically precise languages, no effort has been made to exploit the executable character or to harness the abstraction and unification features,

the context management and unformity of declarative programming languages as a specification formalism.

In our opinion, declarative languages have a decade's track record of success in solving problems that are essential components of software specification, prototyping and development. They are ideally suited to the task. Some evidence for this claim:

Metaprogramming: Declarative Languages are among the most successful languages for formalizing and manipulating a target language. Here even conventional Prolog has an impressive track record, and there have been many proposals and a few implementations of metaprogramming extensions of the Prolog core, as discussed in the preceding section. Lipton and his students Chapman and Ruhlen have implemented a logic program transformation and rewriting system using SWI Prolog.

But there have been more substantial improvements. λ-prolog permits the formalization of higher-order abstract syntax: its abstraction mechanism allows abstraction of object language variables, β-reduction implements substitution and α-conversion renaming. Higher-order unification permits sophisticated pattern matching of programs for program transformation. Miller, Pfenning, Hannan, Felty and others have used λProlog to specify the operational semantics of a polymorphic functional programming language in this manner. Felty has implemented tactics for building proofs in a powerful target theory (Coquand's Calculus of Constructions).

The specification *matching*, interface binding of components and constraint specification discussed in connection with software architecture in e.g. [233], would be naturally expressed using the same tools.

Formalization of Program Transformation: De Moor and his colleagues have experimented with functional program transformation in λProlog, exploiting higher order-unification for pattern matching with transformation rules.

Miller and Hannan [124] have illustrated how it is possible to transform high-level, provably correct, λProlog code into more explicit, low-level logic code that could be directly implemented as a lowlevel stack machine.

Recently, Buneman and Kosky introduced a declarative language, WOL, based on Horn-clause logic, for specifying database transformations and constraints. They also proposed a method of implementing transformations specified in this language, by manipulating their clauses into a normal form which could then be translated into an underlying database programming language.

Interface specification: Prolog and extensions have been successful in specification and implementation of interfaces between components, and for the design of front ends to software components.

Testing, validation and debugging platforms: Hermenegildo and his co-workers have implemented a program validation and testing system based on an assertion language that extends Prolog, based on abstract interpretation. Comini, Levi and others have shown how to implement error diagnosis and debugging using similar ideas.

Specification of state- and resource sensitive architectures and structures: Using Miller's declarative language FORUM, Miller and Chirimar have developed a natural and modular specification of the pipelined, concurrent architecture of the DLX RISC hardware processor. FORUM has also been used to capture most of the operational semantics of ML, with exceptions and imperative features.

Model Checking: A significant recent success of logic programming is its application in the area of model checking (Ramakrishna et al. [243]), a technique aimed at proving that a system specification meets properties which are expressed as temporal logic formulae. Two important requirements in using this technique are a mechanism that supports a clear and concise presentation of these formulae and the rules governing their derivation and a method for calculating the fixed-points that are usually entailed in the formulae. The logic underlying Prolog is well suited to the first requirement, and an approach to computing with this logic that uses memoization provides an efficient and effective means for fixed-point calculations. These ideas have been applied to a variety of process languages and logics and have quickly proved to be competitive with well-established systems for performing the same tasks.

Some of these recent developments will play a seminal role in our work. Our proposed research will build on these results in various ways, extending the required theory and the implementations with the aim of identifying the essential components of a declarative-based specification formalism and toolkit.

We now consider some of these points in more detail.

5 Details of Proposed Research and Implementation

Higher-order hereditary Harrop formulas and λProlog This language epitomizes one of the richest extensions to the logic of logic programming in the last decade. It introduces many important specification and programming features previously missing from this paradigm, including polymorphic typing, higher-order programming that captures abstractions over procedures, scoping mechanisms that are important in modular development of programs and a stronger term-language that considerably enhances the meta-programming capabilities of logic programming. Moreover, all these additions are achieved in a principled way by using an expressive underlying logic.

Much recent work has been devoted to harnessing the power of this language in practice. Sublanguages that sacrifice a limited amount of expressiveness in acceptable ways in exchange for efficiency have been described, the most notable of these being Miller's L_λ. At the level of language design, effort has been invested in defining a principled, yet practical, module system useful for programming-in-the-large. Nadathur and colleagues have devised implementation techniques for the efficient realization of scoping mechanisms and higher-order aspects and to support separate compilation of modular programs. The latter work has relied on a deep understanding of the theoretical underpinnings and of analysis tools: it has, for instance, introduced a calculus for the explicit treatment of lambda calculus substitutions and utilized program equivalence notions in compilation.

We propose continuing these investigations at the interface between theory and practice in relation to λProlog. We will develop reasoning tools that are useful at the language level as well as in particular application areas; an example in the former category is a calculus for reasoning

about module interactions and, in the latter category, analysis techniques for higher-order abstract syntax. New approaches to operational semantics will be extended and exploited towards improved compilation and the development of a satisfactory programming environment. We will also investigate the coexistence of λProlog-like features with other extensions to logic programming. The end result of these studies is to come as close as possible to a clean, comprehensive semantical account and to give a description of interfaces between different approaches to computation. Finally, we will test the practical relevance of our research ideas using a compiler-based implementation of λProlog being produced by Nadathur and colleagues.

Declarative treatment of state and resources: the use of linear logic Linear logic incorporates within it an elegant understanding of the dual notions of resource and agent. Andreoli and Pareschi have used these aspects of linear logic to make an effective combination of object-oriented programming with logic programming. Their initial design, a language called LO, is the inspiration behind the coordination language they are now developing at Xerox in Grenoble France. Hodas and Miller have developed a linear refinement of the logic underlying λProlog, providing a logically clean notion of state that can be encapsulated within objects. Such declarative control over resources has also been shown to be useful in tasks such as natural language processing, automated deduction, databases, and programming language specification. More recently, Miller has shown that all of linear logic can be presented in such a way that it modularly extends all of the logic programming languages designed using the goal-directed-search design criterion. The resulting presentation of linear logic, called Forum, has been used in a number of recent PhD theses and also in various computationally challenging specifications. Two impressive applications of this language have been the natural and modular specification of (most) of the operational semantics of the Standard ML programming language (including such imperative features as references, exceptions, and continuations) and of the pipe-lined, concurrent architecture of the DLX RISC hardware processor. Not only were these specifications more modular and flexible than the specifications often given for these systems, it was possible to use simple facts about linear logic and proof theory to prove various properties of these specifications. For example, it was possible to show that the sequential semantics of the DLX processor was equivalent to its pipe-line semantics, an important and generally difficult result to establish.

The use of linear logic in declarative programming thus paves the way for significantly more efficient applications of declarative languages and greater ease of specification of state and resource-sensitive information. It also makes possible testing and validation (via the assertion/diagnosis environment environment discussed below and in the opening summary) of resource and state-related properties of programs.

At this point it is possible to begin to make precise ways in which specification languages for the working system developer can incorporate information that would be formalized in linear logic, tapping this new dimension of expressive power for system specification and design. This will be one of the goals of our research.

Verification and Debugging based on Abstract Interpretation: the research goals We plan to devise and implement abstract interpretations for an extended collection of non-trivial program properties. We also plan to extend the CIAO framework to deal with certain higher order language features and specifications. In addition, we will extend the analyses themselves

to deal in an accurate and efficient way with these higher order features and with state change. This extension will include support for dynamic modular analysis, where modules are implemented dynamically via higher order features as opposed to the static syntactic modules supported in our previous work. We believe that the incremental analysis techniques that we have recently developed provide a good basis for tackling these problems. We also plan to explore other applications of the resulting instantiations of the framework, which we see as ranging from static debugging and program optimization to semi-automatic program documentation. All these elements are expected to be fundamental components of the project deliverables.

