INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen
Oettingenstraße 67, D–80538 München

Ludwig
Maximilians–
Universität
München

Logic implemented Functionally

Norbert Eisinger, Tim Geisler, and Sven Panne

appeared in *Proc. Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP)*,
Springer LNCS, 1997
http://www.pms.informatik.uni-muenchen.de/publikationen
Forschungsbericht/Research Report PMS-FB-1997-2, June 1997

Logic implemented Functionally

Norbert Eisinger, Tim Geisler, and Sven Panne

Institut für Informatik, Universität München,
Oettingenstr. 67, D-80538 München, Germany

Abstract. We describe a course intended to introduce second-year undergraduates to medium-scale programming. The project of the course is to implement a nonconventional logic programming language using a functional implementation language. This exercise reinforces two declarative paradigms and puts the students in experimental touch with a wide range of standard computer science concepts. Declarativity is decisive in making this wide range possible.

Keywords. programming course, functional programming, disjunctive logic programming, model generation

1 Introduction

The undergraduate curriculum at our department contains a practical programming course, the major purpose of which is to let the students experience medium-scale programming in their second undergraduate year. Both, the problem to be implemented and the implementation language can be chosen to the taste of the organizers, taking into account what the students learned during their first year.

For our instance of the course, we chose the problem to implement a nonconventional logic programming language using a functional language. The implementation language was Scheme, which had been the first language for the majority of our students. The inference engine for the logic programming language was nonconventional in that it was not restricted to the Horn case and it allowed the derivation of models for a specification.

The implementation of scanner, parser, and inference engine introduced students to a wealth of computer science concepts in an experimental way, emphasizing two declarative paradigms at the same time. The logic programming paradigm was covered both by implementing a logic programming language, which touched upon topics from foundations of logic programming [16], disjunctive logic programming [17], deductive databases [4,2], partial evaluation [7], and others, and by using the implemented system for a wide range of applications from simple theorem proving problems [27,24] to model-based diagnosis [12]. The functional paradigm was covered by implementing in a functional language using a purely functional style and typical programming techniques [8].

The next four sections of this paper describe the curriculum, the problem setting, our audience, and the didactic goals of the course in more detail. The subsequent core section then presents the actual course description. Finally we report about our experiences and consequences.

2 The Undergraduate Computer Science Curriculum at Universität München

The undergraduate curriculum covers a period of normally 4 semesters, that is 2 years. Computer science undergraduates are supposed to take the following courses, apart from mathematics courses and courses for their minor fields.

semester	course	contents
1	*Computer Science I*	introduction to programming
2	*Computer Science II*	algorithms and data structures
3	*Computer Science III* *Technical Fundamentals* *Programming Lab*	machine-oriented progr., op. systems digital circuits, Boolean algebra a larger programming project
4	*Computer Science IV* *Seminar*	theoretical foundations selected computer science topics

In this paper we discuss the *Programming Lab* scheduled for the 3rd semester. Its purpose is to let the students work on some medium-scale, pre-structured programming project, thus consolidating, reinforcing and extending skills they acquired during the first two semesters. The *Programming Lab* is the first (and often the only) experience of undergraduate students with programming beyond small scale, and it can therefore be quite influential in forming their programming habits.

Actually, there are alternatives to the *Programming Lab* not listed in the table above, but more than 50% of each age-class typically choose this course.

One of the difficulties in designing such a practical programming course is that in spite of the seemingly well-defined curriculum the participants have rather heterogeneous backgrounds.

This is due to two idiosyncrasies of the German academic system. First, professors are not subject to any directives concerning the contents of their teaching. As the staff take turns giving the undergraduate courses, the contents usually vary. In particular, the programming languages and even the programming paradigms used in *Computer Science I* and *Computer Science II* may be different every year. Second, students are rather free when to attend which course and when to take which examination, as long as they complete all achievement tests before a certain time limit. Many students postpone or repeat courses.

Hence a course scheduled for the 3rd semester is usually attended by a majority of 3rd-semester students, but also by a substantial percentage of 5th-semester and higher-semester students who were molded on different paradigms.

An instance of the *Programming Lab* is offered every year. Instances given by different lecturers usually differ significantly, but when it is the same person's turn again, an instance may be repeated with little or no change.

3 The Problem Setting

For our instance of the *Programming Lab* we defined the following project. The students have to implement a declarative logic programming language, complete with scanner, parser, and a nonconventional inference engine. The system to be implemented takes as input a text representing a specification, and produces as output a representation of the models of the specification. Moreover, a number of application problems are to be formalized as specifications and solved using the implemented system.