Logic Programming and Program Synthesis Declarative specifications are not only suitable for rapid prototyping: they can also serve as the starting point for synthesizing more efficient implementations. The Algebra of Programming research group at Oxford (see references at http://www.comlab.ox.ac.uk/publications/books/algebra) has developed a calculus for deriving programs from specifications in relation algebra. A pivotal role is played by Freyd's theory of allegories, which allows straightforward generalisation of some of the higher-order programming idioms in functional programming to a relational setting. The derivation process also benefits by categorical language in focusing on the derivation of classes of algorithms (such as dynamic programming, greedy algorithms and branch-and-bound), rather than individual problem instances. It is this latter feature that sets this work apart from most other attempts in formal program synthesis, which are either very general (at the level of combining systems), or extremely detailed, focusing on the reconstruction of a nifty algorithm for one particular problem.

The work has now reached a stage where it is possible to write meta-programs that take a declarative specification, and apply a particular algorithm design strategy. There exist commercial systems that support such meta-programming (in particular the Reasoning SDK, and Microsoft's Intentional Programming), but the meta-language employed there is mostly ad hoc. More specialised systems for meta-programming have been built on top of these products (in particular Kestrel Institute's Kids and SpecWare systems), but the basis of meta-programming for transformation remains ill understood. We propose to remedy this situation by exploiting the recent advances in logic programming described above, in particular λProlog, to build a library of primitives for program transformation. We shall test this library against commonly known techniques for refining declarative specifications to efficient imperative programs, and also against the algorithm design techniques referred to above.

The surge of interest in meta-programming is demonstrated by the rapid growth of Reasoning (http://www.reasoning.com), and the importance Microsoft places on Intentional programming, (http://www.microsoft.research.com/research/ip). As indicated above, it is clear that while the demand exists, the technology could benefit significantly from the expressiveness of modern declarative languages such as λProlog.

Mechanisms for concurrency System programming requires the capability to specify interaction and communication between the various components. This will be one of the main focusses of Palamidessi's research in this project.

The first attempts of enriching logic programming with explicit mechanisms for concurrency started around the 80's, and resulted in languages like Concurrent Prolog, PARLOG, and Guarded Horn Clauses. All these proposals, however, suffered from a lack of declarativity of the concurrency mechanisms. Their meaning was specified only at the implementation level, and as a result

the resulting languages departed from a purely logical interpreation, and were quite obscure to understand from the programmer's point of view.

The most successful proposal coming from the Logic Programming community was the paradigm of Concurrent Constraint Programming by Saraswat [256]. In this proposal, the Constraint Logic Programming paradigm was enriched with communication mechanisms similar to those of Milner's CCS, which allowed to rely on a well-understood semantic theory.

Another proposal for integrating constraints and concurrency has been developed by Gert Smolka and his group [269]. Their paradigm, also called Concurrent Constraint Programming, has been the most successful as it is very expressive (in particular, it has higher-order features), has been efficiently implemented, and it is currently being extended so to support Object-Oriented features. Our project differs from the above proposals in that we aim at defining concurrency mechanisms which can be justified from a logical point of view. Our approach is to use certain combinators of linear logic to specify process interaction, along the lines which have been explored in the Join Calculus [106], the Gamma Multiset Rewriting [19, 20], and the Chemical Abstract Machine [31].

Interoperability of Declarative and Imperative Programming Pure declarative programming provides two different sorts of advantages:

1. algorithmic tools, such as unification and backtracking search in Prolog, pattern matching and lazy evaluation in functional languages;

2. analytic security, particularly referential transparency.

We need to enjoy these advantages, without giving up the corresponding advantages of well-crafted imperative programming. To some extent, the algorithmic tools of various languages may be combined into kitchen-sink languages, usually with some loss of efficiency. But, the analytic security of declarative languages is utterly lost in mixed declarative/imperative programming.

Instead of providing a specific hybrid language, with declarative as well as imperative constructs, we will design and implement a clean interface for combining declarative and imperative programs. We already have the basis for a clean and powerful interface between lazy functional languages and imperative languages. Samuel Rebelsky, in a dissertation supervised by O'Donnell, designed *Term Tours*, a new I/O protocol for lazy functional programming, generalizing *pipes* and *streams* to allow the order of traversal of an output tree to be determined dynamically by the reader.

We propose to implement a simple lazy functional language, based on *Equational Logic Programming* or on the *Gopher* subset of *Haskell*, as a *Java* library, using *Term Tours* as the procedural interface. Using the results of Stephen Bailey's dissertation on fast compiling of *ELP* and *Gopher*, we can offer both interpreted and compiled lazy functional programming as functionally equivalent alternatives, to be selected depending on performance needs. The main evaluation code of existing interpreters and compilers remains intact in such an implementation: only the I/O procedures, which are rather small for pure functional programming, need to be changed.

We will also seek a distributed implementation of the *ELP*/*Java* system, and an analogous way of integrating λProlog with *Java*.

Semantics Research The text of a logic program is a formal specification, hence a mathematically precise term. Different logic programs semantics make precise what the code is supposed to

mean mathematically (as a specification) and computationally (how its meaning is connected to its computational behavior). Both approaches are essential to the applications proposed here, and are needed to understand the uses of the extensions in combination. A clear semantics is essential to program analysis, to the design of debuggers that retain or extend the logic structure of the original program, and to the design of tools for program development, specification refinement, diagnosis and validation of code.

Operationality and Compositionality in Modelling Conventional Prolog semantics assigns sets of input-output values to programs. It has two severe limitations: failure of compositionality and blindness to operational behavior.

- Non-compositionality means the following: if P_1 and P_2 are two program components, one cannot infer the meaning of the combined program $P_1 \cup P_2$ from the meanings of the components. This means new models must be made each time new components are added to a system, or preexisting components are combined. This deficiency would seem to render conventional semantics useless for large scale system design.

- Conventional semantics are insensitive to critical operational features. For example they do not account for non-termination, or distinguish between programs that compute the same value in different ways, or compute it only once, instead of repeatedly, or use different system resources. Unfortunately, these kinds of distinctions are essential, especially in integrated systems. If a module interface is written in badly behaved (i.e. wasteful) declarative code the whole system may perform miserably, even if the components are efficient and well-designed. Semantic tools must provide the means for analyzing such behavior and validating the operation of critical declarative components.

Over the past fifteen years, a research group headed by Prof. Levi has developed semantics that address both these shortcomings, including models which are fully compositional, and a hierarchy of semantics that distinguish between essentially all nuances of program behavior that are of interest to programmers. This work must now be applied to *Prolog extensions* if they are to be effectively used in system specification and development.

This kind of work will require cooperation between the various PI's, e.g. Levi-Nadathur-Miller on compositional semantics for λProlog, Levi-Lipton-Freyd on categorical notions of operational semantics, O'Donnell-de Moor for treatment of state and specification refinement, Dougherty-De Moor for functional-logic extensions, Buneman-Lipton-De Moor for Relational and Database extensions, etc.