The inference engine is based on the model generation approach [18], more specifically on the technique of positive unit hyper-resolution tableaux, or PUHR tableaux for short [3]. Apart from being able to detect the unsatisfiability of a specification, it can also derive models of a satisfiable specification. This ability transcends the power of SLD-resolution and its variants.

A *specification* is a finite set of rules. Rules are clauses in implication form $A_1 \wedge \cdots \wedge A_n \longrightarrow B_1 \vee \cdots \vee B_m$ with atomic formulae A_i and B_j. We call the conjunction $A_1 \wedge \cdots \wedge A_n$ the *body* and the disjunction $B_1 \vee \cdots \vee B_m$ the *head* of the rule. An empty body can be understood as *true*, and a rule with an empty body corresponds to a fact. An empty head can be understood as *false*, and a rule with an empty head corresponds to an integrity constraint. A *non-Horn* rule, that is a rule with more than one atomic formula in its head, gives rise to alternative solutions. Thus, the specification language is of higher expressive power than the Horn fragment of first-order predicate logic that is usually used in logic programming.

As an example consider the specification in Fig. 1. We use Prolog's lexical convention that variable names start with capital letters and other identifiers with lower case letters.

```
learning(Person)  ⟶  successful(Person).
student(Person)  ∧  at(Person,uni)  ⟶  playing(Person)  ∨  learning(Person).
lecturer(Person)  ∧  at(Person,uni)  ⟶  teaching(Person)  ∨  researching(Person).
computerscientist(Person)  ∧  playing(Person)  ⟶  .
⟶  student(worf).
⟶  computerscientist(worf).
⟶  at(worf,uni).
```

Fig. 1. Sample specification.

Interpretations (more precisely, Herbrand interpretations) are represented by the set of ground atomic formulae they satisfy. The following interpretation is a model of the specification:

```
student(worf)
computerscientist(worf)
at(worf,uni)
learning(worf)
successful(worf)
```

Declaratively, this means that the model satisfies each ground instance of each rule in the specification. An interpretation satisfies a ground rule iff it does not violate it. An interpretation violates a ground rule iff it contains all atomic formulae from the body but none from the head of the rule.

Operationally, models are obtained by a forward reasoning process, which iterates an adapted version of the *immediate consequence operator* $T_P(I)$ defined for Horn clause programs [16], until it reaches a fixpoint. The process is illustrated in Fig. 2.

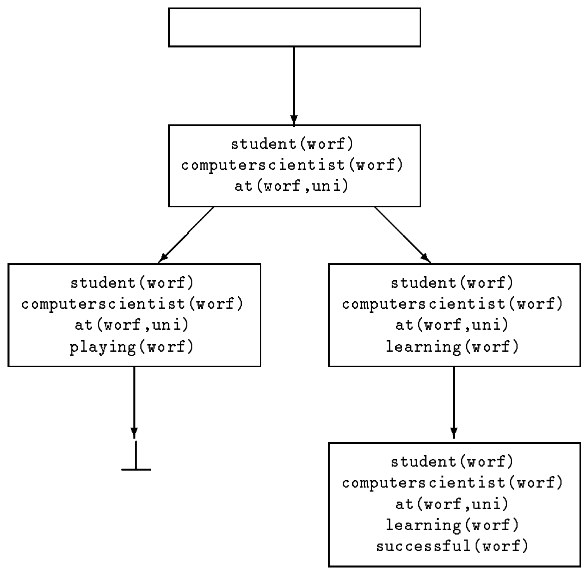

Fig. 2. Development of interpretations for the sample specification.

The inference engine starts with the empty interpretation, which violates the three rules with empty body. In order to satisfy them, the interpretation is extended by their heads.

At this stage the interpretation violates the instance of the second rule `student(worf)` $\wedge$ `at(worf,uni)` $\longrightarrow$ `playing(worf)` $\vee$ `learning(worf)`. In order to satisfy it, the interpretation must be extended by the disjunction `playing(worf)` $\vee$ `learning(worf)`. In order to satisfy a disjunction, the interpretation can be extended by either of the atomic formulae in the disjunction, thus defining separate branches of the search space. (Further branches inserting several of them simultaneously would be redundant.)

In the first branch, the extended interpretation violates the rule with empty head (which states, perhaps contrary to reality, that computer scientists just don't play). If an interpretation violates a rule with an empty head, so does any extension of the interpretation. Thus the current interpretation cannot lead to a model and the branch is *closed*.

In the other branch, the interpretation violates the first rule and is therefore extended by `successful(worf)`. Now all rules are satisfied and the current interpretation is a model of the specification.

This specification happens to have exactly one model. With the additional rule `successful(worf)`$\longrightarrow$ in the specification, the second branch would also be closed, and the specification would be unsatisfiable. If we replaced the rule $\longrightarrow$`student(worf)` by the rule $\longrightarrow$`lecturer(worf)`, the inference engine would derive two models:

`lecturer(worf)`
`computerscientist(worf)`
`at(worf,uni)`
`teaching(worf)`
	`lecturer(worf)`
	`computerscientist(worf)`
	`at(worf,uni)`
	`researching(worf)`

In this way the inference engine works properly, provided that the specifications are *range-restricted*: each variable in the head of a rule also occurs in its body. For more details and theoretical results about PUHR tableaux see [18,3].

The participants of the *Programming Lab* did not have to read any literature about PUHR tableaux. They were given an introduction that was even more informal than this section, just for a general idea of the expected behavior of their inference engine. Forward reasoning appears to be sufficiently intuitive for this approach. We supplied additional information as needed during the course.

4 Our Audience

We had about 40 participants in our instance of the *Programming Lab*. The 3rd-semester students had attended *Computer Science I* based on Scheme and the Abelson/Sussman textbook [1] and *Computer Science II* based on Modula-2 and Wirth's textbook [26] on algorithms and data structures. Parallel to the *Programming Lab*, they got an insight into the structure of compilers from *Computer Science III* and an idea of (propositional) logic from *Technical Fundamentals* and some math courses. The higher-semester participants all knew Modula-2, and some of them also ML.

Thus, most of the students were familiar with the Abelson/Sussman philosophy of *learning by implementing.* Our own experience with this philosophy was very encouraging, so we tried to adhere to it in the *Programming Lab* as well.

5 Didactic Goals of the Course

The overall didactic goals of the *Programming Lab* are as follows:

- consolidation of what students learned during their first two semesters;
- exposing them to computer science concepts that may be new to them;
- experience of medium-scale programming.

As to consolidation, a number of techniques used in the *Programming Lab*, for instance higher-order functions and continuation-passing, reinforced skills the majority of participants had acquired in *Computer Science I*, while others, such as lazy list processing by streams and syntactic abstraction by macros [8], filled gaps omitted by previous courses.

As to exposing the students to concepts they did not know yet, we were rather ambitious.

Foremost, the project introduced another declarative paradigm, logic programming. The students practiced this paradigm both by implementing a logic programming language and by applying it to a wide range of examples. These examples were essential to ensure the acceptance of the new paradigm. They also provided an insight into some important application areas of logic. Several of them were selected from the TPTP benchmark collection [24].

Further, quite a lot of standard computer science concepts appeared in the project. From compiler construction, there were syntax definition, finite automata and the attainment of their determinism, scanner, recursive-descent parsing, abstract syntax trees, etc. From logic programming and automated deduction, there were propositional and first-order syntax and semantics, matching and unification, immediate consequence operator $T_P(I)$, fixpoint, (un)satisfiability, model, refutation, formalization techniques, etc. From deductive databases, there were range-restriction, materialization, incremental forward reasoning by Δ-iteration, integrity constraints, etc. From functional programming techniques, there were higher-order functions, currying, continuation passing, delayed evaluation, implementation techniques for automata, efficient functional data structures, etc. Our purpose was to let the students work with such concepts to the extent required by their task, but not to emphasize the theoretical background. We expect that they benefit from the *Programming Lab* experience when they fully learn this background in later graduate courses taught by our group, such as Higher-Level Programming Languages, Compiler Construction, Logic Programming Techniques, Theory of Logic Programming, Automated Deduction, Deductive Databases, or Knowledge-Based Systems.

Finally, the students were also put in touch with a number of advanced topics, such as model generation, generation of minimal models, incremental and partial evaluation. Here our intention was to prepare the ground for training young talents in our own fields of interest.

As to medium-scale programming, the project was of nontrivial size and required careful structuring and distribution of the work load. We predefined most of the structure, such that the students could get to know the structuring techniques by filling in appropriate gaps. The students were grouped into teams of three, and each team had to implement the same assignments. Students in a team were expected to experience and solve typical communication and coordination problems. We made sure that at least one member of each team knew Scheme and that the team members complemented each other's background as far as possible, hoping thus to help the higher-semester students catch up on the functional paradigm more easily. We also encouraged the use of version control systems and similar tools supporting group work.

6 Course Description

6.1 Organization of the Course

The participants in our instance of the *Programming Lab* are expected to have a working knowledge of the functional programming language Scheme including techniques for data abstraction and the usage of higher-order functions.