Operational Semantics and Observables We propose to extend the semantics approach pioneered by Levi and his colleagues to more complex and powerful languages (higher-order hereditary Harrop formulas) and by formalizing it within the categorical framework. Moreover, we want to develop our taxonomy of observables, by taking into account refinement operators. Finally, we want to develop semantics-based tools in the area of verification and debugging. This research will include some of the following components.

- (The s-semantics of higher-order hereditary Harrop formulas) The s-semantics approach, being essentially driven by the operational semantics, can be useful to obtain a denotational

semantics (and various versions of the T_P operator), for hereditary Harrop formulas and to reason about operational properties, as needed in program analysis.

- (Categorical formalization of the framework of observables) Categorical tools similar to those used in (Freyd-Finkelstein-Lipton 1994) can be useful to set a single formal framework for reasoning about operational semantics and abstraction. We expect several useful properties, as compositionality and precision, to be very natural to formalize in such a framework.

- (Observables and refinement operators) Refinement operators (Filè-Giacobazzi-Ranzato 1996) allow one to systematically improve the precision of observables. We want to study the relation between the various refinement operators and the properties of the resulting denotation, with the goal of making more systematic (i.e., almost automatic) the design of optimal observables.

- (Reconstruction of verification techniques) The framework of observables was recently used as a foundation for abstract diagnosis, a novel combination of abstract interpretation and declarative debugging (Comini-Levi-Meo-Vitiello 1996). Abstract diagnosis can be viewed as a special case of verification without preconditions. We want to reconstruct existing verification techniques (Apt 1996), through a suitable abstract semantics modeling preconditions. The specification being a pre-fixpoint of the corresponding T_P operator should be the new partial correctness condition, thus turning the "theorem proving" involved in the traditional partial correctness conditions into a simpler "abstract execution". The new formalization should allow one to tackle the issue of completeness and to make the results on the systematic design of observables available to verification.

6 Conclusion

We propose a research project involving 11 researchers (1 PI, 1 co-PI and 9 subcontracted collaborators) in the area of declarative specifications and software development. The main aim of the research is to exploit recent extensions to the syntax and semantics of the declarative programming paradigm that have greatly expanded its scope and potential usefulness as a specification and software development language. Our research will provide tools for modelling and specifying such extensions with the following aims.

- To clarify language principles for building modular and logically correct large-scale systems.

- To exploit the potential for system development in the dual use of declarative programming as both a programming language for rapid prototyping and a specification language for subsequent implementation in more widespread languages.

- To develop programming techniques for utilizing the new declarative features and methods for verifying and understanding these programs and to study principles for incorporating such features into existing programming language paradigms and for their smooth interaction.

- To develop techniques based on abstract machines and compilation for the efficient implementation of the new language features.

- To build a program specification and testing prototype that extends the validation and abstract diagnosis work done by Levi and Hermenegildo, and the program synthesis and transformation research of de Moor, Miller and Buneman, discussed above.

Precursor Research and Results from prior NSF work

7 Some of the Collaborators' Contributions to the Development of Logic Programming Extensions

- Miller and Nadathur have exposed the critical, language-independent, principle of *uniform provability* that facilitates declarative language design as well as a simultaneous treatment of declarative and operational semantics. When used in conjunction with rich logical languages this principle can lead to systematic extension of control and abstraction capabilities without compromising declarity. It has, in fact, been already used in this fashion to introduce scoping devices, mechanisms for modular programming, higher-order programming, higher-order abstract syntax for metaprogramming and notions of state into declarative programming. The family of declarative languages resulting from this work includes λProlog, L_λ, Lolli, Forum.

- Levi and his group (including Catuscia Palamidessi) at the University of Pisa have developed operationally sensitive semantics for Prolog and the CLP constraint languages that make it possible to reason precisely about program behavior, and to focus on computational observables, using powerful techniques, such as abstract interpretation and collecting and S-semantics. Levi's group has introduced several forms of compositional semantics that permit incremental development and analysis of programs. This is a fundamental contribution towards the understanding and principled development of large scale program structure and heterogeneous systems.

  These developments have yielded powerful new tools for debugging, automatic error diagnosis of programs and compilation.

  Palamidessi has also worked extensively with the development, semantics, and implementation of concurrent logic programming, as well as abstract interpretation, denotational semantics and validation of logic programs.

- Hermenegildo and his group at the Technical University of Madrid (UPM) have designed the CIAO assertion environment which is one of the most practical tools developed to date for analyzing, debugging and diagnosing declarative programs. This toolkit permits programmers to analyze program content and behavior via high-level test assertions. The system is capable of locating errors in code, and pointing up "symptoms" of disagreement with specifications. In conjunction with the declarative system design research proposed here, the potential of such an environment for end-to-end system development is enormous. It is one of the few tools that helps the programmer mediate between working specifications and formal ones, as well as between formal specifications and code. This work is strongly based on the technique of abstract interpretation, which Hermenegildo and his group have applied successfully to the design of fast and correct optimizing and parallelizing compilers. They have developed what is currently perhaps the most frequently used framework for analysis of logic and constraint logic programs, PLAI. This work demonstrates unequivocally the relevance of semantical tools to implementation questions.

- Buneman and his group at the University of Pennsylvania have incorporated monadic syntax, structural recursion, polymorphism and collection types into database languages in a way that transparently extends the underlying relational database paradigm. This work provides excellent guidelines on how to incorporate such features into declarative programming, as well as a framework for reasoning about these extensions.

This group has also explored the use of declarative specifications for merging databases, which is one of the more promising applications of Declarative Programming to merging large systems.

- De Moor, working with R. Bird and colleagues at Oxford, has developed a calculus for deriving programs from relation algebra specifications. His work is based on Freyd's theory of Allegories. De Moor has developed a metaprogramming framework for program derivation, and is studying the use of expressive declarative languages (such as Miller and Nadathur's λProlog) in conjunction with these techniques to build a declarative environment for program transformation.

- Freyd, Lipton, Dougherty, Finkelstein and McGrail have defined a categorical notion of operational semantics for logic programming with constraints, as well as categorical and (together with Claudio Gutierrez at Wesleyan) relational formalisms for extended program syntax and compilation of declarative extensions. Lipton and McGrail are developing monadic extensions to incorporate, in a clean and modular fashion, data types, control and side-effects.

- O'Donnell has developed the foundations of equational logic as a programming device. The emphasis in his work has been on *complete* equational inference. This leads naturally to a very precise requirement for laziness, to new and more powerful memoing techniques, to new ideas about I/O interfaces, and to a different style of code generation from the subroutine-per-function code of lazy functional programming.

8 Precursor Research from Partner Programs

Peter Buneman

Database Applications

Starting from a new approach to the construction of database queries that uses the categorical notion of a monad to generalize set-theoretic operations to other collection types, the database group at the University of Pennsylvania has developed a practical data query and transformation language. The language is implemented and has been widely applied to database integration problems, especially those arising in the field of Genomics which is characterized by the complexity and volatility of the schemas. The language is capable of communicating not only with conventional (relational and object-oriented) databases but also with a variety of data exchange and archive formats. These formats have relatively complex type systems, and probably hold more of the world's data than conventional databases.

A variety of conservativity results establish a close connection between the hierarchy of database languages and various extensions to this monad-based language. Moreover the categorical basis for the language provides an equational theory that is the basis for its optimization system that is essential to a working database language. This system is now being extended to include aggregate functions and arrays. In another direction, the monadic optimization rules have also been used to design an efficient and powerful language for "semi-structured" data – essentially data based on a dynamic type system. The current research includes work on integrating these two – structured and semi-structured – forms of data and on a uniform and compositional approach to database constraints.