Recommended references for questions on Scheme are the Abelson/Sussman textbook [1] and the Scheme report [6]. Unfortunately, no existing textbook covers all the course material. Therefore, we used detailed slides, which were also accessible to and printable by the students via the World Wide Web.

The course runs for 14 weeks, partitioned into seven two-week blocks. Each block starts with a three hour lecture that presents new material and the assignments for the next two weeks.

Assessments are based on sessions led by a teaching assistant at the end of each block, where students present, explain and demonstrate their solutions. We did not plan additional examinations, although this would be possible.

The total time to be spent by a student for this *Programming Lab* is estimated at about eight hours per week.

We used the public domain Scheme implementation Scheme48 rather than MIT Scheme, which had been used in *Computer Science I*, because it is much smaller, more portable and easier for the students to install at home.

6.2 Material Partitioned into Seven Blocks

Each block defines implementation assignments and/or application assignments that have to be solved by the students within two weeks. The application assignments become more interesting and more numerous as the power of the inference engine increases in later blocks. The implementation assignments are normally based on a program frame with gaps to be filled by the students and sometimes provide signatures of functions to be implemented. Wherever suitable, they come with appropriate test-beds.

This has several advantages. The program frames establish a certain programming style by setting an example. In many cases, the students can obtain their solutions by analogy once they have understood the program frame. The program frames also define the interfaces and thus ensure the interchangeability of different solutions, including our own. This is important to have reentry points in case a team simply cannot come up with a working solution for an assignment.

Block 1: Sequences of Characters

Presentation

- Scheme warmup by introducing additional language constructs.
- Syntactic abstraction with macros.
- Streams (lazy lists).

Implementation Assignments

- Abstract datatype for sequences of characters.
- Conversion of files or strings to sequences of characters.

Didactic Considerations On the one hand, the students know just the Scheme core language. On the other hand, the students are used to a syntactically rich language, Modula-2. The introduction of the corresponding basic Scheme datatypes, records (as proposed in [8]), input and output with files, and additional control structures like `case` helps to refresh the knowledge of Scheme, to comply with the student's desire for a more convenient language, and to increase the acceptance of Scheme.

The stream framework is well suited to structure large parts of the system to implement: A stream of characters is transformed via many intermediate transformations to a stream of models. Therefore, streams (or lazy lists) are introduced, as well as syntactic abstraction with macros, which is needed to build some Scheme special forms constructing streams.

To facilitate testing, both input from a string and input from a file has to be provided in interface functions.

Block 2: Lexical Analysis

Presentation

- Abstraction and classification of character sequences as tokens.
- Semi-systematic construction of a finite automaton recognizing tokens from an EBNF defining tokens.
- Implementation of finite automata in Scheme.

Implementation Assignments

- Development of EBNF and automaton for tokens needed for the specification language (paperwork).
- Conversion of sequences of characters into sequences of tokens.

Didactic Considerations The major utility of the scanning phase, namely abstraction that leads to a simplification of the subsequent parsing phase, can be illustrated with an ill-formed specification including comments.

The whole development from an EBNF specifying tokens to a functional implementation of a deterministic finite automaton is presented stepwise using a simple example grammar that specifies phone numbers. This should enable the students to solve their assignment by analogy.

The syntax of specifications is rich enough to give an idea of attributes and the computation of attribute values.

The implementation technique for hand-coded scanners (adopted from [8]) is straightforward: States are represented as functions and transitions as function calls. For a language with tail-recursion optimization like Scheme, this is just another, but much cleaner, notation for a 'goto' to a state [23].

Furthermore, the functional implementation of a scanner provides the opportunity to teach the usage of higher-order functions as building blocks.

Block 3: Syntactic Analysis

Presentation

- Extraction of structure from token sequences.
- Semi-systematic construction of a parser.

Implementation Assignment

- Conversion of sequences of tokens into abstract syntax trees, that is, development of a parser for specifications.

Didactic Considerations Again, the whole development from a sequence of tokens to an abstract syntax tree is presented stepwise using a simple example grammar. This grammar specifies a tiny command language with a sequencing and an iteration construct and a single primitive print command.

The implementation technique is similar to that used in lexical analysis, with the only difference that the parser can call itself recursively. Note that with partial evaluation of the higher-order functions the parser can be transformed into a standard recursive-descent parser [8].

The first version of the inference engine developed in the next block can handle only a subset of the specification language. Later blocks will successively extend the power of the inference engine to larger subsets. It would be possible to define scanner and parser only for the initial subset of the language and to extend them in parallel with the inference engine. We did not choose this option, because it would result in a much higher workload for the students. A step-by-step extension of scanner and parser would (painfully) open their eyes to modular design, though.

Block 4: Ground Horn Clauses

Presentation

- Review of syntax, semantics, and pragmatics for specifications without variables and without disjunctions.
- Functioning of the inference engine.
- Result of the inference engine: either failure or a model.

Implementation Assignments

- Equality on abstract syntax.
- Abstract datatype for interpretations.
- Inference engine.

Application Assignment

– Missionaries and cannibals (see [27] for a description).

Didactic Considerations The inference engine for the ground Horn case is rather simple. Syntactic equality can be used instead of matching or unification, and no provisions for more than one model are required. Yet, the inference engine for this restricted case uses the same principle of fixpoint iteration as more general cases and can be implemented by the same continuation passing technique. Its stepwise generalization is possible by local modifications of the implementation and can be assigned to later blocks.

The representation of interpretations may be simply a list of ground atomic formulae. But data abstraction is important, because efficiency problems in later blocks are likely to call for more sophisticated representations.

The missionaries and cannibals problem can be formalized as a ground Horn specification, though somewhat awkwardly with a lot of copy and paste. This provides a splendid motivation for the introduction of variables in the next block. Another motivation is that formalizations without variables do not allow the extraction of the sequence of actions solving the problem.

Block 5: Horn Clauses with Variables

Presentation

– Motivation for using variables: More compact specifications and the possibility of infinite structures.
– New concepts and terminology needed for variables.
– Modifications of the inference engine for variables.

Implementation Assignments

– Abstract datatypes for bindings and substitutions.
– Matching.
– Retrieval of matching formulae.
– Test for range-restrictedness of rules.
– Extension of the inference engine.

Application Assignments

– Simplifying the missionaries and cannibals example by using variables.
– Simple logical puzzle (persons criticizing each other).
– Queries to a simple deductive database about geography.

Didactic Considerations There are two possible paths from an inference engine for ground Horn clauses to an inference engine for general clauses: The first path introduces variables first and disjunctive heads afterwards, the second path reverses the order. In our opinion, the first path allows more interesting applications and makes the pragmatics of programming in logic more elaborate.

In order to implement variables, the concept of bindings and substitutions (which is known from *Computer Science I* as environments) is introduced.

The continuation-passing technique introduced in the previous block can excellently be used to implement matching of an atomic formula with variables against a ground formula, resulting in either a substitution or a failure. The combination of substitutions can be implemented just as easily. The continuation-passing technique avoids the introduction of dummy elements for failure and duplicate tests for failure.

The abstract datatype for interpretations is extended by an operation for the retrieval of matching formulae instead of the simple member test.

The PUHR calculus is only correct for range-restricted rules. Similarly to the variable declaration/usage problem in compiler construction, this property is impossible to handle in the syntactic analysis phase based on context-free grammars. Therefore, a separate phase testing rules for range-restrictedness is necessary to ensure correct input to the inference engine.

Logical puzzles are well suited to exercise the pragmatics of logic: When to use terms and when predicates? They are instructive for this kind of course because they usually exhibit complex problems in a compact form and because their understanding requires no special background knowledge about an application domain. And if they are fun, all the better.

The deductive database example introduces the logical analogon to the operations *join*, *select*, and *project* from relational database theory. Beyond that, the response times with larger databases are a good motivation for more efficient data representation and the incremental inference engine introduced in block 7.

Block 6: Fairness, Built-In Predicates

Presentation

- Fairness is necessary for refutational completeness.
- Implementation of a fair inference engine with a queue of derived rule heads, efficient functional implementation of queues [13].
- Application: Theorem proving in group theory.
- Built-in predicates increase efficiency and prevent the generation of infinite models.
- Integration of built-in predicates into the inference engine.

Implementation Assignments

- Functional queues.
- Fair inference engine.
- Built-in predicates.

Application Assignments

- Theorem proving in group theory.
- Mathematical puzzle (pouring buckets), using built-in arithmetic predicates.
- Missionaries and cannibals, using built-in arithmetic predicates and representing the path of the boat.

Didactic Considerations Two independent themes, fairness and built-in predicates, are covered by this block. They have to be scheduled after the introduction of variables, and might be scheduled after block 7.

Fairness (i.e., no applicable rule instance may be delayed infinitely long) is an important concept occuring not only in automated reasoning, but nearly in all fields of computer science. In addition to that, the simple implementation technique for ensuring fairness in the inference engine, queues, gives the possibility to touch the topics of functional data structures and amortized complexity of algorithms.