Results from prior NSF research

Project Title: Collection Types in Programming Languages and Databases
Project Period and Amount: 1995-8, $210,000
PI: Val Tannen
co-PI: Peter Buneman
Goals, Objectives, and Targeted Activities. The purpose of this contract was to study the efficient representation and querying of collection types in both programming and databases. It started from work on an simple algebra and syntax for complex object languages that uniformly handles a variety of collection types: lists, multisets and sets, as well as record and variant types. This work has been applied to the problem of querying, transforming, and integrating a variety of non-standard databases such as scientific data formats. It also included the investigation of optmization techniques, aggregate operations and the extension of the basic principles of collection types to other types such as arrays and semi-structured data.
Indication of Success. The greatest success of this project has been its contribution to the Kleisli (now K2) sustem of integration of non-standard database sources. One of the most interesting developments in scientific databases, especially those used in biology, is the diminishing importance of standard database technology. The data associated with the human genome project is complex and is evolving rapidly. Because the data is so volatile and the boundaries of the domain are ill-defined, efforts to build large integrated repositories in using standard relational or object-oriented

technology have met with mixed success, and there is increasing reliance on a tools for integrating a variety of complex heterogeneous data formats that have been designed for the transmission and archiving of data. The problem of providing scientists with tools for understanding, integrating, and analyzing the proliferating data sources at their disposal is one of the great challenges facing database research.

By developing "drivers" for most current data sources have been constructed, Kleisli has been used extensively within bio-informatics projects both within and outside Penn. The software is now deployed in a number of pharmaceutical companies. We have applied for patents on optimization techniques for aggregate queries and for query languages for semi-structured data.

Potential Related Projects. Much of the research in this project resulted from the fusion of ideas in programming languages and in databases. The PIs feel that there is more to be gained from this interaction, especially in the study of type systems for object-oriented languages.

Project Impact.

- The project has been used partly to fund two PhD students, both current, one female.
- Some of the material on collection types is used in Penn undergraduate and graduate courses on databases. It is also used in a course in parallel computation.
- The ability to integrate biological databases was of key importance in the foundation of Penn's Center for Bioinformatics, the first in any US academic institution.
- A number of industrial collaborations have been are based upon our database integration work, notably with SmithKline Beecham. As mentioned above we have filed for two patents based on optimization work.

Publications resulting from award:

1. "Path Constraints in Structured and Semistructured data", P. Buneman, W. Fan and S. Weinstein, to appear in PODS '98
2. "A Kleisli interface to Shore", C. Hara, S Davidson and L. Popa. Brazilian Database Conference, September 1997
3. "Adding Structure to Unstructured Data", P. Buneman, S.Davidson, M. Fernandez and D. Suciu, Proc. ICDT, Delphi, Jan 1997.
4. "A Typed Pattern Calculus", D. Kesner, L. Puel, and V. Tannen. Information and Computation, 124, 1996, pp. 32–61.
5. "Principles of programming with complex objects and collection types", P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Theoretical Computer Science, 149, 1995, pp. 3–48.
6. "Challenges in Integrating Biological Data Sources," S.B. Davidson, C. Overton and P. Buneman. Journal of Computational Biology 2:4 (Winter 1995), pp 557-572.
7. "A Data Transformation System for Biological Data Sources," P. Buneman, S.B. Davidson, K. Hart, C. Overton and L. Wong. Proceedings of the 21'st International Conference on Very Large Data Bases (September 1995), pp. 158-169.

8. "BioKleisli: A Digital Library for Biomedical Researchers," S.B. Davidson, C. Overton, V. Tannen and L. Wong. Journal of Digital Libraries 1:1 (November 1996).

9. "A Query Language and Optimization Techniques for Unstructured Data," P. Buneman, S.B. Davidson, G. Hillebrand and D. Suciu. Proceedings of SIGMOD'96.

10. "Programming Constructs for Unstructured Data," P. Buneman, S.B. Davidson and D. Suciu. Proceedings of the Workshop on Database Programming Languages (Sept. 1995).

Software created and now being made available to the research community. Details about the Kleisli prototype are available at `http://www.cis.upenn.edu/~db/home.html`. This page contains:

- CPL Web Service: allows you to interactively evaluate CPL queries, and to create and execute parameterized queries encoded in HTML forms.
- Form Based Human Genome Map Search: a sample parameterized CPL query encoded in an HTML form.
- CPL manual: description and examples of the language.

Kleisli is also being used in several industrial and research projects outside Penn: 1) within SmithKline Beecham; 2) within Lockheed-Martin; and 3) at the University of Manchester; 4) by Bob Grossmans' group at University of Illinois - Chicago; and 5) numerous projects associated with the Limsoon Wong's group in the Bioinformatics Center, Kent Ridge Digital Laboratories, Singapore.

Dan Dougherty

Dan Dougherty has conducted research relevant to the present proposal under the Office of Naval Research grant Computing with Relations — Combinatory Logic Programming, Types and Allegories. Office of Naval Research grant N-00014-95-1-0634, 1995–96.

This work has been concerned with the interaction between first-order and higher-order language features, both in the programming language and theorem-proving contexts.

Unification and matching is fundamental in theorem-proving and logic programming. Unification and matching under an equational theory embodies a strategy of building semantic reasoning directly into the computational engine itself, as opposed to explcitly mixing general facts about the underlying domain with the representation of the specific problem at hand. In collaboration with Frederic Otto and Paliath Narendran, we investigated the relationships between equational unification, equational matching, and second-order matching and unification.

There is work in progress on the higher-order matching problem. Taking a model-theoretic approach to the problem has lead to simple proof of decidability of fourth-order case. From the work of Tannen and Gallier, we know that such a decidable fragment remains decidable when a matching-decidable first-order equational theory is added. A typical example is the addition of asociative-commutative operators to a theory. The applications here are to program transformation and the automated derivation of programs.

In collaboration with Pierre Lescanne at Ecole Normale Superieure de Lyon, we are exploring some aspects of explicit substitutions as applied to implementation of functional programming

languages. Specifically, we prove standardization and strong normalization theorems; these are the foundations for the correctness of various evaluation strategies (call-by-value, call-by-need, lazy evaluation).

Our graduate student Claudio Gutierrez has shown the decidability of the equational theory of allegories, and given reasonably tight bounds on the computational complexity. The decidability of the unification problem for this theory is still open. As part of the attack on this problem, Gutierrez has achieved a much-improved bound (single-exponential space) on the complexity of unification under associativity.

Peter Freyd

Categorical Logic Programming

The research described here was supported under
ONR grant NOOO14-92-J-1878 PO5.
Budget Period: 06/01/92 - 02/27/97
Total Award: $400,000
Title: Relations, categories and Computations
(Categorical Foundations of Constraints & Logic Programming)

New Foundations for declarative programming Categorical logic extends the scope of logical syntax and semantics by generalizing certain components: the syntactic domain of individuals, connectives, identity of terms and predicates, unifiablity, the notion of proof and transition, and the meaning of quantifiers.

The extra expressive power results in more powerful declarative languages and also gives natural framework for modelling many proposed extensions of the declarative paradigm.

In joint research with Finkelstein and Lipton [100] we showed how to define programs, proof rules, and an interpreter algebraically, over finite product categories with certain canonical structure. This led to new declarative extensions, including monadic extensions for cleanly adding data types, and notions of control, studied by McGrail and Lipton [173].