A fair inference engine opens up a wide range of theorem proving applications. Simple theorems from group theory, with which the students are familiar from their mathematics courses, are formulated in such a way that they can be directly translated into rules in implication form. With that, students learn theorem proving techniques like refutation proofs, the extraction of proofs from traces, and formalization techniques for algebraic laws (e.g., the formalization of associativity) without requiring much theoretical background such as clausal normal form or Herbrand's theorem.

Built-in predicates for arithmetic in our logic language reflect the omnipresence of built-ins in programming languages and built-in theories in automated reasoning. Efficiency and more natural formalizations of applications are good motivations for introducing them. The integration of built-in predicates into the inference engine is straightforward. Furthermore, their implementation gives the possibility to touch the topic of constraint reasoning.

The missionaries and cannibals example links the two topics: With fairness, the path of the boat can be collected in an additional argument and thus a solution of the problem can be extracted. With arithmetic, some of the example's parameters can be changed more easily.

Block 7: Incrementality, General Clauses with Disjunctions

Presentation

- Avoiding redundant work with an incremental inference engine.
- Rules with disjunctive heads.
- Modifications of the inference engine.
- Result of the inference engine: A stream of models.
- Application: Model-based diagnosis of digital circuits.
- Application: Multiplication tables of quasigroups.

Implementation Assignments

- Incremental inference engine.
- Inference engine for rules with disjunctive heads.
- Minimal models.

Application Assignments

- Modeling and diagnosis of some digital circuits.
- Formalization of some quasigroup problems.
- Logical puzzle (from a current newspaper).
- Theorem proving in number theory.

Didactic Considerations Again, two independent extensions to the inference engine are discussed in this block: Incrementality and rules with disjunctive heads.

The inference engine implemented in previous blocks suffers from an efficiency problem: In every iteration, all applicable rule instances have to be recomputed. This can be avoided by an incremental algorithm, which is known from deductive databases as *Δ-iteration*: Compute only the rule instances that became applicable due to the atomic formula added to the interpretation most recently. Surprisingly, this technique can be implemented straightforward and efficiently: Instead of using the original rules to determine the next applicable rule instance, use the rules obtained by combining the added atomic formula with the original rules in all possible ways. This *partial evaluation* typically results in a much smaller set of rules.

Up to now, the inference engine either failed or returned just one model, which was implemented with a continuation-passing technique. For rules with disjunctive heads, more than one model may be returned. A stream-based implementation of this behavior, as proposed in [14,25,1], is more suitable and more easy to understand than a solution based on the continuation-passing technique. By transforming the old inference engine to a stream-based implementation, the students learn that these two organizational principles are just two different points of view. Furthermore, the students again practice the stream paradigm and become acquainted with the virtues of delayed (or lazy) evaluation.

Model-based diagnosis [12] of digital circuits seems to be the most realistic application. As an example known from *Technical Fundamentals*, the formalization of the correct and incorrect behavior of a simple half-adder circuit is presented. As an assignment, the students have to formalize a full-adder circuit. With this example, a clever formalization can make evident the compositionality of a declarative language like logic. To implement a usable interface for this diagnosis application, the parser has to be reused. The concept of minimal diagnoses gives a motivation to implement the extraction of minimal models from the stream of computed models. For this, the procedure described by [3] would fit even better into the stream-based implementation.

The quasigroup problems can help to demonstrate the possibilities of the implemented inference engine—previously open existence problems were solved some years ago with nearly the same techniques [9].

7 Results, Experiences, Consequences

Our instance of the *Programming Lab* confronted the students with many new topics. The logical concepts underlying the inference engine were somewhat demanding for second-year undergraduates. On top of that, our decision to distinguish surface syntax from abstract syntax, unlike Abelson/Sussman's logic programming section 4.4 in [1], dragged in a host of concepts related to scanning and parsing.

Indeed the students confirmed that sometimes they needed more effort to understand the problems than to implement them. However, they did master the concepts, and on hindsight they found the course interesting exactly for this reason. A contributing factor to this judgment may have been that our time planning turned out to be adequate to let them grasp what they needed. We have not got enough feedback yet whether the students benefit from this *Programming Lab* in their graduate courses, nonetheless we think so.

If we have to give the course again, we will probably not change the amount of new material, especially as we now have available new tools that facilitate some presentations [15]. If desired, the material can be reduced by omitting surface syntax altogether or by employing scanner and parser generators [21]. It can be extended by allowing more complex surface syntax, for example operator precedences or general first-order formulae that have to be Skolemized and transformed to conjunctive normal form and perhaps to a range-restricted form.

We feel most confident about making the logic programming paradigm the general theme of the course. It complements the functional paradigm many students get to know early in their education, and together the two declarative paradigms might better counterbalance the strong imperative bias most students entertain nevertheless. However, we do favor a clear distinction between the two paradigms. In addition to that, our ulterior motive is to attract students to our own research interests. This motive could persuade us to include even more specialized topics, such as compilation approaches to the inference engine [22] or efficient representations for sets of terms [11].

Our choice of the implementation language is more debatable. We wanted it to be declarative, and we wanted it to be different from the paradigm to be implemented, for the following reasons.

Since the project requires lots of dynamic data structures, any language that burdens the programmer with memory management problems would be difficult to use. One of our teams attempted a reimplementation in C in order to speed up the inference engine. They failed because of the overwhelming low-level details. This consideration rules out most imperative programming languages (Java or, better yet, Pizza [20] might be an option, though).

We were able to cover such a considerable amount of material, mainly because we could rely on the support of a high-level declarative language, although our pre-structuring certainly helped, too. The students' 'harder to understand than to implement' reaction also means 'easier to implement than to understand' and thus corroborates the claim by other authors [19] that more material and more difficult material can be covered with declarative languages than with imperative languages.

So if it is to be a declarative language, why not implement in a logic programming language, say Prolog? It provides unification, a representation for terms, backtracking, and many other amenities. In fact, an inference engine of the kind we have in mind can be implemented in Prolog in a remarkably concise, simple, and elegant way [18]. However, the point of the *Programming Lab* is to teach students medium-scale programming. With Prolog the danger would have been too high that they simply reuse what the implementation language provides without learning *how* it can be implemented.

What we want, then, is a declarative language, but not a logic programming language. This suggests a functional implementation language.

Among the functional languages, Haskell would have had a number of advantages over Scheme. Being strongly typed, it would have prevented many programming errors at compile time. Its module system would have supported the structuring of the program, whereas Scheme was in this respect a step back from Modula-2, which most students knew well. Abstract data types would have been available without further ado. Streams and other lazy data structures would have been straightforward. Moreover, we noticed that the signatures of Scheme functions we gave to the students were often appropriate for our intended solution only, but became counterintuitive and difficult to handle as students came up with different algorithms (e.g., different nestings of iterations). This inhibition of student creativity could have been avoided by curried definitions of the interfaces, which would have been the natural form with Haskell. After the course, we experimented with a Haskell reimplementation [10] that might be the basis for the next time.

This time we chose Scheme mainly because the students already knew it, and experiences by other lecturers with a previous instance of the *Programming Lab*, where students had to learn C++ and InterViews, strongly discouraged us from introducing a new language. It is not trivial for second-year undergraduates to learn a new programming language. Had we introduced Haskell in this course, too many resources would have had to be allocated for technical teething problems with the language. The same, by the way, would have been true about Prolog.

Even with Scheme, we were not entirely spared such problems. The minority of students whose background was exclusively imperative had enormous trouble adapting to the functional style. With uncanny sureness, they discovered `set!` and the `do` construct straightaway, and then wrote C programs in syntactic disguise. As Clack and Myers [5] observe: once students have learned imperative programming, they find it difficult to escape its grasp. Our experience suggests to require an introduction to functional programming as a mandatory prerequisite. The *Programming Lab* cannot really make up for its lack.

Some minor difficulties were caused by the heterogeneous background of students. The good students complained that the program frames we gave them with the assignments kept them on too tight reins. The weaker students found some of those frames too higher-order for comfort. We tried to keep the balance, but may have to adjust some assignments next time.

Many students prefered to do part of the work on their own computers at home. Therefore the use of version control systems and similar tools became more problematic than we expected. This experience may well be typical for any course emphasizing programming.

The one message every participant got out of this *Programming Lab* is that it pays to spend effort on algorithmic improvements rather than $O(1)$ optimizations. There are ample opportunities to make the inference engine prohibitively inefficient, and the students hardly missed any. They experimented with alternative modifications of their code, and observing the effect on the performance was often an unexpected, but in any case a valuable experience to them, regardless of the fact that it would have been the same in any paradigm.

8 Conclusion

In this paper we described a course that reinforces two declarative paradigms, one through implementing and applying a logic programming language, the other through using a functional implementation language.

The primary concern of the course is programming. Declarativity makes it possible, however, to design the course such that it puts the students in experimental touch with a multitude of computer science concepts.

We hope that this paper helps to reproduce the course in other contexts. It is not necessary to cover as wide a range of topics as we did—we were in the lucky position to benefit from a diversity of backgrounds among our staff. Anyone interested in giving a similar course is welcome to contact the authors.