Using the Kowalski-Van Emden fixed point theorem as a starting point we developed a categorical notion of operational semantics and abstract machine. We gave a unified treatment to a number of logic programming extensions, including programming with constraints (built into the underlying categorical syntax) and we outlined an axiomatic approach based on continuous functors.

Cartesian and alternation logic: The preceding results were based on some parallel research into new ways to formalize categorical syntax. I developed the syntax and proof theory corresponding to "regular' and "extensive" categories. Among the theories that appear as special cases of this syntax are "essentially algebraic theories" and almost every theory of "categories with structure". Perhaps the most important single case is the theory of cartesian categories. (Note that categories with products are most definitely not a special case of equational algebraic theories.)

Motivated originally by the purest of considerations, an important pay-off has been understanding how freely to adjoin various kinds of new types to a programming language, and predicates to a base category in logic programming. The construction can be relativised to produce the categeory of predicates generic with respect to a given cartesian or alternation theory, which allows constraints on predicates to be "hard-wired" into the syntactic category over which resolution is carried out. This leads to a new approach to disjunctive and negative goals in logic programming as well as a new and very general way to build in data types and control into the structure of a logic programming interpreyer.

Semantics of polymorphism:

This work was supported under
NSF DMS 90-08052
Title: Semantics of Programming
Amount: $46,000
Period: 8/1/91-1/31/94
Location: University of Pennsylvania
Person Months Support: 2 summer months

as well as under the aforementioned ONR contract.

It involved extensive research into the semantics of polymorphism useful in understanding functional programming languages, in particular, analysis of the notion of "parametricity" in polymorphism as introduced by Reynolds. (Papers with E.S.Bainbridge, A.Carboni, J.Y.Girard, P.Mulry, E.Robinson, G.Rosolini, A.Scedrov, D.Scott, P.J.Scott [107, 111])

Recursive types: ONR and NSF-supported research into problem of recursively defined types for mixed variance. As Plotkin had shown, defining a type on a covariant recursive type-variable is equivalent to asking for an "initial T-algebra". The invention of "algebraically compact categories" [110] succeeded in resolving the problem for mixed variance. These categories are ubiquitous in computer science but entirely novel from the point of view of the pure mathematical origins of category theory.

Manuel Hermenegildo

Verification and Debugging based on Abstract Interpretation.

Abstract Interpretation (Cousot and Cousot 1977, [65]) is a semantics-based approach to program analysis. The associated theory provides a framework for inferring properties of programs by computing fixpoints over semantic equations. These equations are defined over a domain of approximations rather than the concrete domain used by the program. Despite the generalized acceptance of abstract interpretation as an elegant framework for proving the correctness and termination of sophisticated program analyses (including most traditional types of dataflow analysis), it has only relatively recently been applied directly to the construction of analyzers in practical compilers. However, the practical results obtained with this technique in the field of logic programming have been surprisingly good in terms of both accuracy of the program information inferred and efficiency

of the resulting compilers, comparing very favorably with traditional ad-hoc techniques, and with the added advantage of being proved correct. Significant practical results have been obtained for example in automatic parallelization (Bueno, Garcia, Muthukumar, Hermenegildo 1990-95) and optimization of Prolog programs (VanRoy 1992). Recently, some results have been extended to constraint logic programming (Garcia, Hermenegildo, Janssens, Bruynooghe 1997). The results of applying these techniques are that Prolog programs can be compiled to run in a time that is comparably close to equivalent programs written in C, and that significant speedups beyond that can be obtained automatically by exploiting parallelism.

Recently, we have proposed a novel view of specifications as abstract properties (Bueno et al. 96, Hermenegildo et al. 97) through which it is possible to relate the traditional concepts of validation and diagnosis with approximations of semantic fixpoints. This view allows modeling (and implementing) verification of specifications as an abstract interpretation. A clear example that fits this model well is the case of types as specifications and type inference as corresponding verification. Types can be seen as abstract properties and the type inference procedure as an abstract interpretation. This work provides a unified framework for static and dynamic verification of a very general kind of properties. Safe approximate verification (in the sense that some properties may not be verified statically) can coexist with run-time checking techniques. We have implemented a prototype framework for such a system (the CIAO assertion environment), and it has been instantiated for some properties for which abstract interpreters were available. The system is capable of locating errors in code, and pointing up "symptoms" of disagreement with specifications, for this limited set of properties.

Related Grants:

- In ESPRIT Project PRINCE (BR 5264), a framework was developed for analysis of constraint logic programs, based on the technique of abstract interpretation. This framework is generic in that it allows inferring classes of properties of programs by defining for each such class a corresponding abstract domain and a fixed set of operations on this domain. This approach has been applied to infer the types of procedure arguments and the calling modes of procedures.

- In ESPRIT Project PARFORCE (BR 6707), new abstract domains were defined and used to infer properties such as upper and lower bounds on the computational cost of procedure calls. The approach has since been extended to also infer non-failure and determinacy of procedures.

- In ESPRIT Project DISCIPL (LTR 22532), an assertion-based program development environment has been designed. This environment uses the abstract interpreter to infer properties of programs which are checked against partial specifications given by the programmer in the form of assertions. Also, an abstract program specializer has been developed which allows specializing generic programs with respect to (possibly infinite) classes of values, as opposed to the fixed set of concrete values typically used by traditional specializers.

Giorgio Levi

Observables, operational and denotational semantics

The s-semantics approach (Falaschi-Levi-Martelli-Palamidessi 1989, Falaschi-Levi-Martelli-Palamidessi 1993, Bossi-Gabbrielli-Levi-Martelli 1994, Gabbrielli-Levi-Meo 1995, Gabbri elli-Levi-Dore 1995, Gabbrielli-Levi-Meo 1996,Comini-Meo 1997) gives a denotation to a pure logic program starting from the operational property (observable) one wants to model. Examples of observables are computed answers (answer constraints in the constraint logic programming case), partial answers, correct answers, call patterns, resultants and SLD-derivations. The denotation can always be equivalently defined by an operational construction (observables computed by a transition system for pure atomic goals) and a denotational construction (least fixpoint of a T_P-like operator). The denotation is AND-compositional, i.e. it uniquely determines the behaviour for all goals.

The various observables can mutually be related by using abstract interpretation techniques (Comini-Levi-Meo 1995). This has allowed us to systematically derive the (abstract) denotations from the observable and the collecting semantics (SLD-derivations). We have also characterized classes of observables according to their semantic properties (compositionality, precision, relation between denotational and operational semantics). The framework allows us to handle approximate observables useful for static program analysis.

Related Grants

1. (European Community) Esprit Project 6707 (Parallel Formal Computing Environment, PARFORCE) from July 24 1992 to January 23 1996 160,000 Ecus

2. (European Community) EC-Israel Exploratory Collaborative Activity ISC-IL-90-Parforce (Bottom-up Analysis of Logic Programming Languages: Theory, Practice and Applications) from January 1 1995 to June 30 1997 50,000 Ecus

Jim Lipton

Results from ONR supported Research

The work described below was carried out with partial support of the ONR under research grant (ONR)n-00014-95-1-0634 *Relations, Categories and Computation* (1995-97) and the *Computing with Relations* project (ONR grant 4331-001-srp-01) and an ONR travel grant to Cambridge University for the Fall of 1995.