Acknowledgments

We thank all of our colleagues for many useful discussions. Slim Abdennadher and Mathias Kettner helped in preparing and giving the course. François Bry, Thom Frühwirth, and Ingrid Walter read a preliminary draft of this paper and gave us useful comments for its improvement.

References

1. H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.
2. F. Bry, R. Manthey, and H. Schütz. Deduktive Datenbanken. *KI – Künstliche Intelligenz – Forschung, Entwicklung, Erfahrungen*, 3:17–23, 1996. In German.
3. F. Bry and A. Yahya. Minimal model generation with positive unit hyper-resolution tableaux. In *Proceedings of the 5th Workshop on Theorem Proving with Tableaux and Related Methods*, number 1071 in Lecture Notes in Artificial Intelligence, pages 143–159. Springer-Verlag, 1996.

4. S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer-Verlag, 1990.
5. C. Clack and C. Myers. The dys-functional student. In *Proceedings of the First International Symposium on Functional Languages in Education, FPLE '95*, number 1022 in Lecture Notes in Computer Science, pages 289–309. Springer-Verlag, 1995.
6. W. Clinger and J. Rees. Revised^4 report on the algorithmic language Scheme. *ACM Lisp Pointers IV*, July–Sept. 1991.
7. O. Danvy, editor. *Partial evaluation*. Number 1110 in Lecture Notes in Computer Science. Springer-Verlag, 1996.
8. D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. MIT Press, 1992.
9. M. Fujita, J. Slaney, and F. Bennett. Automatic generation of some results in finite algebra. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence, IJCAI 93*, pages 52–57. Morgan Kaufmann, 1993.
10. T. Geisler, S. Panne, and H. Schütz. Satchmo: The compiling and functional variants. *Journal of Automated Reasoning*, 18(2):227–236, 1997.
11. P. Graf. *Term Indexing*. Number 1053 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1996.
12. W. Hamscher, editor. *Readings in model-based diagnosis*. Morgan Kaufmann, 1992.
13. R. R. Hoogerwoord. A symmetric set of efficient list operations. *Journal of Functional Programming*, 2(4):505–513, 1992.
14. K. M. Kahn and M. Carlsson. How to implement Prolog on a LISP machine. In J. A. Campbell, editor, *Implementations of PROLOG*, pages 117–134. Ellis Horwood, 1984.
15. M. Kettner and N. Eisinger. The tableau browser SNARKS—system description. In W. McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction, CADE-14*, number 1249 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1997.
16. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1984.
17. J. Lobo, J. Minker, and A. Rajasekar. *Foundations of Disjunctive Logic Programming*. MIT Press, 1992.
18. R. Manthey and F. Bry. SATCHMO: A theorem prover implemented in Prolog. In E. Lusk and R. Overbeek, editors, *Proceedings of the 9th International Conference on Automated Deduction, CADE-9*, number 310 in Lecture Notes in Computer Science, pages 415–434. Springer-Verlag, 1988.
19. M. Núñez, P. Palao, and R. Peña. A second year course on data structures based on functional programming. In *Proceedings of the First International Symposium on Functional Languages in Education, FPLE '95*, number 1022 in Lecture Notes in Computer Science, pages 65–84. Springer-Verlag, 1995.
20. M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159. ACM Press, 1997.
21. S. Panne. EAGLE — Ein Generator für erweiterte attribuierte LR(1)-Grammatiken. Diplomarbeit, Universität Erlangen-Nürnberg, IMMD8, December 1994. In German.
22. H. Schütz and T. Geisler. Efficient model generation through compilation. In M. McRobbie and J. Slaney, editors, *Proceedings of the 13th International Conference on Automated Deduction, CADE-13*, number 1104 in Lecture Notes in Artificial Intelligence, pages 433–447. Springer-Verlag, 1996.

23. G. L. Steele Jr. Debunking the “expensive procedure call” myth or, Procedure call implementations considered harmful or, Lambda: The ultimative goto. AI Memo 443, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1977.
24. G. Sutcliffe, C. B. Suttner, and T. Yemenis. The TPTP problem library. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction, CADE-12*, number 814 in Lecture Notes in Artificial Intelligence, pages 252–266. Springer-Verlag, 1994.
25. P. Wadler. How to replace failure by a list of successes. In *Proceedings of the 1985 Conference on Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 113–128. Springer-Verlag, 1985.
26. N. Wirth. *Algorithmen und Datenstrukturen mit Modula-2*. Teubner, 1986. In German.
27. L. A. Wos, R. Overbeek, E. Lusk, and J. Boyle. *Automated Reasoning*. McGraw-Hill, 2nd edition, 1992.