The principal lines of research carried out by PI's Peter Freyd and Jim Lipton and by co-PI's Dan Dougherty, Stacy Finkelstein, and their students were:

1. Logic Programming via (Relational) Rewriting

2. Computing in Allegories.

3. Monads and Data enrichment in logic programming

4. Categorical Foundations for Extended Logic programming

5. The logic of Cartesian and Alternation categories

The main results are contained in a series of papers by the Pi's as well as two theses, one prototype and two dissertations in progress. We will discuss 1-3 below along with a brief description of the research areas just outlined. Items 4 and 5 are discussed in the results from Peter Freyd's precursor research.

Logic Programming via Relational Rewriting The aim of our research was to extend the results obtained under the preceding ONR grant, summed up in our proposal, and published in Lipton and Broome's [157]. Briefly, the point of view put forth in that work was that, using a translation based on theorems of Tarski, Maddux and Freyd, *Logic programs define equational and categorical constraints on relation variables*.For example, a program like

```
connected(X,Y):-edge(X,Y).
connected(X,Y):- connected(X,Z),edge(Z,Y).
```

can be viewed as a specification of a least solution to the equation

$$connected = edge \cup connected; edge,$$

which we call the *translated program* , subject to additional constraints that capture the class of domains over which the equation is to be solved. Queries to the original Prolog program are compiled to relation expressions built from the displayed equation. The search for a solution is carried out via rewriting in the appropriate free relational structure or allegory.

We proposed, in the second phase of the grant, to make the translation more efficient, and to extend it to a more general class of programs, as well as to work out a detailed theory of rewriting in Relational structures (e.g. distributive & division allegories) along the lines already initiated, including a study of termination and CR.

A considerable amount of progress was made. We substantially improved the translation above, defining a *Relational Abstract Machine* for Logic program compilation. We assocated a rewriting system with each program and constraint-set which is CR, and has better termination properties than Prolog depth-first search. The abstract machine is defined and shown correct in [53], submitted to the proceedings of Relmics '97 (Relational methods in Computer Science). In this paper we show how to compile and extend the resulting rewriting system to programs with equational and disequational constraints.

A compiler based on these results was implemented by Emily Chapman, and is described in detail in her thesis [52]. A rewriting engine for the relation calculus, used to execute compiled programs, was implemented by Matthew Rhulen at Wesleyan and formally described and shown correct in [252].

Computing in Allegories

The research described in the preceding section, originally aimed at extending logic programming using the relation calculus, inevitably led to a study of relational rewriting for its own sake as a new paradigm for computing. This was first touched on in the [157] paper. Under the current grant, PI Jim Lipton and co-PI Daniel Dougherty at Wesleyan, together with graduate student Claudio Gutierrez began investigating rewriting systems for the pure theory of allegories, and tabular and

distributive allegories, based on a graph translation idea of Freyd [112], and refined by Freyd in the first phase of this grant. Several SN calculi have been defined in Gutierrez' dissertation providing new proofs of decidability of these fragments, originally established by Freyd.

Dougherty and Gutierrez are currently studying the complexity of these fragments using graph rewriting.

Dale Miller

CURRENT

"An Effective Framework for Implementing Derivation Systems." Funded by NSF, July 15, 1998 to June 30, 2000 for a total of $70,000. CCR-9803971

"Structuring of Proof Search in the Logic Programming Paradigm." Funded by NSF under NSF-INRIA Collaborative Research Program. Total of $18,000 for travel and workshop. Other researcher on this proposal is Pierre Deransart INRIA/Rocquencourt. INT-9412553

PENDING

"A Framework for Specifying, Prototyping, and Reasoning about Programming Languages", submitted October 1998 to the NSF Program for Software Engineering and Languages, CCR, CISE. PIs: Dale Miller and Catuscia Palamidessi. For a total of $315,765 for three years.

Proposal for a French-American Collaboration, "Research on Declarative Programming Languages". Submitted to the National Science Foundation under NSF-INRIA Collaborative Research Program. Travel and workshop funds. Other researchers on this proposal are Peter Lee and Frank Pfenning (CMU) and Yves Bekkers, Charles Consel, Pascal Fradet, Daniel Le Metayer, and Olivier Ridoux (INRIA/Rennes).

COMPLETED

Funds from Consiglio Nazionale delle Ricerche (CNR) to support one month of sabbatical stay in Siena, Italy, January 1997.

Principal investigator on NSF grant, "Concurrency and Proof Theory". Total of $190,031 for three years, October 1993 – September 1996.

Principal investigator with Andre Scedrov on NSF grant, "Proofs as Computations". Total of $222,882 for September 1994 – August 1996. Extended until February 1997.

Principal investigator on ARO on "Formal Methods for Software Engineering" for $900,000 for three years starting Jan 1995. Professors Lee and Gunter are co-PIs.

Principal investigator on ARO on "Formal Methods in Software Engineering: Interfaces between Software Components" for $12,000 for a workshop to be held in 1996. Professors Lee, Gunter, and Tannen are co-PIs.

Funds from Consiglio Nazionale delle Ricerche (CNR) to support 2 months of sabbatical stay in Genoa, Italy, September and October 1996.

Principal investigator on ONR grant, "Programming Language Design and Proof Theory". Total of $195,000 for three years, September 1993 – August 1996.

Principal investigator with Andre Scedrov on NSF grant titled "Analysis and Development of Meta-Logics and Logical Frameworks." for three years, ending December 1995. Supplement for tuition for minority student award received September 1994.

Prinicpal investigator on the ONR grant "The Role of Logic Programming as a Specification Language for Theorem Provers", Office of Naval Research, 1985 - 1988.

Prinicpal investigator with 7 others from Penn, Cornell, and Stanford on NSF-INRIA grant for travel to France. Total of $62,000 for two years, starting May 1989.

Principal investigator with Andre Scedrov on NSF grant titled "Higher-order proof systems." Total of $338,807 for three years, starting July 1987.

A grant from SERC for travel to the UK. Total of $4545 for two years, starting June 1988.

Oege de Moor

St. John's JRF (1991-93)

The purpose of this research fellowship was to establish the foundations of a calculus for program derivation, for deriving functional programs from relational specifications. The starting point was a purely functional calculus, invented by Richard Bird and Lambert Meertens, which has clear categorical foundations. Indeed, the simplicity of the calculus largely derives from the fact that all reasoning is conducted with total functions. This has the disadvantage that many specifications (involving nondeterminism and function converses) cannot be allowed in that calculus.

Using Freyd's theory of allegories, De Moor showed how the mathematical structures important to functional programming (in particular inductive data types) could be generalised in a canonical way to relations. Furthermore, applying the construction twice gives a similar correspondence between relations and specifications as rely/guarantee pairs.

The resulting calculus of relational specifications has been highly successful, and is now being pursued by research teams at Oxford, Eindhoven, Milan, Rio and Cape Town. The connection with rely/guarantee pairs has not been as popular, although there is still a steady stream of papers on the subject, mostly from Turku and Queensland (Australia).

Rely/guarantee pairs give a neat model of the combination of angelic and demonic nondeterminism. It has been suggested (by Ralph Back) that this model might be used to give a straightforward yet rigorous semantics to disciplined use of the 'cut' operator in logic programming. If this is indeed possible, it would clearly be highly relevant to the present proposal.

Faculty member of Fujitsu chair (1993-1994)

This 8-month research appointment was used to collect the above research in a textbook, which has since appeared as a special Volume 100 in the well-known "red-and-white" Prentice Hall series.

Visiting Fellow, Chalmers University (1994)

This 4-month research appointment was used to implement an extension of the functional programming language Haskell, to allow a certain kind of higher-order polymorphism. This has since been called "polytypism" by Jeuring and others. It allows one, for example, to write a single function for unification, which is parameterised by the datatype of terms.

During this fellowship I also developed the first generic program for sequential decision processes - which was presented in an invited talk at PLILP '95.

9 Generic Solutions to Optimisation Problems (1996-99)

The objectives of this travel grant are:

- to encapsulate algorithmic paradigms in generic programs
- to unify and relate existing algorithms by expressing them as instances of such generic programs
- to use these case studies as a testbed for recent advances in programming language design

This grant has now been running for two years, and Oxford's participation in the present consortium is a natural consequence of the work: it turned out that we need the facilities of extended logic programming languages to realise the above objectives. Certain generic programs, in particular for the branch-and-bound paradigm, can be expressed in modern programming languages using features such as higher-order polymorphism. Although the asymptotic time complexity of these generic solutions is on a par with specialised, hand-written code, the constant factors are high because many little domain-specific optimisations are missed out by current compiler technology. It would be preferable to state such optimisations as annotations on the generic solution: the translation process then requires the power of extended logic programming to explore various ways of applying the optimisations. Standard logic programming does not suffice, because we need, for example, higher-order unification to express common techniques such as the introduction of accumulation parameters in a convenient manner. The goal is to develop a new kind of environment for transformational programming that permits software to be composed from a set of independent design decisions or "intentions", using domain-specific notations and optimization strategies. The specific aim of the Oxford component of the work is to design a meta-language for the environment, within which domain-specific abstractions can be described, implemented and reused.

This project provides a perfect practical complement to the more foundational studies in the present proposal. The overlap is in the study of language features that facilitate meta-programming. While in the collaboration with Microsoft we take a short-term, pragmatic approach, the present proposal seeks more radical solutions in recent extensions of logic programming.

10 A Meta-language for IP (1998-2001)

Oxford University Computing Laboratory recently started a three-year research project in collaboration with Microsoft Research Laboratories. The goal is to develop a new kind of environment for transformational programming that permits software to be composed from a set of

Related Grants

Generic Solutions to Optimisation Problems The objectives of this travel grant are:

- to encapsulate algorithmic paradigms in generic programs

- to unify and relate existing algorithms by expressing them as instances of such generic programs
- to use these case studies as a testbed for recent advances in programming language design

This grant has now been running for two years, and Oxford's participation in the present consortium is a natural consequence of the work: it turned out that we need the facilities of extended logic programming languages to realise the above objectives.

A Meta-language for Intentional Programming Oxford University Computing Laboratory recently started a three-year research project in collaboration with Microsoft Research Laboratories. The goal is to develop a new kind of environment for transformational programming that permits software to be composed from a set of independent design decisions or "intentions", using domain-specific notations and optimization strategies. The specific aim of the Oxford component of the work is to design a meta-language for the environment, within which domain-specific abstractions can be described, implemented and reused.

This project provides a perfect practical complement to the more foundational studies in the present proposal. The overlap is in the study of language features that facilitate meta-programming. While in the collaboration with Microsoft we take a short-term, pragmatic approach, the present proposal seeks more radical solutions in recent extensions of logic programming.

Gopalan Nadathur

Prior NSF Supported Research

Nadathur has obtained funding for related work from four previous NSF grants extending over a period of eight years. The first of these was NSF Grant IRI-88-05696 entitled "Extending the Domain of Logic Programming," obtained jointly with Donald W. Loveland. The grant amount was $91,033 with active period Aug 1, 1988 - July 31, 1989. The second was NSF Grant CCR-89-05825 entitled "Higher-Order Metalanguages for Implementing Derivation Systems," which provided $148,556 over the period Jan 1990 - Jan 1992. The third one, entitled "Towards Practical Higher-Order Metalanguages," was originally numbered CCR-92-08465 but changed to CCR-9596119 on being moved to the University of Chicago. This grant provided a total funding of $205,222 during the period July 15, 1993 - Dec 31, 1997. The final grant is an ongoing one numbered CCR-9803849 and entitled "An effective framework for realizing derivation systems." The projected award value of this grant is $163,000.

The mentioned grants have supported research towards understanding the connections between proof search and computation, devising new programming and specification languages based on this understanding and exploring the implementation and use of these languages. The results obtained specifically under these grants are summarized as follows:

- A particular form of goal-directed provability called *uniform provability* was identified and used to provide a foundation for logic programming [188]. This work also described the full logic underlying λProlog. The foundational work has been quite influential, its general approach being adopted by other researchers towards a principled addition of other capabilities

to logic programming. In recent work, Nadathur have used the uniform proof notion to analyze disjunctive logic programming [163, 207] and to describe proof procedures for classical logic [202, 203].

- Programming language related features of λProlog such as its type system [214] and its higher-order capabilities [212] have been studied and exposed.

- Questions relevant to providing an efficient and robust implementation of the new features of λProlog have been investigated. A new notation has been developed for lambda terms that supports their deployment in λProlog as representational devices [217, 219] and the use of this notation in realizing intensional operations such as comparison and unification [201] has been studied. Along other dimensions, treatments have been provided for types [153], scoping primitives [140, 199, 205], higher-order aspects [204, 206] and modules [152, 216]. All these ideas have been embedded in an abstract machine that has been designed for the overall language.

- An implementation of λProlog that uses the above mentioned ideas has been developed [213]. This implementation is based on a software emulator for the abstract machine and a compiler for translating λProlog programs into instruction sequences for this machine. This implementation has several important attributes, including the fact that it supports the separate compilation of modular program units.

- Significant parts of the described implementation have been verified. An especially noteworthy aspect is the verification of a large portion of the abstract machine structure, a task carried out in the doctoral dissertation of Keehang Kwon [150, 151].

The grants have supported two undergraduate and four graduate students, the latter being Keehang Kwon, Paul Szymkiewicz, Guanshan Tong and Debra Wilson. Kwon and Tong have obtained doctoral degrees from Duke University and University of Chicago, respectively. Kwon is currently employed as an Assistant Professor at DongA University, Korea and Tong is a member of the staff at Santa Cruz Operation (SCO), New Jersey. Szymkiewicz and Wilson have obtained masters degrees from Duke University. Along a different direction, Nadathur has developed a course entitled "Higher-Order Logic Programming" that covers a spectrum of topics related to the research funded by the NSF grants. This course has been offered at both graduate and advanced undergraduate levels at the University of Munich and the University of Chicago and will also provide material for a series of expository lectures to be given at ESSLLI 1999 in Utrecht, Netherlands.

Michael O'Donnell

Michael O'Donnell's latest grant relevant to this proposal was *Theory and Implementation of Equational Logic Programming* ($161,136) National Science Foundation, 1991–1993, which produced the following results.

The Equational Logic Programming group at the University of Chicago performed substantial research on improvements to the utility and performance of equational compilers. We improved a compiler for Equational Logic Programming (ELP), implemented under a previous grant, and also wrote a completely new compiler for ELP.

We also worked on full-text information retrieval and semantics for constructive logic, we began a new study of digital sound synthesis, and we founded a scholarly journal on the InterNet.

Besides Prof. O'Donnell, the grant supported research by a postdoctoral associate from the People's Republic of China, four doctoral students who completed dissertations with Prof. O'Donnell, and two college students.

Equational logic programs are sets of equations; their computations are highly disciplined proofs using the given equations as hypotheses. The definition of equational logic programming is analogous to Prolog, which computes by theorem proving in the predicate calculus, but the style of programming in ELP is essentially lazy functional programming. The ELP project produced and studied several experimental compilers, including a portable version compiling to C, and a fast interactive version compiling to DEC MIPS and Sun SPARC. We adapted congruence closure—a powerful technique for avoiding recomputation of values that are already computed, previously used for theorem proving—as a new technique for efficient computation in lazy functional programs. The project also developed new optimization techniques for Prolog based on compile-time abstract evaluation.

We designed and implemented *term tours,* a novel protocol for lazy functional Input/Output, previously one of the least satisfactory aspects of functional programming. *Term tours* also allow functional programs to interact naturally with conventional procedural programs. We used lazy functional programming to implement a full-text information-retrieval system for the 2,000-book ARTFL database of French literature.

The project made theoretical advances in term-rewriting systems, including new techniques for determining the most efficient sequence in which to evaluate an expression, and more general criteria for guaranteeing the uniqueness of the final result under different evaluation orders.

The project produced 1 Ph.D. dissertation, 2 refereed journal articles, 10 refereed conference articles, and 2 invited conference presentations during the funding period.

Catuscia Palamidessi

The Human Capital and Mobility project EXPRESS Funded by: the European Economic Community Contract number: CHRX-CT93-0406 Signed in: December 93 Starting Date: January 94 Duration: 4 Years (3 in the original contract + 1 extension) Total budget: 470,000 ECU (about 540,000 USD). Structure of the network: 3 main sites and 6 subcontractors.

— More detailed description:

EXPRESS was a Human Capital and Mobility project (i.e. a Cooperation Network involving several sites) funded by the European Union. EXPRESS aimed at investigating the Expressiveness of Formalisms for Computing. More precisely, the purpose was to achieve a general understanding of the interconnections and relations between formal systems, ranging from programming languages to related axiom systems or rewrite systems. One of the key parts of the project was the systematic study and development of formal methods to compare programming concepts on the basis of their relative expressive power. Such methods provide a tool to classify the variety of programming languages, and therefore a formal basis for the design and implementation of programming languages. Similarly, the project aimed at developing tools for comparing the formal methods for specification and verification developed within the various programming paradigms.

The project EXPRESS started in 1994 and had a duration of four years. The total budget was 470,000 ECU (about 540,000 USD). The Network was composed by the following main sites: the

Centre for Mathematics and Computer Science, with co-PI's Jan Willem Klop and Frits Vaandrager (NL), the University of Genova, with PI Catuscia Palamidessi (IT), the University of Hildesheim, with PI Eike Best (DE). The other associated (sub-contractor) sites are: the University of Roma, with PI Rocco De Nicola (IT), the University of Utrecht, with PI Frank de Boer (NL), the Swedish Institute of Computer Science, with PI Joachim Parrow (SE), the INRIA Sophia-Antipolis, with PI Ilaria Castellani (FR), the INRIA Rennes, with PI Philippe Darondeau (FR), and the University of Sussex, with PI Mathew Hennessy (UK).

The project has sponsored visits among the various sites, and also with sites outside the network, whenever relevant for EXPRESS. It has sponsored participation in conferences for presenting results achieved in the scope of EXPRESS, and has funded the organization of three workshops and one conference: The EXPRESS workshops were organized by Frits Vaandrager (Amsterdam, NL, 1994), by Rocco de Nicola (Tarquinia, IT, 1995), and by Ursula Goltz (Dagstuhl, DE, 1996). The EXPRESS conference was chaired by Catuscia Palamidessi and Joachim Parrow (Santa Margherita Ligure, IT, 1997). The proceedings of this conference were published on *ENTCS* [229] and a selection of the best papers will be published on *Information and Computation* [230].

The project has also sponsored a Post-doc program, allowing the following long-term visits: E. Marchiori at CWI for 6 months, D. Kullmann at CWI for 3 months, A. Mader at KUN for 12 months, J. Rathke at the University of Genova for 3 months, A. Schmitt at the University of Roma for 6 months, A. Huhn at INRIA S.A. for 3 months, W. Thielecke at INRIA S.A. for 3 months, J. Rathke at INRIA S.A. for 3 months, F. Raamsdonk at INRIA S.A. for 3 months, C. Veglioni at the University of Amsterdam for 3 months, and F. Oliver at INRIA R. for 5 months.

The project has been quite successful in raising the interest of scientists on the topic of expressiveness. Even after the end of the project, the EXPRESS meetings are continuing along the line of EXPRESS'97, i.e. as conferences with an open call for papers, invited speakers, panels, etc. This year (1998) the EXPRESS conference was chaired by Catuscia Palamidessi and Ilaria Castellani, and was held in Sophia-Antipolis (FR). Next year it will be chaired by Ilaria Castellani and Biörn Victor, and will be held in Eindhoven (NL).

Research carried on by Catuscia Palamidessi in the context of the EXPRESS project

Formalization of the concept of expressive power for concurrent languages. It is a matter of fact that up to now there is no general formalization of the concept of expressiveness of programming languages. In the theory of formal languages there is a well-established classification, which has at the top the Turing-complete formalisms. However this classification is too coarse: All "reasonable" programming languages are Turing-equivalent. On the other hand, it is clear that there are important differences between programming languages, otherwise we would not have invented so many of them. Obviously a formal theory of "expressiveness" would be extremely valuable, both for design and implementation issues.

In the setting of concurrent languages, this problem becomes even more challenging, because the intuitive gaps in expressive power between the various proposals existing in literature are even more dramatic.

A first step towards the definition of a notion of expressiveness for concurrent languages, and the development of a framework for comparing the relative expressive power, was done in [68], where the following definition was proposed: L can express L' if there exists a translation (compiler) $\mathcal{C}$ from

L' to L *homomorphic* w.r.t. certain operators, and a function $\mathcal{D}$ (decoder) from the observables $\mathcal{O}$ of L to the observables $\mathcal{O}'$, *preserving certain termination modes* (proper termination, deadlock or failure, non-termination), such that for every $P \in L'$, $\mathcal{D}\mathcal{O}\mathcal{C}P = \mathcal{O}'P$ holds.

This framework was used to show various separation results among concurrent constraint languages, among (synchronous) CSP and Asynchronous CSP, and among various CSP dialects.

Investigation and comparison of various versions of Concurrent Constraint Programming. In [91] various fragments of CCP have been investigated. For some of them a very simple semantics, based on closure operators, has been developed. This semantics can be regarded as an approximated semantics of (full) CCP, and be used as a basis for static analysis [94, 91, 95], by means of *abstract interpretation* techniques. Furthermore, for some of these fragments it has been possible to develop a system for proving partial correctness [67].

First steps towards Distributed Constraint Programming. In [35] a new notion of constraint system has been investigated, with the aim of providing foundations for a distributable variant of CCP [34, 228].

Comparison of the expressive power of synchronous and asynchronous communication. In [227] it has been shown that it is not possible to encode the π-calculus [190] into the Asynchronous π-calculus (a subset of the π-calculus which has been implemented by B. Pierce and D. Turner, [43, 130, 222]). This result was quite surprising, since several publications in well respected journals and conference proceedings had conjectured that the two languages were equivalent.